Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Kristjan Sešek

Analiza in implementacija predpomnenja v samostojni Java aplikaciji

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Marko Bajec

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Predpomnjenje je v računalništvu zelo pogosta operacija, ki je realizirana tako na strojnem kot tudi sistemskem in čisto aplikacijskem nivoju. Pripomore namreč k večji učinkovitosti računalniških sistemov. V okviru diplomskega dela preglejte možnosti za predpomnenje v aplikacijah, ki so napisane v jeziku Java. Posebej preverite, kako bi lahko realizirali predpomnenje, kadar gre za uporabo velikega števila trajnih objektov (se sinhronizirajo s podatkovno bazo), ki so med seboj v kompleksnih relacijah. Predlagajte in opišete najustreznejšo rešitev.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Kristjan Sešek sem avtor diplomskega dela z naslovom:

Analiza in implementacija predpomnenja v samostojni Java aplikaciji

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Marka Bajca
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 5. september 2015 Podpis avtorja:

Kazalo

Povzetek

V delu obravnavamo problem predpomnenja trajnih objektov v javanski aplikaciji v okolju Java Enterprise Edition. Najprej spoznamo teorijo predpomnenja in v programskem jeziku Java implementiramo preprost predpomnilnik. V nadaljevanju bolj podrobno spoznamo ogrodje Java Enteprise Edition. Osredotočimo se predvsem na tehnologije za prenos podatkov med javansko aplikacijo in strežnikom in med strežnikom in podatkovno bazo, saj neposredno vplivajo na problem predpomnenja. Na praktičnem primeru prikažemo, kako lahko ob spreminjanju povezanih trajnih objektov predpomnilnik postane nekonsistenten. Ugotovimo, da noben visokonivojski predpomnilnik ne rešuje tega problema, zato razvijemo svojo rešitev, ki ohranja predpomnilnik konsistenten. Rešitev implementiramo v javanski aplikaciji in predstavimo rezultate.

Ključne besede: predpomnilnik, Java, Java EE.

Abstract

Our work discusses problem of persistent object caching in Java application in Java Enterprise Edition environment. We begin with the theory of caching and develop a simple cache in Java programming language. We continue by presenting Java Enterprise Edition framework. We especially focus on technologies that transfer data between Java application and application server and between application server and database, because they directly affect the problem of persistent object caching. We show how changing connected objects can make cache data inconsistent. No other high level cache solves this problem, so we develop our own solution that keeps cache data consistent. We implement the solution in Java application and analyse the results.

Keywords: cache, Java, Java EE.

Poglavje 1

Uvod

V računalništvu se pogosto srečujemo z zahtevnimi operacijami, ki se ponavljajo. Osnovna ideja predpomnenja je, da si rezultate teh operacij za nekaj časa beležimo v manjši, hiter, ločen prostor, imenovan predpomnilnik. Vsak naslednji izračun določene operacije enostavno preberemo iz predpomnilnika.

Preprosto predpomnenje se v brskalniku izvaja vsakič, ko dostopamo do poljubne spletne strani. Naslov spletne strani se preko strežnika DNS prevede v IP-naslov, kjer se nahaja dejanski strežnik. To prevajanje traja nekaj časa, zato brskalnik vsako prevajanje in vsebino spletne strani shrani v predpomnilnik. Brskalnik ob vsakem naslednjem dostopu do že obiskane spletne strani prebere IP-naslov iz predpomnilnika in dostopa direktno do strežnika. Če se vsebina spletne strani ni spremenila, potem jo brskalnik naloži iz predpomnilnika. Uporaba predpomnilnika v brskalniku bistveno pohitri dostop do že obiskanih spletnih strani.

Bolj kompleksne operacije in rezultati zahtevajo bolj napreden predpomnilnik. V sodobnih poslovnih aplikacijah so pogost rezultat operacij kompleksni objekti, ki so med seboj povezani in predpomnenje enega izmed njih vpliva tudi na ostale. Z drugimi besedami, rezultat ene operacije je povezan z rezultati drugih operacij. Preprost primer sta dva objekta, ki sta povezana.

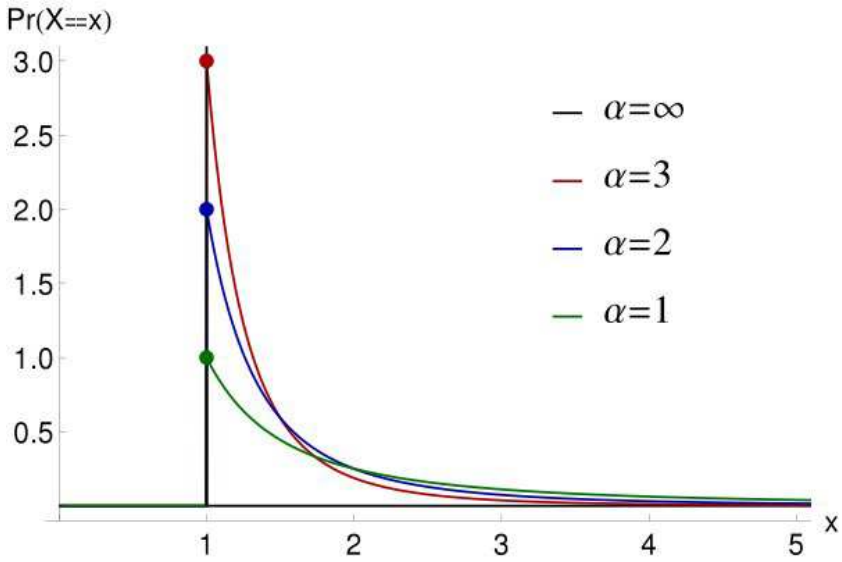
Sprememba na prvem objektu vpliva na drugi objekt, ki ga je potrebno ustrezno posodobiti.

V Javi obstaja veliko programskih knjižnic, ki ponujajo predpomnenje, a nobena izmed njih ne ponuja predpomnenja kompleksnih objektov, ki so odvisni eden od drugega. V diplomski nalogi bom predstavil teorijo višjenivojskega predpomnilnika in opisal problem predpomnenja kompleksnih objektov v javanski aplikaciji.

Poglavje 2

Predpomnenje

Predpomnenje se v računalništvu uporablja povsod. V strojni opremi centralna procesna enota uporablja predpomnilnik, kjer hrani kopije najbolj pogosto dostopanih delov glavnega spomina. Na višjem nivoju spletni brskalnik v predpomnilniku hrani domenska imena, pripadajoče IP-naslove in vsebino spletne strani. Vse to deluje na osnovi Paretove porazdelitve, ki pravi, da se majhen delež celote uporablja veliko bolj pogosto kot preostali del. Slika 2.1 prikazuje porazdelitev neodvisne spremenljivke x pri različnih vrednostih α. Večja kot je vrednost α, bolj pogosto se uporablja manjši delež spremenljivke x. Za primer vzemimo 20 % računalniških operacij, ki se izvajajo 80 % časa. To porazdelitev dobimo pri α=1.16. Če torej izboljšamo čas izvajanja navedenim 20 % operacijam, bo sistem kot celota bolj učinkovit in hitrejši.



Slika 2.1: Paretova porazdelitev

2.1 Predpomnilniški zadetek in zgrešitev

Kadar element, ki ga potrebujemo, najdemo v predpomnilniku, govorimo o zadetku. Neuspešen dostop, ko za iskani ključ v predpomnilniku ne najdemo elementa, imenujemo zgrešitev. Predpomnilnik je smiselno uporabiti, če smo pripravljeni porabiti nekaj več pomnilnika, da izboljšamo hitrost, in če smo prepričani, da bo večina dostopov do predpomnilnika uspešnih.

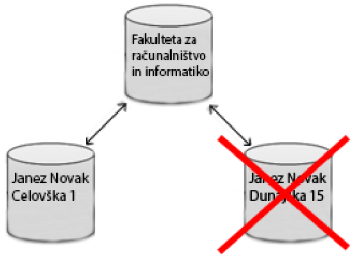
2.2 Konsistentnost predpomnilnika

Ena od pomembnih zahtev predpomnilnika je, da zagotavlja konsistentnost podatkov. Če v predpomnilniku posodobimo objekt X, ki je povezan z objektom Y, potem se mora v isti posodobitvi posodobiti tudi objekt Y. To pomeni, da direkten dostop do objekta X vrne enak rezultat kot dostop do objekta X preko objekta Y. Poskrbeti moramo torej, da se ob vsaki posodobitvi posodobijo vsi povezani objekti. To zagotavlja, da je predpomnilnik konsistenten.

Za primer vzemimo dva povezana objekta, in sicer objekt *fakulteta* in objekt *študent*, kot prikazuje slika 2.2. Oba objekta vstavimo v predpomnilnik.

Slika 2.2: Začetno stanje

Denimo, da se študent preseli in posledično spremeni svoj naslov. V predpomnilnik shranimo posodobljen objekt z novim naslovom. V tem trenutku je objekt *fakulteta* še vedno povezan s starim objektom *študent* in predpomnilnik je nekonsistenten. Objekt *fakulteta* moramo povezati z novim objektom in odstraniti povezavo s starim objektom, kot prikazuje slika 2.3. Predpomnilnik je zopet konsistenten.



Slika 2.3: Stanje po spremembi

2.3 Velikost predpomnilnika

Velikost predpomnilnika je omejena. V naprednejših predpomnilnikih se velikost nastavi programsko z največjim številom elementov v predpomnilniku, na splošno pa je velikost omejena s fizično velikostjo predpomnilnika v računalniku oz. z največjim dovoljenim spominom, ki ga procesu dopušča operacijski sistem. Vedno pa potrebujemo mehanizme za pametno odstranjevanje objektov iz predpomnilnika in zato bomo spoznali naslednje algoritme:

- **algoritem najstarejši najprej** ob vsakem vstavljanju in branju posodobi časovni žig objekta in hrani dostop do najstarejšega objekta v predpomnilniku; ko je predpomnilnik poln, se ta objekt odstrani;
- **algoritem najnovejši najprej** enako kot algoritem najstarejši najprej hrani časovni žig zadnjega dostopa do objekta v predpomnilniku, s to razliko, da ob polnem predpomnilniku najprej odstrani najnovejši objekt;
- **algoritem najmanj pogosto uporabljen najprej** ob vsakem branju objekta poveča števec, ki šteje število dostopov do tega objekta, in posledično hrani dostop do najmanj pogosto uporabljenega objekta v predpomnilniku; ko je predpomnilnik poln, se ta objekt odstrani;
- **algoritem prvi noter, prvi ven** ob vsakem vstavljanju doda objekt v vrsto in posledično hrani dostop do zadnjega objekta v vrsti; ko je predpomnilnik poln, se zadnji objekt odstrani.

V večini predpomnilnikov lahko določimo, kaj se zgodi z odstranjenimi objekti. Ali se shranijo na trdi disk, ali se za v naprej določen čas še ohranjajo v sekundarnem spominu, ali pa se dokončno izbrišejo.

2.4 Sinhronizacija s podatkovno bazo

V visokonivojskem predpomnilniku v večini primerov predpomnimo podatke iz podatkovne baze. Zato bomo spoznali dva predpomnilniška mehanizma,

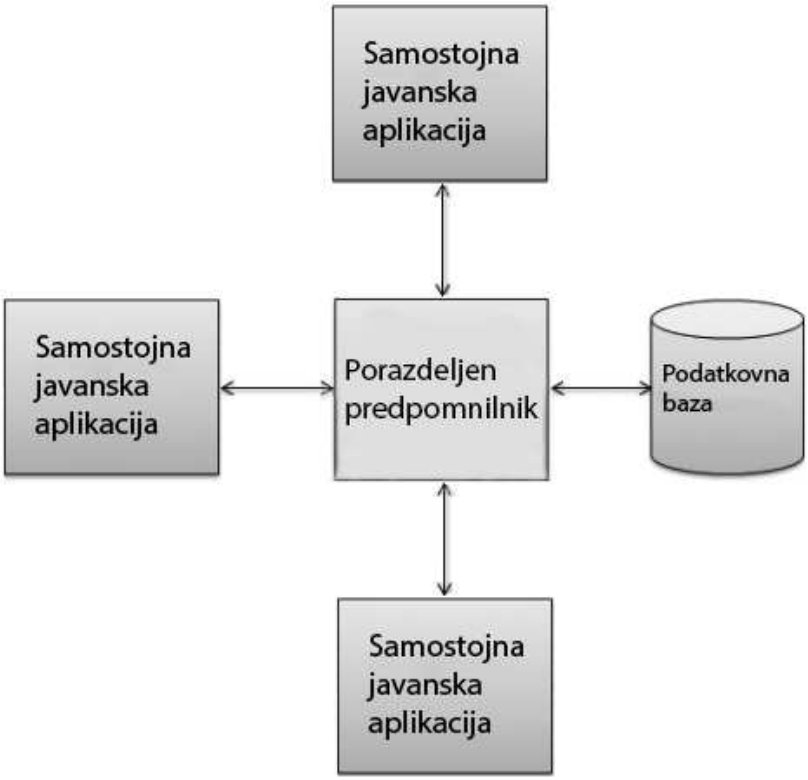
ki ponujata sinhronizacijo podatkov s podatkovno bazo:

- **predpomnilnik s sprotnim zapisovanjem** povežemo s podatkovno bazo in predpomnilnik avtomatsko sinhrono posodablja spremenjene objekte; predpomnilnik se v tem primeru obnaša kot fasada pred podatkovno bazo; ob vsaki spremembi, ki se zgodi v predpomnilniku, se aplikacija zaustavi in počaka, da se sprememba prenese v podatkovno bazo; ko se podatkovna baza posodobi, aplikacija nadaljuje z izvajanjem; pisanje preko predpomnilnika je preprosto konfigurirati, moramo pa se zavedati, da bo vsako dodatno funkcionalnost predpomnilnika zelo težko implementirati;
- **predpomnilnik z zakasnjenim zapisovanjem** je enak kot predpomnilnik s sprotnim zapisovanjem, le da se sprememba predpomnilnika prenese v podatkovno bazo asinhrono v ozadju; to pomeni, da se aplikacija ne zaustavi, ampak se izvaja naprej; glavna prednost pisanja za predpomnilnikom je, da sprosti dostop do podatkovne baze; večkratna posodobitev določene entitete bi sinhrono potrebovala več samostojnih posodobitev oz. transakcij; pri uporabi predpomnilnika z zakasnjenim zapisovanjem se vse posodobitve združijo v eno in izvede se le ena transakcija ob vnaprej določenem času; posodabljanje ob vnaprej določenem času nam omogoča, da izvajamo posodobitve takrat, ko je podatkovna baza najmanj zasedena in s tem naredimo sistem bolj odziven; slaba stran pisanja za predpomnilnikom je časovni zamik in možnost, da predpomnilnik postane nekonsistenten, kar se zgodi, če ima katera izmed kasneje združenih posodobitev napako in je posledično celotna transakcija neuspešna; pisanje za predpomnilnikom uporabljamo le takrat, ko lahko dopustimo, da so podatki v sistemu nekonsistentni.

2.5 Porazdeljeno predpomnenje

Porazdeljeno predpomnenje zahteva povezovanje sistemov ali aplikacij v mrežo. Vsaka aplikacija hrani svoj lokalni predpomnilnik, ki komunicira s predpo-

mnilniki preostalih aplikacij. Aplikacija zahteva nov objekt iz strežnika ali iz aplikacije, ki ima ta objekt že v predpomnilniku. Vsaka sprememba v vsaki aplikaciji je poslana ostalim v mreži. Poznamo dva načina delovanja ob spremembah, in sicer invalidiranje in repliciranje.



Slika 2.4: Porazdeljeno predpomnenje

2.5.1 Invalidiranje

Kadar neka aplikacija spremeni objekt (v svojem predpomnilniku), se podatek o tem, da je prišlo do spremembe, sporoči tudi drugim aplikacijam. Le-te ta objekt odstranijo iz svojega predpomnilnika (saj ne odraža več pravega stanja). Če aplikacija odstranjen objekt spet potrebuje, ga naloži iz strežnika ali iz aplikacije, ki je naredila spremembo. Aplikacije ob spremembi ena drugi sporočajo le enolični identifikator spremenjenega objekta in enolični identifikator aplikacije, ki je naredila spremembo. Tak način delovanja zagotavlja nizko porabo pasovne širine in dobro razširljivost sistema.

2.5.2 Repliciranje

Kadar aplikacija spremeni objekt (v svojem predpomnilniku), ga posreduje tudi drugim aplikacijam. Tak način delovanja potrebuje veliko pasovne širine, saj se čez omrežje pošiljajo celotni spremenjeni objekti. V večini sistemov vsaka aplikacija v mreži aktivno uporablja le svojo podmnožico objektov. To pomeni, da ne potrebujejo vsi enakih objektov v predpomnilniku in repliciranje bi v tem primeru le tratilo procesorsko moč in pasovno širino.

Poglavje 3

Implementacija preprostega predpomnilnika

Najbolj preprosta oblika predpomnenja v Javi je razred *HashMap<key, value>*, kjer kot ključ uporabimo enolični identifikator objekta, kot vrednost pa sam objekt. To predpomnenje ponuja osnovno vstavljanje in hranjenje objektov, dokler niso eksplicitno odstranjeni. Za prikaz delovanja programa je v programskih odsekih uporabljena knjižnica *JUnit* in metoda *assertEquals(string, object1, object2)*, ki pri istih vhodnih objektih izpiše podan niz.

```
Map<Integer, Object> predpomnilnik = new HashMap<>();
predpomnilnik.put(1, "Element1");
assertEquals("Niza sta enaka!", predpomnilnik.get(1), "Element1");
```

Preprost predpomnilnik deluje zelo dobro, dokler ga ne integriramo v večji sistem, ki podatke preko strežnika bere in shranjuje v podatkovno bazo in jih posreduje tudi spletni ali mobilni aplikaciji. Samostojna implementacija takega sistema je časovno zelo potratna, zato so se razvila ogrodja, ki ponujajo najbolj pogosto uporabljene funkcionalnosti, kot so centralizirana poslovna logika, komunikacija s podatkovno bazo, spletnimi storitvami in samostojnimi aplikacijami. Java aplikacija, v kateri želimo implementirati predpomnilnik, pridobiva podatke iz ogrodja Java Enterprise Edition.

Najbolj zahtevna in časovno potratna operacija je prenos podatkov iz aplikacijskega strežnika v Java aplikacijo, zato je smiselno, da prejete podatke shranimo v predpomnilnik. Od tehnologije, ki prenaša podatke, je odvisna implementacija predpomnilnika. Za učinkovito predpomnenje podatkov moramo spoznati tehnologijo za dostop Java aplikacije do strežnika in dostop strežnika do podatkovne baze.

Poglavje 4

Java Enterprise Edition (Java EE)

Ogrodje Java EE je bilo razvito z namenom, da poenostavi in standardizira razvoj poslovnih aplikacij. Sistem poslovnih aplikacij mora biti zanesljiv, varen, razširljiv in mora imeti možnost zunanjega dostopa, kar ogrodje Java EE tudi omogoča. Razvijalec se osredotoči le na implementacijo poslovne logike. Ogrodje je razvito v programskem jeziku Java. To je objektno usmerjen, prenosljiv in brezplačen programski jezik. Ogrodje je del aplikacijskega strežnika in razvijalcu ponuja naslednje tehnologije:

- **Java Servlet**: tehnologija za dinamično ustvarjanje spletnih aplikacij v programskem jeziku Java;
- **JavaServer Pages (JSP)**: tehnologija za dinamično ustvarjanje spletnih aplikacij v jeziku XML in HTML (osnovan na tehnologiji Java Servlet);
- **Remote Method Invocation (RMI)**: tehnologija za komunikacijo Java aplikacij z ostalimi aplikacijami;
- **Java Database Connectivity (JDBC)**: tehnologija za dostop do relacijske podatkovne baze v programskem jeziku Java;
- **Java Message Service API (JMS)**: tehnologija, ki omogoča ustvarjanje, pošiljanje in branje asinhronih sporočil med javanskimi kompo-

nentami;

- **Java Transaction API (JTA)**: tehnologija, ki omogoča uporabo distribuiranih transakcij v programskem jeziku Java;

- **Java Persistence API (JPA)**: specifikacija za objektno-relacijsko preslikovanje v programskem jeziku Java;

- **Enterprise JavaBeans (EJB)**: tehnologija, ki skrbi za življenjski cikel, varnost in transakcije komponent s poslovno logiko;

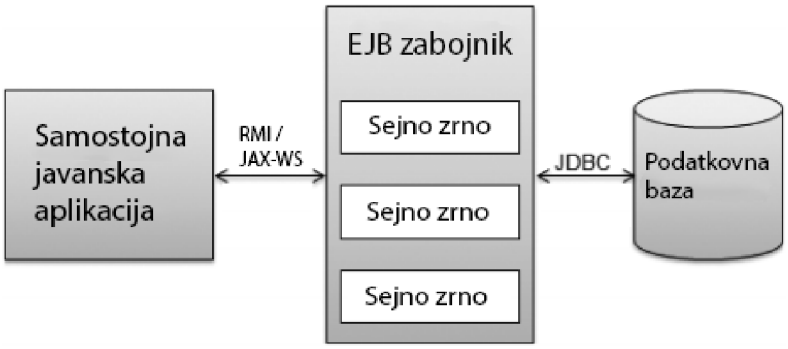
- **Java Web Services (JAX-WS)**: tehnologija, ki omogoča, da izpostavimo poslovno logiko kot spletno storitev.

V nadaljevanju bomo bolj podrobno spoznali tehnologije, ki so kritične za implementacijo učinkovitega predpomnilnika v samostojni Java aplikaciji, saj neposredno vplivajo nanj. Vsako tehnologijo bomo površinsko opisali in se osredotočili na del, ki je pomemben za implementacijo predpomnilnika.

4.1 Enterprise JavaBean (EJB)

Tehnologija EJB predstavlja arhitekturo za razvoj poslovnih aplikacij osnovanih na samovzdrževanih komponentah. Vse komponente so nameščene znotraj EJB zabojnika, ki skrbi za transakcije, varnost, življenjski cikel, sočasni dostop in preprosto razširljivost sistema.

Slika 4.1 prikazuje arhitekturo sistema pri uporabi EJB zabojnika. Samostojna aplikacija lahko do podatkovne baze dostopa le preko sejnih zrn, zato je pomembno, da spoznamo, kako sejna zrna delujejo.



Slika 4.1: Arhitektura sistema pri uporabi EJB zabojnika

V EJB zabojniku je bazen zrn, ki se glede na število zahtev za posamezno zrno ustrezno povečuje ali zmanjšuje. Vsako zrno v bazenu je dodeljeno svoji zahtevi. Ko je zahteva izvedena, se zrno doda nazaj v bazen. Model EJB ponuja uporabo sejnih in sporočilnih zrn. Za nas so pomembna predvsem naslednja sejna zrna:

- *sejno zrno brez stanja* — zrno ne ohranja stanja in se po uporabi vrne v bazen;
- *sejno zrno s stanjem* — zrno ohranja stanje in ostane povezano z aplikacijo, ki ga je zahtevala;
- *sejno zrno edinec* — ob zagonu aplikacije je ustvarjeno eno samo zrno, ki ohranja stanje in ga uporablja več aplikacij.

EJB zrna se najbolj pogosto uporabljajo za izvajanje poslovne logike. Vsaka implementacija zrna vsebuje poslovne metode, ki jih zunanja aplikacija proži. Celotna zrna ali posamezne poslovne metode v zrnu preprosto administriramo z anotacijami in definiramo naslednje mehanizme:

- nivo varnosti
- tip transakcije

- prestreznike, ki se izvedejo pred ali po klicu metode ali ob kreiranju ali uničenju zrna
- način izvajanja (sinhrono ali asinhrono izvajanje)
- način dostopa (lokalni ali oddaljen dostop)
- upravljanje z napakami
- povezovanje komponent med seboj (injiciranje odvisnosti)

Vsi zgoraj navedeni mehanizmi nam zelo olajšajo razvoj, saj z minimalno konfiguracijo poskrbijo za pomembne aspekte poslovne aplikacije. V naslednjem poglavju bomo spoznali, kako se podatki iz podatkovne baze preslikajo v javanske objekte in kako do njih varno dostopamo.

4.2 Java Persistence API (JPA)

Implementacija kompleksnega poslovnega domenskega modela v objektni podatkovni bazi je zelo velik izziv, zato se v ogrodju Java EE primarno uporablja relacijska podatkovna baza. Shranjevanje javanskih objektov v relacijsko podatkovno bazo je zelo zapleteno, saj moramo preslikati graf povezanih objektov v plosko strukturo tabel in stolpcev. Upoštevati moramo dedovanje, preslikati povezane objekte v tuje ključe in preslikati javanske tipe podatkov v pripadajoče podatkovne tipe v jeziku SQL. To zahteva uporabo objektno-relacijskega preslikovanja. To je preslikava javanskega objekta v tabelo v podatkovni bazi, tako da vsak stolpec v tabeli predstavlja točno eno polje oz. atribut v javanskem objektu. Objekti, ki so preslikani v podatkovno bazo, se imenujejo entitete. Entitete so v javanskem razredu definirane z anotacijami.

4.2.1 Anotacije

Anotacija *@Entity* nad definicijo razreda ločuje entiteto od navadnega javanskega objekta. Tehnologija JPA ponuja ogromno anotacij, in sicer od

označevanja polja, ki naj se uporabi za primarni ključ, do označevanja kardinalnosti. Kardinalnost pove, na kakšen način so entitete med seboj povezane. V tehnologiji JPA poznamo štiri vrste kardinalnosti: ena proti ena, ena proti mnogo, mnogo proti ena in mnogo proti mnogo. Pri izbiri kardinalnosti moramo dodati še parameter, ki pove, ali naj se povezana entiteta implicitno naloži. Nalaganje vseh povezanih entitet je zelo časovno in prostorsko zahtevno, zato nam tehnologija JPA omogoča, da sami določimo, katero povezano entiteto želimo naložiti. Če želimo, da se povezana entiteta naloži, jo označimo z anotacijo *FetchType.EAGER*, drugače pa kot *FetchType.LAZY*. To lahko določimo tudi na nivoju aplikacije z globino, do katere naj se povezane entitete nalagajo.

4.2.2 Implementacije

Standard JPA je zbirka vmesnikov, ki podrobno opisujejo, kako implementirati objektno-relacijsko preslikovanje, tako da bo fleksibilno za možne kasnejše spremembe. Najbolj znane implementacije standarda JPA so:

- EclipseLink
- Hibernate
- TopLink Essentials
- KODO
- OpenJPA

Javanska aplikacija, v kateri želimo implementirati predpomnenje, uporablja implementacijo Hibernate.

4.3 Hibernate

Za implementacijo učinkovitega predpomnilnika je pomembno, da spoznamo, kako v sistemu Hibernate delujejo transakcije in kako se entitete nalagajo in shranjujejo v podatkovno bazo.

4.3.1 Transakcije

V ogrodju Java EE poznamo dva mehanizma za administracijo transakcij, in sicer Java Database Connectivity (JDBC) in Java Transaction API (JTA). Sistem Hibernate podpira oba mehanizma in prepušča aplikaciji, da administrira transakcije s podatkovno bazo. Vmesnik *org.hibernate.TransactionFactory* prepozna, kateri transakcijski mehanizem se uporablja in na podlagi tega izdela ustrezen objekt *org.hibernate.Transaction*, s katerim lahko aplikacija upravlja transakcije.
Samostojno upravljanje transakcij nad preslikanimi entitetami je zelo zahtevno, saj bi morali hraniti seznam entitet, na katere vpliva trenutna transakcija, ustrezno koordinirati spremembe preslikanih entitet in ob tem poskrbeti še za možnost sočasnega dostopa do podatkovne baze. Sistem Hibernate je zato uvedel sejo *org.hibernate.Session*.

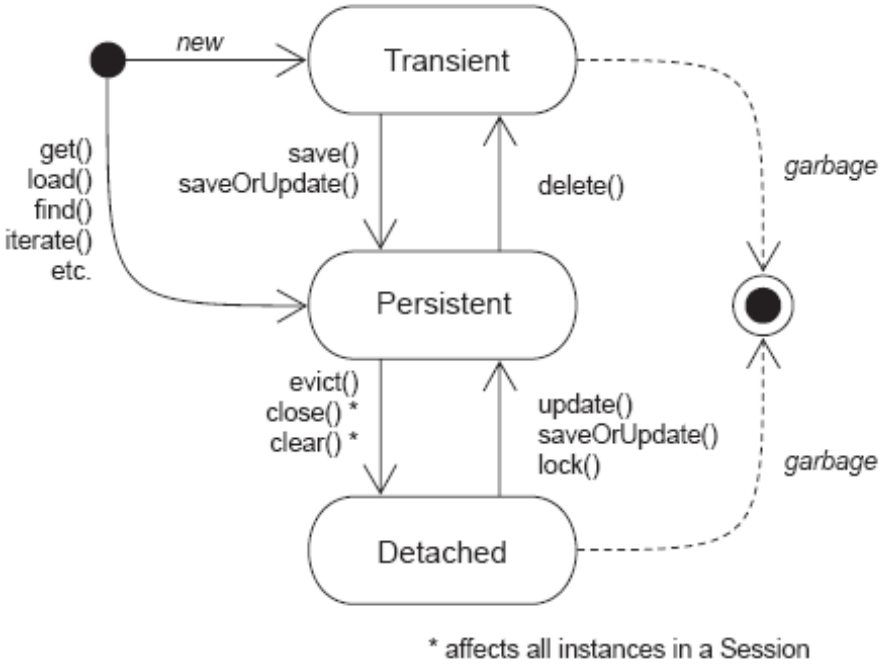
4.3.2 Seja

Seja predstavlja eno logično enoto dela. To je serija zaporednih operacij, ki jih želimo izvesti na podatkovni bazi. V eni enoti dela se lahko zgodi več transakcij. V sistemu bodo vedno nove enote dela, ki jih je potrebno izvesti, zato mora biti ustvarjanje seje poceni računalniški vir. To je razlog, da seja ne uporablja mehanizma za varno uporabo v več nitih, saj je računsko zelo zahteven. V novi niti raje ustvarimo novo sejo s pomočjo razreda *org.Hibernate.SessionFactory*, ki je edinstven in varen za uporabo v večnitnem okolju. Vsaka seja lahko sočasno dostopa do istih podatkov v po-

datkovni bazi, vendar so entitete, ki se iz teh podatkov ustvarijo, v vsaki seji drugačne. Glavna naloga seje je shranjevanje, branje in brisanje preslikanih entitet. Preslikane entitete so lahko v treh različnih stanjih:

- *transient*: entiteta ni shranjena v podatkovni bazi in ni povezana z nobeno sejo
- *persistent*: entiteta je shranjena v podatkovni bazi in je povezana z enolično sejo
- *detached*: entiteta je bila shranjena v podatkovni bazi in ni več povezana z nobeno sejo

Seja ponuja metode za prehajanje entitet med zgoraj navedenimi stanji. *Transient* entitete lahko postanejo *persistent* s klici metod *save(), persist()* ali *saveOrUpdate(). Persistent* entitete lahko postanejo *transient* s klicem metode *delete()*. Vsaka entiteta, ki jo dobimo z *get()* ali *load()* je *persistent*. *Detached* entitete lahko postanejo *persistent* s klicem metod *update(), saveOrUpdate(), lock()* ali replicate(), ampak le pod pogojem, da v seji še ni entitete z enakim enoličnim identifikatorjem. Če v seji ta entiteta že obstaja, sistem Hibernate ne ve, katera entiteta je prava in vrne napako *NonUniqueObjectException*. Če nismo prepričani, ali entiteta v seji že obstaja, lahko *detached* entiteto spremenimo v *persistent* tudi s klicem metode *merge()*, vendar se moramo zavedati, da nam ta metoda vrne novo entiteto. Vsaka *persistent* entiteta je opazovana za spremembe in je ob splaknjevanju, tj. klic metode *flush()*, tudi avtomatično posodobljena.



Slika 4.2: Življenjski cikel entitet

Seja je dodatni nivo med preslikanimi entitetami in podatkovno bazo. Vsaka seja hrani interni predpomnilnik vseh entitet, s katerimi je do sedaj upravljala. Če bo seja odprta predolgo časa in se bo v njej naložilo preveč entitet, se bo aplikacija ustavila z napako *OutOfMemoryException*. To lahko preverimo s preprosto metodo *generirajPodatke()*, ki shranjuje podatke v podatkovno bazo. Entiteta podatek ni povezana z nobeno drugo entiteto. Shranjena entiteta je *persistent*, kar pomeni, da je v internem predpomnilniku seje. Pri približno 50.000 podatkih v internem predpomnilniku seje strežniku zmanjka spomina. Ta številka bi bila še manjša, če bi shranjevali podatke, ki bi bili povezani z drugimi entitetami, saj bi morali v internem predpomnilniku seje hraniti tudi vsako povezano entiteto. Največje število entitet v internem predpomnilniku seje je torej odvisno od velikosti entitete, s katero upravljamo.

V tem preprostem primeru problem rešimo z ročnim odstranjevanjem shranjenih elementov z metodo *session.evict(podatek)*, ali preprosto z metodo *session.clear()*, ki izprazni celotno sejo. Vendar to pomeni, da vsi podatki v seznamu podatkov spremenijo stanje v *detached* in izgubijo povezavo s sejo.

```
List<Podatek> generirajPodatke() {
   Session seja = sessionFactory.openSession();
   Transaction tx = session.beginTransaction();
   List<Podatek> podatki = new ArrayList();
   for ( int i=0; i<100000; i++ ) {
         Podatek podatek = new Podatek(.....);
         podatki.add(podatek);
         seja.save(podatek);
   }
   tx.commit();
   seja.close();
   return podatki;
}
```

Glede na dolžine enote dela poznamo dva načina uporabe seje, in sicer sejo na zahtevo in sejo na pogovor.

Seja na zahtevo

Uporabnik pošlje zahtevo na strežnik, kjer se odpre nova Hibernate seja in začne nova transakcija. Izvede se poslovna logika in branje ali pisanje v podatkovno bazo. Na koncu se seja splakne, transakcija se konča in seja se zapre.
Najbolj pogoste rešitve za implementacijo tega vzorca so ServletFilter, AOP prestreznik nad metodami ali prestrezniški zabojnik. Te rešitve omogočajo prestrezanje začetka in konca zahtevka, kjer ročno poskrbimo za začetek in konec transakcije. Bolj elegantna rešitev je uporaba EJB zabojnikov, kjer deklarativno določimo začetek in konec transakcije.

Seja na pogovor

Veliko poslovnih procesov potrebuje več kot eno interakcijo uporabnika s podatkovno bazo. Preprost primer je dialog, kjer so prikazani podatki iz podatkovne baze. Uporabnik lahko poljubno spreminja prikazane podatke in čez nekaj časa pošlje zahtevo za shranjevanje sprememb. Rešitev je podaljšana seja, ki se ob zaključeni transakciji odklopi od podatkovne baze in se ob naslednjem zahtevku v naslednji transakciji spet priklopi. Seja se ne zapre in njen interni predpomnilnik naloženih entitet se ohranja.

Denimo, da uporabnik poleg zgornjih zahtev želi, da je edini, ki je v določenem času spreminjal podatke. Preprosta rešitev je uporaba daljše transakcije, ki onemogoči dostop do podatkovne baze, dokler uporabnik ne shrani sprememb. To je zelo slab pristop, saj bi bili medtem podatki nedostopni za ostale uporabnike. V praksi večino računalniških aplikacij uporablja več uporabnikov hkrati in to moramo upoštevati že pri načrtovanju sistema. Vsak uporabnik ima dodeljeno svojo enolično sejo, ki lahko ureja iste podatke v podatkovni bazi. Sočasno uporabo podatkov rešujemo z zaklepanjem podatkovne baze.

4.3.3 Zaklepanje in sočasni dostop

Zaklepanje je metoda, ki preprečuje spreminjanje relacijske podatkovne baze med časom, ko podatek preberemo, in časom, ko podatek uporabimo oz. spet zapišemo. Zaklepanje je lahko optimistično ali pesimistično. Sistem Hibernate omogoča obe implementaciji.

Optimistično zaklepanje

Optimistično zaklepanje predvideva, da se lahko več transakcij izvaja sočasno, brez vplivanja ene na drugo. Posamezna transakcija je kratka in ne zaklene dostopa do podatkov. Hibernate implementira optimistično zaklepanje tako, da uporablja dodatno polje, kjer se nahaja verzija ali časovni žig. Ob uspešni transakciji se ustrezno posodobi tudi to polje. Vsaka naslednja

transakcija najprej preveri, če se verzija oz. časovni žig na spremenjenem podatku ujema s tistim v bazi. Ob ujemanju se posodobitev izvede, sicer pa se transakcija razveljavi in Hibernate vrne napako *StaleObjectException — row was updated by another transaction*. S tem zagotovimo, da transakcija ne prepiše zapisa druge transakcije in obenem ohranimo sistem skalabilen.

Pesimistično zaklepanje

Pesimistično zaklepanje predvideva, da bodo sočasne transakcije pogosto vplivale ena na drugo. Posamezna transakcija je dolga in zaklene dostop do podatkov, dokler aplikacija ne preneha uporabljati zahtevanih podatkov. Sistem Hibernate omogoča pesimistično zaklepanje ob vsaki poizvedbi v bazi, kjer kot parameter definiramo tip ključavnice.

4.4 Remote Method Invocation (RMI)

Javanska aplikacija se lahko s poslovno logiko v aplikacijskem strežniku poveže na dva načina, in sicer preko tehnologije JAX-WS ali preko tehnologije RMI. Javanska aplikacija, v kateri želimo implementirati predpomnenje, uporablja tehnologijo RMI.

RMI omogoča javanski aplikaciji, da se poveže z ostalimi aplikacijami. Največkrat se uporablja za komunikacijo med dvema javanskima aplikacijama, a ponuja tudi možnost povezave na Java Native Interface (JNI) in posledično komunikacijo z izvornimi programi. Spletne storitve in podobne rešitve lahko prenašajo le v naprej definirane podatkovne tipe in to zelo omeji njihovo uporabo. RMI lahko brez dodatne kode prenaša celotne javanske objekte. To nam poleg objektno orientiranega programiranja omogoča tudi prenos implementacije objektov in posledično razbremenitev strežnika, saj lahko prejeto implementacijo poženemo sami v željeni komponenti. RMI uporablja tudi vgrajene javanske varnostne mehanizme in po potrebi prepreči komunikacijo z zunanjo komponento.

Tehnologija RMI je za implementacijo predpomnenja pomembna, ker omo-

goča prenašanje celotnega objekta. S tem ohranja identiteto objekta in identiteto vseh povezanih objektov. Prenašanje podatkov preko tehnologije JAX-WS ob vsakem prejemu ustvari nov objekt in zato se identiteta objekta in povezanih objektov ne ohranja.

Poglavje 5

Problem predpomnenja entitet v javanski aplikaciji

V prejšnjih poglavjih smo spoznali okolje javanske aplikacije, v kateri želimo implementirati predpomnenje. Sledi podroben opis, kako je aplikacija povezana z vsako komponento v okolju in zakaj enostavno predpomnenje ne deluje.

Javanska aplikacija dostopa do aplikacijskega strežnika s pomočjo tehnologije RMI. Do strežnika poleg aplikacije dostopa še več spletnih in mobilnih aplikacij. Na strežniku je nameščen EJB zabojnik, v katerem je bazen zrn brez stanj s poslovno logiko. Vsako zrno dobi ob nastanku pripadajoč objekt seje, ki deluje v načinu seja na zahtevo. To zagotavlja dobro zmogljivost pri veliko sočasnih zahtevah na strežnik.

EJB zabojnik vsaki zahtevi dodeli prosto zrno iz bazena. V zrnu se začne prazna seja in nova transakcija. Izvede se poslovna logika, transakcija se konča in seja se zapre. Zrno se doda nazaj v bazen. Morebitne vrnjene entitete niso več povezane s sejo in so v stanju *detached*. Morebitne vrnjene entitete bodo zato vedno novi objekti. To pomeni, da bo več zaporednih zahtev za enak enolični identifikator vrnilo več različnih objektov. Podatki na vsakem objektu bodo enaki, vendar ti objekti ne bodo zasedali istega prostora v spominu. Problem nastane pri predpomnenju vrnjenih objektov, ki

so povezavani z drugimi objekti. Vsaka sprememba objekta v predpomnilniku mora spremeniti tudi vse ostale objekte, ki predstavljajo enak podatek v podatkovni bazi, sicer bo predpomnilnik postal nekonsistenten.

5.1 Praktični primer

Problem predpomnenja entitet bomo bolj podrobno spoznali skozi naslednji primer. Definirali bomo dve preprosti entiteti, tj. entiteto *Podjetje* in entiteto *Uporabnik*, ki sta povezani s kardinalnostjo ena proti mnogo.

```
public class Uporabnik {
   private Long id;
   private Podjetje podjetje;

   public void Uporabnik(Podjetje podjetje) {
      this.podjetje = podjetje;
   }
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id=id;
   }
   public Podjetje getPodjetje() {
      return podjetje;
   }
   public void setPodjetje(Podjetje podjetje) {
      this.podjetje = podjetje;
   }
}

public class Podjetje {
   private Long id;
   private String imePodjetja;
   private List<Uporabnik> uporabniki;

   public void Podjetje(String imePodjetja, List<Uporabnik> uporabniki) {
      this.imePodjejta = imePodjetja;
      this.uporabniki = uporabniki;
   }
```

```
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id=id;
    }
    public String getImePodjetja() {
        return imePodjetja;
    }
    public void setImePodjetja(String imePodjetja) {
        this.imePodjetja = imePodjetja;
    }
    public List<Uporabnik> getUporabniki() {
        return uporabniki;
    }
    public void setUporabniki(List<Uporabnik> uporabniki) {
        this.uporabniki = uporabniki;
    }

}
```

V podatkovni bazi imamo dva uporabnika z enoličnima identifikatorjema 1 in 2, ki sta povezana s podjetjem z enoličnim identifikatorjem 1.

V javanski aplikaciji preko RMI upravljalca izvedemo dve zaporedni zahtevi na strežnik. Za vsako zahtevo se na strežniku iz bazena pridobi sveže zrno, kjer se začne prazna seja in nova transakcija. Ko se zahteva zaključi, se transakcija zapre in seja izprazni. Prejete entitete dodamo v predpomnilnik, kot je definirano v poglavju 3. Ključi v predpomnilniku imajo format *imeRazreda* + *idRazreda*, ker imajo različni tipi entitet enake enolične identifikatorje.

V tem trenutku spremenljivka *uporabnik1* in *predpomnilnik.get("Uporabnik1")* kaže na isti objekt v računalniškem spominu. Enako velja za spremeljivko *uporabnik2* in *predpomnilnik.get("Uporabnik2")*. Javanski virtualni stroj namreč upravlja z referencami na obstoječe objekte. Nov objekt se v spominu ustvari le ob klicanju konstruktorja objekta. Drugače velja za podjetje na obeh uporabnikih. Čeprav obe podjetji nosita in predstavljata iste podatke v podatkovni bazi, to nista ista objekta. Razlog za to sta dve različni zahtevi na strežnik. V vsaki zahtevi je sveža seja, ki iz podatkovne baze prebere

podjetje, ustvari nov objekt in ga doda na zahtevanega uporabnika.

```
@Stateless
public class UpravljalecUporabnikovEJB {
   @PersistenceContext(unitName = "GlavnaAplikacijskaEnota")
   private Session session;

   public Uporabnik pridobiUporabnika(long id) {
      return (Uporabnik) session.get(Uporabnik.class, id);
   }
   public Uporabnik posodobiUporabnika(Uporabnik uporabnik) {
      return (Uporabnik) session.merge(uporabnik);
   }
 }

Uporabnik uporabnik1 = RMIUpravljalec.pridobiUpravljalcaUporabnikov().
    pridobiUporabnika(1L);
predpomnilnik.put("User" + uporabnik1.getId(), uporabnik1);
assertEquals("Isti objekt!", uporabnik1, cache.get("Uporabnik1"));

Uporabnik uporabnik2 = RMIUpravljalec.pridobiUpravljalcaUporabnikov().
    pridobiUporabnika(2L);
predpomnilnik("Uporabnik" + uporabnik2.getId(), uporabnik2);
assertEquals("Isti objekt!", uporabnik2, predpomnilnik.get("Uporabnik2"));

assertEquals("Razlicna objekta!", uporabnik1.getPodjetje(), uporabnik2.
    getPodjetje());
```

Predpomnilnik je zaenkrat konsistenten. V nadaljevanju preberemo prvega uporabnika iz predpomnilnika in njegovemu podjetju spremenimo ime. Sedaj v predpomnilniku ime podjetja pri prvem uporabniku ni več enako imenu podjetja pri drugem uporabniku in predpomnilnik je nekonsistenten.

```
Uporabnik uporabnik = predpomnilnik.get("Uporabnik1");
uporabnik.getPodjetje().setImePodjetja("NovoImePodjetja");
assertEquals("Imeni podjetij nista enaki!", predpomnilnik.get("Uporabnik1"
    ).getPodjetje().getImePodjetja(), predpomnilnik.get("Uporabnik2").
    getPodjetje().getImePodjetja());
```

Problem predpomnenja povezanih entitet lahko rešujemo na več načinov.

Prva možnost je, da spremenimo način delovanja strežnika tako, da nam bo vedno vračal iste objekte. To dosežemo z uporabo sejnih zrn, ki ohranjajo

stanje in uporabljajo podaljšano sejo. V tem primeru bi aplikacija ob vsaki zahtevi dostopala do istega zrna in iste seje z istimi objekti. Več zaporednih zahtev določene entitete bi vedno vrnilo iste objekte.
Pri tem je pomembno vedeti, da bi ta pristop deloval le, če do strežnika dostopamo preko tehnologije RMI. Dostop do strežnika preko spletnih storitev bi v vsakem primeru vedno ustvaril nove objekte in tudi sejno zrno s podaljšano sejo ne bi rešilo problema.

Druga možnost je uporaba porazdeljenega predpomnilnika na strežniku in v aplikaciji. Predpomnilnika bi med seboj komunicirala in invalidirala oz. replicirala entitete glede na spremembe.
V našem primeru nastane problem, ker je v poslovnem modelu več sto entitet z več tisoč podatki, ki bi jih morali pri obeh rešitvah ohranjati na strežniku, v podaljšani seji v vsakem sejnem zrnu ali v porazdeljenem predpomnilniku. To zelo upočasni strežnik, zato bomo poskusili najti rešitev, ki bi sprostila strežnik in delo prepustila javanski aplikaciji.

Tretja možnost je, da ob vsaki spremembi posodobimo vse objekte v predpomnilniku, ki predstavljajo isti podatek v podatkovni bazi. To bi bilo časovno in prostorsko zelo zahtevno (v smislu pomnilnika), saj bi morali hraniti seznam vseh entitet, ki predstavljajo enak podatek.

Četrta možnost je hitrejša, a ima tudi višjo ceno. Ob vsaki spremembi bi lahko pobrisali vse entitete v predpomnilniku. S tem bi zagotovili, da so podatki v predpomnilniku vedno konsistenti, saj bi jih morali znova naložiti iz podatkovne baze. V sistemih, kjer so spremembe redke, bi bila ta rešitev zadovoljiva.

Peta in najbolj učinkovita možnost je upravljanje z objekti in njegovimi povezanimi objekti tako, da je v predpomnilniku vedno le en objekt, ki predstavlja določen podatek v podatkovni bazi. To zagotavlja, da bo sprememba tega objekta vidna iz vseh njegovih povezanih objektov. V nadaljevanju bomo preverili, ali katera izmed naprednih implementacij predpomnenja v Javi že

ponuja to funkcionalnost.

5.2 Primerjava obstoječih javanskih predpomnilnikov

Tabela 5.1: Primerjava obstoječih Javanskih predpomnilnikov

	Guava cache	cache2k	ehcache	POJO Cache	Infinispan	ShiftOne	WhirlyCache	SwarmCache	Cache4j	Java Caching System	Java Object cache
Odstranjevanje glede na velikost	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Odstranjevanje glede na čas	✓	✓	x	x	✓	✓	x	✓	✓	✓	✓
Shranjevanje na sekundarni medij	x	✓	✓	✓	✓	x	x	x	x	✓	✓
Omogoča sprotno zapisovanje	✓	✓	✓	✓	✓	x	x	x	x	x	✓
Omogoča zakasnjeno zapisovanje	x	✓	✓	✓	✓	x	x	x	x	x	✓
Omogoča distribuiranost	x	x	✓	x	✓	✓	x	✓	✓	✓	✓
Omogoča transakcije	x	x	x	✓	x	x	x	x	x	x	x
Avtomatsko posodabljanje spremenjenih polj objekta	x	x	x	✓	x	x	x	x	x	✓	x
Statistika	✓	✓	✓	✓	✓	✓	x	x	x	✓	✓
Večnitna podpora	✓	x	✓	x	✓	✓	✓	x	✓	✓	✓
Konsistentnost podatkov	x	x	x	x	x	x	x	x	x	x	x

Zgoraj navedene javanske implementacije predpomnilnikov so uporabne za samostojne neodvisne aplikacije. Predpomnenje je posebej učinkovito, če entitete večinoma beremo in redko spreminjamo. Noben od navedenih predpomnilnikov ne ponuja rešitve za problem, kjer aplikacijski strežnik vrača različne objekte.

Poglavje 6

Implementacija predpomnilnika po meri

Kot sledi iz prejšnjega poglavja, bomo morali v javanski aplikaciji ob nalaganju posameznega objekta ročno upravljati njegove povezane objekte. Razvili bomo pametni sistem, ki bo za vsak podatek vedno hranil le en objekt. Vse spremembe objekta bodo tako vidne iz vseh ostalih povezanih objektov.

Najprej bomo osnovni predpomnilnik nadgradili, da bo razlikoval med različnimi razredi objektov. Prvi ključ v zunanji mapi predstavlja ime razreda. Različni tipi objektov imajo lahko enak enolični identifikator, zato predpomnilnik s ključem za enolični identifikator ni dovolj. Gnezdena mapa predstavlja vse objekte istega razreda. Ključ je enolični identifikator, vrednost pa par, v katerem sta objekt in logični izraz. Slednji pove, ali je objekt v celoti naložen. Kot smo spoznali v poglavju 4.2.1, se lahko zgodi, da nekateri objekti še niso v celoti naloženi. Logični izraz nam pove, ali je dostopan objekt že v celoti naložen. Če še ni in če ga potrebujemo, ga moramo najprej naložiti iz podatkovne baze.

```
Map<String, Map<Long, Pair<Object, Boolean>>> predpomnilnik = new HashMap
    <>();
```

Osnovna ideja algoritma je, da vse vrednosti na prejetem objektu iz strežnika zapišemo v objekt shranjen v predpomnilniku. To nam nudi tehnologija Java Reflection API, ki v realnem času omogoča dostop in klic metod določenega razreda.

6.1 Algoritem

Metoda kot vhodni parameter sprejme objekt, ki ga je potrebno posodobiti oz. dodati v predpomnilnik in trenutno globino posodabljanja. Slednjo potrebujemo, ker se metoda kliče rekurzivno z vsakim povezanim objektom vhodnega objekta. V predpomnilnik je potrebno dodati tudi vse povezane objekte. Naivni pristop je posodobitev vseh povezanih objektov in posledično celotne strukture. Izkaže se, da to ni potrebno, saj moramo zagotoviti pravilno povezanost objektov le do nivoja, do katerega se lahko objekti spreminjajo. V našem primeru uporabnikov in podjetij je največja globina ena, saj iz uporabnika, ki je na nivoju nič, spremenimo podjetje, ki je na nivoju ena. Posodabljanje morebitnih drugih povezanih objektov podjetja, ki bi imeli globino dve, bi bilo v tem primeru nesmiselno, saj se niso spremenili. S statično spremenljivko definiramo največjo globino, do katere mora algoritem posodabljati objekte. Največjo globino definiramo glede na to, kako globoko v strukturi se lahko objekti spremenijo. Globina nič vedno pomeni, da je objekt v celoti naložen.

Sledi v javanski pseudokodi zapisan algoritem za prepisovanje oz. dodajanje novih objektov v predpomnilnik. Razlaga delovanja algoritma je napisana na koncu.

```
public class Par<A, B> {
  private A prvi = null;
  private B drugi = null;

  public Par(A prvi, B drugi) {
      this.prvi = prvi;
      this.drugi = drugi;
```

```
    }

    public A getPrvi() {
        return prvi;
    }

    public B getDrugi() {
        return drugi;
    }

    public void setPrvi(A novPrvi) {
        prvi = novPrvi;
    }

    public void setDrugi(B novDrugi) {
        drugi = novDrugi;
    }
}
static Integer NAJVECJA_GLOBINA = 2;
Object kopirajAliDodajVPredpomnilnik(Object novObjekt, Integer globina
    ) {
    Map<Long, Pair<Object, Boolean>> mapaRazreda = predpomnilnik.get(
        novObjekt.getClass().getName());
    if (mapaRazreda == null) {
        mapaRazreda = Maps.newHashMap();
        predpomnilnik.put(novObjekt.getClass().getName(), mapaRazreda);
    }
    Pair<Object, Boolean> predpomnjenObjekt = mapaRazreda.get(novObjekt
        ).getId());
    boolean zePredpomnjen = predpomnjenObjekt != null;
    boolean zePredpomnjenBoPolnoNalozen = false;

    if (zePredpomnjen) {
        if (predpomnjenObjekt.getFirst().equals(novObjekt)) {
            return predpomnjenObjekt;
        }
        if(predpomnjenObjekt.getFirst().getVersion().equals(novObjekt).
            getVersion())) {
            return predpomnjenObjekt;
        }
        zePredpomnjenBoPolnoNalozen = !predpomnjenObjekt.getSecond() &&
             globina == 0;
        if (zePredpomnjenBoPolnoNalozen) {
            predpomnjenObjekt.setSecond(true);
        }
```

```
}

if (globina >= MAX_DEPTH) {
    return zePredpomnjen ? predpomnjenObjekt.getFirst() : novObjekt
        ;
}

for (Method metodaZaDostopanje : novObjekt.getClass().getMethods())
     {
    if ((metodaZaDostopanje.getName().startsWith("get") ||
        metodaZaDostopanje.getName().startsWith("is"))) {
        boolean metodaJePovezava = false;
        boolean metodaJePrehodna = false;

        for (Annotation anotacija : metodaZaDostopanje.
            getAnnotations()) {
            if (anotacija.annotationType().equals(OneToOne.class) ||
                    anotacija.annotationType().equals(OneToMany.class
                        ) ||
                    anotacija.annotationType().equals(ManyToOne.class
                        ) || anotacija.annotationType().equals(
                        ManyToMany.class)) {
                metodaJePovezava = true;
            }
            if (anotacija.annotationType().equals(Transient.class))
                {
                metodaJePrehodna = true;
                break;
            }
        }
        if(metodaJePrehodna) {
            continue;
        }
        Method metodaZaNastavljanje = null;
        try {
            String imeMetode = "set" + metodaZaDostopanje.getName().
                substring(metodaZaDostopanje.getName().startsWith("
                is") ? 2 : 3);
            Class<?> returnType = metodaZaDostopanje.getReturnType()
                ;
            if(boolean.class.isAssignableFrom(returnType)) {
                returnType = Boolean.class;
            }
            metodaZaNastavljanje = metodaZaDostopanje.
                getDeclaringClass().getMethod(imeMetode, returnType)
```

```
        ;
} catch (NoSuchMethodException e) {
    System.out.println(e);
}

try {
    if (metodaJePovezava) {
        if (Collection.class.isAssignableFrom(
            metodaZaDostopanje.getReturnType())) {
            Collection zbirka = (Collection)
                metodaZaDostopanje.invoke(novObjekt);
            if (zbirka != null && Hibernate.isInitialized(
                zbirka)) {
                Set novaZbirka = Sets.newHashSet();
                for (Object povezanObjekt : zbirka) {
                    Object predpomnjenPovezanObjekt =
                        kopirajAliDodajVPredpomnilnik(
                        povezanObjekt, globina + 1);
                    novaZbirka.add(predpomnjenPovezanObjekt);
                }
                if (!zePredpomnjen) {
                    zbirka.clear();
                    zbirka.addAll(novaZbirka);
                }
                if (zePredpomnjenBoPolnoNalozen) {
                    metodaZaNastavljanje.invoke(
                        predpomnjenObjekt.getFirst(),
                        novaZbirka);
                }
            }
        } else {
            Object povezanObjekt = metodaZaDostopanje.invoke(
                novObjekt);
            if (povezanObjekt != null && Hibernate.
                isInitialized(povezanObjekt)) {
                Object predpomnjenPovezanObjekt =
                    kopirajAliDodajVPredpomnilnik(
                    povezanObjekt, globina + 1);
                if (!zePredpomnjen) {
                    metodaZaNastavljanje.invoke(novObjekt,
                        predpomnjenPovezanObjekt);
                }
                if (zePredpomnjenBoPolnoNalozen) {
                    metodaZaNastavljanje.invoke(
                        predpomnjenObjekt.getFirst(),
```

```
                                predpomnjenPovezanObjekt);
                        }
                    }
                }
            } else if (zePredpomnjen) {
                Object preprostoPolje = metodaZaDostopanje.invoke(
                    novObjekt);
                metodaZaNastavljanje.invoke(predpomnjenObjekt.
                    getFirst(), preprostoPolje);
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
if (!zePredpomnjen) {
    mapaRazreda.put(((RootEntity) novObjekt).getId(), new Pair(
        novObjekt, globina == 0));
}
return zePredpomnjen ? predpomnjenObjekt.getFirst() : novObjekt;
}
```

6.2 Opis algoritma

Metoda *kopirajAliDodajVPredpomnilnik* najprej preveri, ali v predpomnilniku že obstaja mapa za razred novega objekta. Če mape še ni, jo ustvari. Če mapa za ta razred že obstaja, metoda preveri, ali v mapi obstaja vrednost za enolični identifikator novega objekta. Če vrednost obstaja in če je ta vrednost enaka novemu objektu, ali ima enako verzijo, potem je bil objekt že posodobljen ali se sploh ni spremenil. V tem primeru enostavno vrnemo predpomnjen objekt. V nasprotnem primeru se metoda z uporabo tehnologije Java Reflection API sprehodi čez vse metode za dostopanje do polj novega objekta. Dostopamo lahko do anotacij metod in ugotovimo, ali gre za povezano entiteto. Kot smo se naučili v poglavju 4.2.1, je povezana entiteta lahko naložena ali ne, odvisno od tipa povezave z drugim objektom in nastavitev sistema Hibernate. Če je metoda za dostopanje do polj objekta anotirana kot povezana entiteta in če je ta entiteta naložena, potem se me-

toda *kopirajAliDodajVPredpomnilnik* kliče rekurzivno s povezano entiteto in globino povečano za ena. Trenutno metodo za dostopanje do polja objekta spremenimo v metodo za nastavljanje polja objekta. Vrnjeno entiteto zapišemo v novi objekt tako, da kličemo zgrajeno metodo za nastavljanje polj objekta. To storimo, ker je vrnjena entiteta vedno tista, ki se nahaja v predpomnilniku. Lahko je bila ravno dodana v predpomnilnik ali pa je že obstajala in se je le prepisala s svežimi vrednostmi. S tem zagotavljamo, da so vsi novi objekti povezani s tistimi, ki so že v predpomnilniku, in da so objekti v predpomnilniku vedno ustrezno posodobljeni. Če metoda za nastavljanje polj objekta ni anotirana kot povezava na drug objekt in če je nov objekt že v predpomnilniku, potem zopet uporabimo zgrajeno metodo za nastavljanje polj objekta in z njo nastavimo novo svežo vrednost na predpomnjen objekt. To pomeni, da prepišemo vsa navadna polja iz novega objekta v predpomnjen objekt. Če objekta še ni v predpomnilniku, potem navadnih polj ni potrebno prepisovati, ampak le dodamo nov objekt v predpomnilnik. Na koncu vrnemo predpomnjen objekt.

Algoritem uporabimo na nivoju med nalaganjem entitet iz podatkovne baze in predpomnilnikom. V praksi to pomeni, da kličemo metodo *kopirajAliDodajVPredpomnilnik* z vsako prejeto entiteto iz podatkovne baze. Pomembno je, da metodo kličemo tudi z vsako prejeto entiteto po shranjevanju z *session.merge(spremenjenaEntiteta)*. Kot smo spoznali v poglavju 4.3, ob shranjevanju *detached* entitet dobimo nazaj novo entiteto s posodobljeno verzijo. Tudi to entiteto je potrebno zapisati v predpomnilnik, da le-ta ostaja konsistenten.

6.3 Integracija v obstoječo javansko aplikacijo

Rešitev sem integriral v obstoječo javansko aplikacijo in izmeril, koliko dodatnega časa je potrebnega pri predpomnenju entitet iz strežnika. Dodatni čas je odvisen od števila povezanih entitet na entiteti, ki jo dobimo iz strežnika,

in od največje globine, ki jo moramo prepisati. Najbolj pogoste so spremembe do drugega nivoja, zato sem to vrednost uporabil tudi pri testiranju. V javanski aplikaciji sem za vsako naloženo entiteto izmeril, koliko časa je algoritem potreboval za predpomnenje. Ker je dodatni čas predpomnenja odvisen od števila povezanih enitet na entiteti, ki jo dobimo iz strežnika, sem za merilo vzel čas, potreben za pridobitev entitete iz strežnika. Čas pridobivanja entitete iz strežnika se na enak način sorazmerno povečuje s številom povezanih entitet, zato je to ustrezno merilo. Dodatni čas predpomnenja entitet sem v odstotkih izrazil po naslednji enačbi:

$$dodatniCasPredpomnenja = \frac{casIzvajanjaAlgoritma}{casPridobivanjaEntiteteIzStreznika}$$

V javanski aplikaciji sem naložil vse entitete, za vsako izračunal dodatni čas v odstotkih in jih na koncu povprečil. Med vsakim nalaganjem sem počistil predpomnilnik, zato da se je algoritem vedno sprehodil čez vse povezane objekte. Dodatni čas predpomnenja je znašal 1.2% časa pridobivanja entitete iz strežnika.

Poglavje 7

Sklepne ugotovitve

Algoritem je smiselno uporabiti, ko v javanski aplikaciji predpomnimo in pogosto spreminjamo veliko število povezanih entitet. Uporaba algoritma v javanski aplikaciji razbremeni strežnik, saj zagotavlja konsistentnost podatkov, ne da bi jih bilo potrebno hraniti v internem predpomnilniku seje. Cena uporabe algoritma je 1.2% daljši čas predpomnenja entitete. Možna izboljšava predlaganega algoritma je uporaba asinhronosti. Algoritem lahko po potrebi integriramo z obstoječimi rešitvami za predpomnenje in s tem pridobimo dodatne funkcionalnosti predpomnenja. Rešitev je možno nadgraditi kot samostojno knjižnico za predpomnenje, ki združi predlagani algoritem in funkcionalnosti obstoječih predpomnilnikov.

Literatura

[1] J. Juneau. *Java EE 7 Recipes*. Apress, 2013.

[2] J. Wetherbee, C. Rathod, R. Kodali, and P. Zadrozny. *Beginning EJB 3*. Apress, 2013.

[3] C. Richardson. *POJOs In Action*. Manning, 2006.

[4] C. Bauer, G. King. *Java Persistence With Hibernate*. Manning, 2007.

[5] Java Remote Method Invocation [Online]. Dosegljivo: `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html`. [Dostopano 19. 8. 2015].

[6] Cache [Online]. Dosegljivo: `https://en.wikipedia.org/wiki/Cache_(computing)`. [Dostopano 5. 8. 2015].

[7] Cache algorithms [Online]. Dosegljivo: `https://en.wikipedia.org/wiki/Cache_algorithms`. [Dostopano 8. 8. 2015].

[8] Guava Cache [Online]. Dosegljivo: `https://github.com/google/guava/wiki/CachesExplained`. [Dostopano 10. 8. 2015].

[9] cache2k [Online]. Dosegljivo: `http://cache2k.org/`. [Dostopano 10. 8. 2015].

[10] Ehcache [Online]. Dosegljivo: http://ehcache.org/generated/2.10.0/html/ehc-all/. [Dostopano 10. 8. 2015].

[11] POJO Cache [Online]. Dosegljivo: http://docs.jboss.org/jbosscache/2.0.0.GA/PojoCache/en/html_single/. [Dostopano 10. 8. 2015].

[12] Infinispan [Online]. Dosegljivo: http://infinispan.org/docs/8.0.x/user_guide/user_guide.html. [Dostopano 10. 8. 2015].

[13] ShiftOne [Online]. Dosegljivo: http://jocache.sourceforge.net/. [Dostopano 10. 8. 2015].

[14] whirlycache [Online]. Dosegljivo: https://code.google.com/archive/p/whirlycache/. [Dostopano 10. 8. 2015].

[15] SwarmCache [Online]. Dosegljivo: http://swarmcache.sourceforge.net/. [Dostopano 10. 8. 2015].

[16] cache4j [Online]. Dosegljivo: http://cache4j.sourceforge.net/. [Dostopano 10. 8. 2015].

[17] Java Caching System [Online]. Dosegljivo: https://commons.apache.org/proper/commons-jcs/. [Dostopano 10. 8. 2015].

[18] Java Object Cache [Online]. Dosegljivo: http://docs.oracle.com/cd/B14099_19/web.1012/b14012/objcache.htm. [Dostopano 10. 8. 2015].

[19] GNU General Public Licence. [Online]. Dosegljivo: https://www.gnu.org/copyleft/gpl.html. [Dostopano 20. 9. 2014].