Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Darko Božidar

Analiza kontekstov, odvisnosti in anotacij v Java EE

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00028/2012
Datum: 11.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **DARKO BOŽIDAR**

Naslov: **ANALIZA KONTEKSTOV,„ODVISNOSTI IN ANOTACIJ V JAVA EE**
CONTEXT, DEPENDENCY AND ANNOTATION ANALYSIS IN JAVA EE

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Proučite koncepte injekcije kontekstov in odvisnosti na platformi Java EE. Analizirajte šibko sklopljenost in močno tipiziranje ter jih preslikajte na načrtovalske vzorce. Analizirajte anotacije v Java EE platformi. Razvijte praktično spletno aplikacijo, v kateri boste prikazali delovanje zgoraj opisanih konceptov.

Mentor:

prof. dr. Matjaž B. Jurič

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Darko Božidar, z vpisno številko **63090023**, sem avtor diplomskega dela z naslovom:

Analiza kontekstov, odvisnosti in anotacij v Java EE

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča,
- 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 v zbirki "Dela FRI".

V Ljubljani, dne 7. septembra 2012 Podpis avtorja:

Za mentorstvo, pomoč in strokovne nasvete se zahvaljujem profesorju dr. Matjažu Branku Juriču.

Iskreno se zahvaljujem tudi svojim staršem in sestri Vesni za podporo in potrpežljivost skozi vsa leta študija.

Kazalo

Povzetek

Cilj diplomske naloge je preučiti tehnologiji Java EE, in sicer CDI in anotacije ter s pridobljenim znanjem razviti preprosto spletno aplikacijo, ki bi vsebovala nekaj lastno razvitih anotacij in bi temeljila na CDI. V ta namen je bilo potrebno razjasniti, kaj sploh CDI je oziroma kaj sploh omogoča. Zato sta bili najprej podrobneje preučeni omenjeni tehnologiji, kjer smo poskušali ugotoviti, kako izboljšujeta platformo Java EE. V nalogi je namen predvsem raziskati, kako preučevani tehnologiji posledično pripomoreta k hitrejšemu in bolj učinkovitemu razvoju. Po opravljenem teoretičnem delu diplomske naloge, je bila razvita preprosta spletna aplikacija namenjena zavarovalnicam. Ta aplikacija omogoča dodajanje in urejanje zaposlenih ter zavarovancev, sklepanje zavarovanj, beleženje dodanih zaposlenih in zavarovancev, opravljanje in beleženje transakcij ter urejanje cenika. Pri implementaciji le-te so bile uporabljene tehnologije CDI kot so močno tipizirano vstavljanje odvisnosti, kvalifikatorji (tudi vgrajeni), imena EL, stereotipi (samo vgrajeni), proizvajalne metode, proizvajalni atributi, odstranjevalne metode, programsko poizvedovanje, dosegi in konteksti, tipi prestrezniških vezav ter dogodki.

Ključne besede

CDI, anotacije, Java EE, vstavljanje odvisnosti, šibka sklopljenost, močna tipiziranost

Abstract

The goal of this bachelor's thesis is to analyze two of Java EE's features, CDI and annotations, and to use the acquired knowledge to build a simple web application based on CDI and developed annotations. For this purpose it was necessary to clarify what CDI does and what it offers. Previously mentioned features were therefore firstly thoroughly examined to find out what improvements to the Java EE platform, if any, they provide. The main purpose of this thesis is to explore and analyse how these features subsequently contribute to a faster and more efficient development process. After the theoretical part of the thesis a simple web application intended for insurance agents was developed. This application enables a user to add and edit information about the policyholders and employees, to record information about new employees and policyholders, start and log new transactions, edit price lists and finalizing new contracts. In the implementation of these user features various CDI features like typesafe dependency injection, qualifiers (built-in ones too), EL names, producer methods, producer fields, stereotypes (only built-in ones), disposer methods, programmatic lookup, scopes and contexts, interceptor binding types and events.

Keywords

CDI, annotations, Java EE, dependency injection, loose coupling, typesafe

Poglavje 1

Uvod

Dandanes vse bolj narašča število poslovnih oziroma spletnih aplikacij. Zaradi vse večje konkurence razvoja le-teh je potrebno, da njihov razvoj poteka kar se da hitro in učinkovito. Zelo pomembno je tudi, da so aplikacije čim bolj dinamične in dobro strukturirane, kar posledično privede do tega, da so tudi bolj prilagodljive na morebitne spremembe. Zaradi tega je poznavanje tehnologij, ki nam omogočajo razvoj po omenjenih kriterijih, ključnega pomena.

Ena izmed platform, ki omogoča takšen razvoj omenjenih aplikacij, je Java EE. Začetki razvoja Jave EE segajo v leto 1998, ko je podjetje Sun oznanilo projekt z nazivom Profesionalna izdaja Java EE (Java Professional Edition). Od takrat njen razvoj poteka zelo hitro. Trenutna aktualna verzija je Java EE 6, pri čemer je naslednja verzija 7 že napovedana za tekoče leto (2012) [8]. Danes ponuja že veliko število tehnologij, ki omogočajo učinkovit razvoj. Naštejemo jih lahko kar nekaj, na primer servleti, JSF, JSP, EJB, JMS, JPA, CDI, anotacije in podobne [2, 8].

Motivacijo diplomskega dela je predstavljalo poznavanje kar se da učinkovitih razvojnih tehnologij, s tem posledično učinkoviteje razvijati spletne in poslovne aplikacije in biti konkurenčnejši v vse hitreje rastočem svetu omenjenih aplikacij. V ta namen v diplomskem delu preučujemo dve tehnologiji Java EE, in sicer CDI in anotacije. V grobem omenjeni tehnologiji objektom

omogočata vezavo na kontekste življenjskega cikla, močno tipizirano vstavljanje, vezavo na prestreznike in dekoratorje, interakcijo s pomočjo dogodkov in integracijo z zunanjimi ogrodji.

V prvem delu diplomske naloge smo najprej opredelili omenjeni tehnologiji. To smo storili tako, da smo se v drugem poglavju v grobem osredotočili na glavne koncepte in funkcionalnosti CDI. V tretjem poglavju smo preučili zrna in opredelili, kaj so zrna, kakšne vrste zrn poznamo, vstavljanje leteh ter dosege povezane z njimi. Bolj kompleksne funkcionalnosti CDI, kot so: proizvajalne metode, prestrezniki, dekoratorji, dogodki, stereotipi, alternative, specializacija in viri komponentnega okolja Java EE, opisuje četrto poglavje. Peto poglavje opisuje namestitev in pakiranje aplikacij CDI ter v grobem opiše prenosne razširitve [3, 4]. Nato smo s pridobljenim znanjem razvili spletno aplikacijo, ki je vsebovala nekaj anotacij in je temeljila na specifikaciji CDI. Pri te smo morali paziti, da je bila uporaba omenjenih tehnologij smiselna. Praktični preizkus nekaterih konceptov CDI na aplikaciji opisuje šesto poglavje. Sklepno sedmo poglavje povzema bistvene ugotovitve diplomskega dela.

Poglavje 2

Koncepti CDI

Specifikacija Konteksti in vstavljanje odvisnosti oziroma angleško Contexts and Dependency Injection (CDI) predstavlja nabor dopolnilnih storitev za izboljšanje strukture kode aplikacije. CDI storitve ponujajo [3, 4]:

- izboljšan življenjski cikel objektov s stanjem, ki so vezani na dobro definirane kontekste,
- močno tipiziran pristop pri vstavljanju odvisnosti, vključno z možnostjo izbire različne odvisnosti v času razvoja ali času namestitve,
- interakcijo med objekti preko dogodkovno obveščevalne strukture,
- boljši pristop pri vezavi prestreznikov na objekte, poleg tega pa tudi novo vrsto prestreznikov, imenovano dekorator,
- vmesnik ponudnika storitev oziroma angleško Service Provider Interface (SPI) za razvoj prenosnih razširitev vsebnika.

Specifikacija platforme Java EE definira funkcionalnost vstavljanja virov komponentnega okolja Java EE s pomočjo imen, osnovanih na nizih. CDI izboljšuje to funkcionalnost, saj omogoča vstavljanje široke množice objektov (poleg virov) na osnovi močnega tipiziranja. Če vstavljamo zrna, ki implementirajo enak vmesnik, nam ni več potrebno navesti celotnega imena razreda, ampak lahko namesto tega uporabimo kvalifikatorje. S tem preprečimo močne odvisnosti med odjemalcem in implementacijo. Specifikacija

CDI izboljšuje tudi specifikacijo EJB in specifikacijo upravljanih zrn, saj ponuja kontekstno upravljanje življenjskega cikla. Vsaka instanca sejnega ali upravljanega zrna, pridobljena s pomočjo vstavljanja odvisnosti, je kontekstna instanca. Vezana je na kontekst življenjskega cikla in je na voljo objektom, ki se izvajajo v istem kontekstu. V trenutku, ko odjemalec potrebuje instanco, jo vsebnik samodejno ustvari in jo prav tako samodejno uniči, ko je kontekst končan. Vsebnik opravlja vstavljanje odvisnosti v sejna in sporočilna zrna, tudi če niso kontekstne instance. Zrna lahko poleg tega sprožajo in opazujejo dogodke. Ta funkcionalnost dopušča popolnoma ločeno interakcijo med zrni, brez odvisnosti v času prevajanja. Opazovalce in povzročitelje dogodkov lahko s pomočjo kvalifikatorjev povežemo na močno tipiziran način. Opazovalci so lahko obveščeni takoj ali določijo zakasnitev dostave dogodka do konca trenutne transakcije. Izboljšana je vezava prestreznikov. Anotacija @Interceptors (uporabljena v upravljanih zrnih in specifikaciji EJB), ima številne slabosti, saj je implementacija prestreznika zakodirana v poslovni kodi. Poleg tega ni enostavno onemogočiti prestreznikov v času namestitve, vrstni red prestreznikov pa ni globalen. CDI ponuja nov, močno tipiziran način vezave prestreznikov, s pomočjo tipov prestreziških vezav, brez neposrednih odvisnosti med prestreznikom in prestreženim razredom. Hkrati zagotavlja enostavno omogočanje prestreznikov in definiranje njihovega globalnega vrstnega reda. Velika pridobitev specifikacije CDI je dekorator, ki omogoča dinamično dodajanje vedenja nekemu objektu v času izvajanja aplikacije. Dekorator je zrno, ki implementira metode prestreženega tipa. S tem posledično pozna celotno semantiko, vezano na ta vmesnik in predstavlja odlično orodje za reševanje poslovnih problemov. Predstavlja veliko izboljšavo podrazredov, kajti podrazred doda vedenje v času prevajanja, kar pomeni, da sprememba vpliva na vse instance prvotnega razreda, dekorator pa omogoča dodajanje vedenja posameznim objektom v času izvajanja. Hkrati pa jih lahko enostavno omogočimo in jim določimo globalni vrstni red. Specifikacija CDI prav tako omogoča, da lahko vsakemu zrnu določimo ime EL, kar omogoča aplikaciji JSF izkoriščanje prednosti modela CDI [4].

CDI rešuje splošne razvojne probleme, tako da uporaba tega ni omejena samo na spletne aplikacije. V okolju Java EE lahko s pomočjo CDI vstavljamo odvisnosti v sporočilna zrna, prestreznike, servlete, servletne filtre in servletne poslušalce dogodkov, storitvene končne točke, rokovalce JAX-WS ter rokovalce z oznakami in poslušalce dogodkov knjižnic oznak JSP. S pomočjo CDI nam ni več potrebno poznati odgovorov na naslednja vprašanja:

- Kakšen je življenjski cikel nekega objekta?
- Koliko ima lahko sočasnih odjemalcev?
- Ali je večniten?
- Kako dostopati do njega preko odjemalca?
- Ali ga je potrebno eksplicitno uničiti?
- Kam je potrebno shraniti njegovo referenco v času, ko ga ne uporabljamo?
- Kako lahko definiramo alternativne implementacije, tako da se te lahko spreminjajo v času namestitve?
- Kako deliti določen objekt med druge objekte?

Osrednja tema CDI je šibka sklopljenost z močnim tipiziranjem. To pomeni, da zrno poda le tip in semantiko zrn, od katerih je odvisno, pri čemer mu ni potrebno poznati življenjskega cikla, konkretne implementacije, modela nitenja ali ostalih odjemalcev kateregakoli zrna, s katerim je povezano. Še več, konkretna implementacija, življenjski cikel in model nitenja zrna, se lahko spreminjajo glede na namestitveni scenarij, ne da bi pri tem vplivali na kateregakoli odjemalca. Opisana šibka sklopljenost olajšuje vzdrževanje kode. Dogodki, prestrezniki in dekoratorji še dodatno povečujejo šibko sklopljenost omenjenega modela.

CDI omogoča vse mehanizme v močno tipiziranem načinu. Pri določanju, kateri sodelujoči objekti spadajo skupaj, se nikoli ne zanaša na identifikatorje, osnovane na nizih. Namesto tega uporablja tipizirano informacijo, ki

je razširjena s kvalifikatorskimi anotacijami. Tako lahko poveže zrna, njihove odvisnosti, prestreznike, dekoratorje in njihove odjemalce dogodkov. Minimizirana uporaba deskriptorjev XML je nujna zgolj za specifične razvojne informacije.

CDI ponuja tudi celovit SPI, s pomočjo katerega lahko vključujemo objekte, ki so definirani po bodočih specifikacijah Java EE ali po nekaterih drugih okvirih, skupaj z njihovo popolno integracijo s CDI. Na ta način je omenjenim objektom omogočena uporaba vseh prednosti storitev CDI in interakcija s katerokoli drugo vrsto zrn [3].

Poglavje 3

Zrna

3.1 Kaj je zrno

Javansko zrno (java bean) je razred, ki zadosti naslednjim kriterijem [2]:

- vsi atributi so privatni,
- ima definiran privzeti konstruktor ali konstruktor anotiran z anotacijo @Inject,
- ima ustrezne get() in set() metode za dostop do privatnih atributov,
- razred implementira vmesnik java.io.Serializable.

Preden jih začnemo vstavljati, jih je potrebno shraniti v arhiv (bodisi tipa jar, modul Java EE war ali EJB jar), ki vsebuje datoteko META-INF/beans.xml.

Večinoma imajo zrna stanje in so kontekstna. Najprej se je potrebno vprašati, kaj pomeni biti konteksten. Ker imajo zrna lahko tudi stanje, je pomembno, katera instanca zrna je dodeljena odjemalcu. Torej v nasprotju s komponentnim modelom brez stanja (kot so na primer sejna zrna brez stanja ali singleton komponente, na primer servleti ali singleton zrna), je za zrna s stanjem značilno, da jih posamezni odjemalci vidijo v različnem stanju. To stanje je odvisno od tega, katere reference instanc zrna ima določen odjemalec. V nasprotju s tem pri modelu brez stanja, odjemalec ne more eksplicitno

kreirati in izbrisati instance zrna, saj ima ta nadzor doseg. Odjemalci, kateri se izvajajo v enakem kontekstu, bodo videli tudi isto instanco zrna.

Velika prednost kontekstnega modela je, da so zrna s stanjem obravnavana kot storitve. Tako se odjemalcem zrna ni potrebno ukvarjati z upravljanjem njegovega življenjskega cikla, saj jim ga že v osnovi ni potrebno poznati [3].

3.2 Anatomija zrn

Zrno vsebuje neprazno množico tipov zrn, neprazno množico kvalifikatorjev, doseg, opcijsko ime EL, množico prestrezniških vezav in implementacijo zrna. Raziščimo, kaj pomenijo omenjeni izrazi.

3.2.1 Tipi zrn, kvalifikatorji in vstavljanje odvisnosti

Vsak vstavljen atribut določa pogodbo, katero mora vstavljajoče se zrno izpolnjevati. Pogodbo sestavlja tip zrna, skupaj z množico kvalifikatorjev. Tip zrna je pravzaprav tip, ki je viden odjemalcem. Zrno ima lahko več tipov zrn. Na primer zrno Knjigarna ima štiri tipe zrn:

public class Knjigarna extends Podjetje implements Nakup<Knjiga>{...}

Torej tipi zrn so Knjigarna, Podjetje, Nakup<Knjiga> in implicitni tip java.lang.Object. Množico tipov zrn lahko omejimo, če zrna anotiramo z anotacijo @Typed in naštejemo želene razrede. Če bi na primer pred deklaracijo zgornjega razreda Knjigarna napisali @Typed(Nakup.class), bi tip zrna omejili na Nakup<Knjiga> in na java.lang.Object.

Včasih tip zrna ne podaja dovolj informacij vsebniku, da bi ta vedel, katero zrno vstaviti. To se zgodi v primeru, če imamo za nek vmesnik več implementacij zrna. V teh primerih mora odjemalec podati kvalifikator. Kvalifikatorji so uporabniško definirane anotacije, označene z @Qualifier [3]. Več o kvalifikatorjih v poglavju *3.4.3 Kvalifikatorske anotacije*.

3.2.2 Doseg

Doseg zrna določa njegov življenjski cikel in vidnost njegovih instanc. Vsak doseg je predstavljen z anotacijo. Nekatere dosege zagotavlja vsebnik. Tak primer je na primer anotacija @SessionScoped za doseg seje. Instanca zrna z dosegom seje je omejena na uporabnikovo sejo in je deljena med vsemi zahtevki, ki se izvajajo v kontekstu seje. Ko je zrno vezano na določen kontekst, ostane v kontekstni enoti, dokler ni uničeno. Ne obstaja način, s katerim bi se dalo ročno odstraniti zrno iz konteksta, zato raje uporabimo kontekst s krajšo življenjsko dobo, kot je na primer zahtevek [3]. Več o dosegih v poglavju *3.5 Dosegi in konteksti*.

3.2.3 Ime EL

Če želimo dostopati do zrna iz kode, ki ni javanska, pri čemer podpira izrazni jezik EL (na primer strani JSP ali JSF), moramo zrnu dodeliti ime EL. V ta namen uporabljamo anotacijo @Named:

public @SessionScoped @Named("kosarica")
class NakupovanlnaKosarica implements Serializable { ... }

To nam omogoča preprosto uporabo zrna na vsaki strani JSP ali JFS:

<h:dataTable value="#{kosarica.metoda}" > ... </h:dataTable>

Če bi pustili vrednost anotacije @Named prazno, potem bi se ime EL samodejno določilo kot ime razreda, začenši z malo začetnico. Torej v zgornjem primeru bi deklarirani razred dobil ime EL nakupovanlnaKosarica [3].

3.2.4 Alternative

Do sedaj smo spoznali, da nam v času razvoja kvalifikatorji omogočajo izbiro želene implementacije vmesnika. Vendar se lahko zgodi, da imamo vmesnik (ali kakšen drug tip zrna), katerega implementacija se spreminja glede na namestitveno okolje. To se nam na primer lahko pripeti, če želimo uporabiti

testno implementacijo v testnem okolju. Alternativo deklariramo s pomočjo anotiranja zrna z anotacijo @Alternative [3, 4]. Več o alternativah v poglavju *3.4.6 Alternative*.

3.2.5 Tipi prestrezniških vezav

CDI podaja nov pristop vezave prestreznikov na zrno s pomočjo tipa prestrezniške vezave (interceptor binding type). To je uporabniško definirana anotacija, anotirana z @InterceptorBinding. Omogoča nam vezave prestreznih razredov na zrna, brez neposrednih odvisnosti med razredoma. Več o tipih prestrezniških vezav v poglavju *4.2 Prestrezniki* [3].

3.3 Vrste zrn

3.3.1 Upravljana zrna

Upravljano zrno (managed bean) je javanski razred. Eksplicitno ga lahko deklariramo tako, da razred anotiramo z anotacijo @ManagedBean, vendar to v CDI ni potrebno. V skladu s specifikacijo CDI je razred upravljano zrno, če [3]:

- ni nestatičen notranji razred,
- je konkreten razred ali anotiran z anotacijo @Decorator,
- ni anotiran s komponentno definirano anotatacijo EJB ali deklariran kot zrno EJB v ejb-jar.xml,
- ne implementira javax.enterprise.inject.spi.Extension,
- ima bodisi konstruktor brez parametrov ali konstruktor anotiran z anotacijo @Inject.

Po tej definiciji so tudi entitete JPA upravljana zrna, vendar temu ni tako, saj imajo svoj življenjski cikel, stanje in identitetni model, poleg tega pa so navadno instancirane s pomočjo JPA ali z uporabo operatorja new [2, 3].

V množico tipov upravljanih zrn, spadajo vsi razredi zrn, vsi nadrazredi in vsi vmesniki, implementirani posredno ali neposredno. Če ima upravljano zrno javen atribut, mora imeti privzeti doseg @Dependant. Podpirajo tudi povratni metodi @PostConstruct in @PreDestroy [3, 4].

3.3.2 Sejna zrna

Sejna zrna spadajo v specifikacijo EJB. Imajo poseben življenjski cikel, upravljanje stanj in model sočasnosti, ki se razlikuje od upravljanih zrn in neupravljanih javanskih objektov. Sporočilna in entitetna zrna so po naravi nekontekstni objekti, katerih ne moremo vstavljati v druge objekte. Vendar pa lahko sporočilna zrna izkoriščajo nekatere prednosti CDI, kot so vstavljanje odvisnosti, prestrezniki ter dekoratorji. CDI pravzaprav izvede vstavljanje zrn v katerokoli sejno ali sporočilno zrno, tudi če ni kontekstna instanca [3].

Množico tipov sejnih zrn sestavljajo vsi lokalni vmesniki zrn, njihovi nadvmesniki in java.lang.Object. Če ima sejno zrno lokalni pogled, potem omenjena množica vsebuje razred zrna in vse nadrazrede. Oddaljeni vmesniki ne spadajo v množico tipov zrn, saj z njimi ne moremo vstaviti sejnih zrn. To lahko storimo le, če definiramo vir komponentnega okolja Java EE [3, 4].

Določanje dosega sejnih zrn brez stanja in singleton zrn ni smiselno, ker življenjski cikel omenjenih zrn nadzoruje vsebnik EJB. Po drugi strani pa imajo lahko sejna zrna s stanjem katerikoli doseg. Uporaba sejnih zrn je smiselna, kadar potrebujemo poslovno informacijske storitve kot so: upravljanje transakcij in varnost na nivoju metod, upravljanje sočasnosti, pasivacija sejnih zrn s stanjem in pridobivanje instanc sejnih zrn brez stanja na nivoju instanc, oddaljeno proženje ali proženje spletne storitve, časovniki ter asinhrone metode [3].

3.3.3 Proizvajalne metode

Vse kar je potrebno vstaviti, niso le razredi zrn, instancirani z vsebnikom. Velikokrat se šele v času izvajanja aplikacije ugotovi, katero implementacijo

zrna je potrebno instancirati in vstaviti. Poleg tega je velikokrat potrebno vstaviti objekt, pridobljen s poizvedovanjem transakcijskega ali storitvenega vira, kot je izvajanje poizvedb JPA.

Proizvajalna metoda (producer method) je metoda, ki predstavlja vir instanc zrna. Deklaracija metode opisuje zrno. Ko v določenem kontekstu ne obstaja nobena instanca zrna, vsebnik sproži metodo, da jo pridobi. Metoda pusti aplikaciji popoln nadzor nad procesom instanciranja zrn. Deklariramo jih z anotiranjem metode zrna z anotacijo @Produces. Naslednja proizvajalna metoda vrača naključno število med 0 in 100:

```
@Produces @Named @Random int getNakljucnoStevilo() {
    return (new Random(System.currentTimeMillis())).nextInt(101);
}
```

Ne moremo napisati razreda, ki je sam po sebi naključno število, vendar lahko napišemo metodo, ki ga vrne. Lahko ji podamo kvalifikatorje (v zgornjem primeru @Random), doseg, stereotipe in ime EL (v zgornjem primeru nakljucnoStevilo). Tako lahko vstavimo naključno število kjerkoli:

```
@Inject @Random int nakljucnoStevilo;
```

Lahko ga pridobimo tudi s pomočjo izraza EL v JSF [3]. Proizvajalna metoda ne sme biti abstraktna metoda upravljanega oziroma sejnega zrna. Lahko je statična ali nestatična. V sejnem zrnu mora biti proizvajalna metoda bodisi poslovna metoda EJB ali statična metoda razreda. Množica tipov zrn proizvajalnih metod je odvisna od tipa, katerega vrača [3, 4]:

- če je vrnjeni tip vmesnik, potem množica tipov zrn vsebuje vrnjeni tip, vse posredno ali neposredno razširjene vmesnike in java.lang.Object,
- če je vrnjeni tip primitiven podatkovni tip ali javanski tip polja, potem množica tipov zrn vsebuje vrnjeni tip metode in java.lang.Object,
- če je vrnjeni tip razred, potem množica tipov zrn vsebuje vrnjeni tip, vse nadrazrede in vse posredno ali neposredno implementirane vmesnike.

Več o proizvajalnih metodah v poglavju *4.1 Proizvajalne metode*.

3.3.4 Proizvajalni atributi

Proizvajalni atribut (producer field) je preprostejša oblika proizvajalne metode. Biti mora atribut upravljanega ali sejnega zrna. Lahko je statičen ali nestatičen [4]. Deklariramo ga z anotiranjem atributa razreda zrna z anotacijo @Produces (enako kot za proizvajalne metode). Pravila za ugotavljanje tipov zrn proizvajalnih atributov so enaka pravilom proizvajalnih metod. Imajo tudi pomembno vlogo kot vmesnik za vstavljanje virov komponentnega okolja Java EE [3].

3.4 Vstavljanje odvisnosti in programsko poizvedovanje

3.4.1 Točke vstavljanja

Točka vstavljanja (injection point) je definirana s pomočjo anotacije @Inject. Vstavljanje odvisnosti se lahko opravi s tremi mehanizmi:

1. vstavljanje parametra konstruktorja zrna, pri čemer ima zrno lahko le en vstavljalni konstruktor:

   @Inject public Blagajna(Kosarica kosarica) {
   this.kosarica = kosarica;
   }

2. Vstavljanje parametra inicializacijske metode. Zrno ima lahko več inicializacijskih metod. Če je zrno sejno, potem ni obvezno, da je inicializacijska metoda poslovna metoda sejnega zrna:

   @Inject void setKosarica(Kosarica kosarica) {
   this.kosarica = kosarica;
   }

3. Neposredno vstavljanje atributa. V nasprotju z upravljanimi zrni JSF, metodi get in set nista potrebni za delovanje vstavljanja atributa:

   @Inject Kosarica kosarica;

Vstavljanje odvisnosti se vedno dogaja v času, ko vsebnik prvič instancira zrno oziroma natančneje:

1. vsebnik pokliče konstruktor zrna (privzeti konstruktor ali tistega, ki je anotiran z anotacijo @Inject), da pridobi instanco zrna,
2. vsebnik inicializira vrednosti vseh vstavljenih atributov zrna,
3. vsebnik pokliče inicializacijske metode zrna (vrstni red ni v naprej določen),
4. vsebnik pokliče metodo @PostConstruct (če obstaja).

CDI podpira tudi vstavljanje parametrov v nekatere druge metode, kot so na primer opazovalne in odstranjevalne metode. Pri vstavljanju odvisnosti v omenjene metode na točki vstavljanja anotacija @Inject ni potrebna [3].

3.4.2 Algoritem Močno tipizirana resolucija

CDI definira proceduro imenovano Močno tipizirana resolucija (Typesafe resolution), katero vsebnik uporablja, ko identificira, katero zrno mora biti vstavljeno v točko vstavljanja. Izvede se ob inicializaciji sistema. Namen algoritma je omogočiti več zrnom implementacijo enakega tipa in bodisi [3]:

- odjemalcu omogočiti izbiro implementacije s pomočjo kvalifikatorja,
- razvijalcu omogočiti izbiro implementacije, primerne za določeno namestitev, brez izvajanja sprememb na odjemalcu, s pomočjo omogočanja oziroma onemogočanja alternativ,
- omogočiti delitev zrn v ločene module.

3.4.3 Kvalifikatorske anotacije

Kvalifikatorji so uporabniško definirane anotacije označene z @Qualifier. Predstavljajo razširitev tipov zrn. Če imamo več zrn, ki implementirajo enak tip zrna, potem lahko s pomočjo kvalifikatorske anotacije natančno določimo, katero zrno mora biti vstavljeno. Predpostavimo, da imamo dve implementaciji vmesnika NacinPlacila, in sicer NacinPlacilaKartica in NacinPlacilaGotovina. Najprej je potrebno deklarirati kvalifikatorja Kartica in Gotovina. Naslednji primer kode podaja deklaracijo kvalifikatorja Kartica:

```
@Qualifier
@Target({TYPE, METHOD, PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Kartica {}
```

Definirani kvalifikator pripišemo zrnu NacinPlacilaKartica. To storimo z anotiranjem zrna s kvalifikatorjem @Kartica:

```
@Kartica public class NacinPlacilaKartica implements NacinPlacila { ... }
```

Potem lahko s kvalifikacijsko anotacijo natančno določimo, katero zrno naj bo vstavljeno v točki vstavljanja. Kot je že omenjeno, lahko to naredimo na tri načine. Naslednja koda prikazuje vstavljanje parametra konstruktorja zrna:

```
@Inject public Blagajna(@Kartica NacinPlacila nacinPlacilaKartica) {
    this.nacinPlacila = nacinPlacilaKartica;
}
```

Kvalifikator prav tako lahko kvalificira argumente proizvajalnih, opazovalnih in odstranjevalnih metod. Točka vstavljanja lahko navaja več kvalifikatorjev. V takem primeru je za vstavljanje upravičeno samo tisto zrno, ki ima vse kvalifikatorje navedene v točki vstavljanja. Zrna ali točke vstavljanja, ki nimajo navedenega kvalifikatorja, imajo le privzeti kvalifikator @Default [3, 4].

3.4.4 Vgrajena kvalifikatorja @Default in @Any

Vsakokrat, ko zrno ali točka vstavljanja eksplicitno ne podajata kvalifikatorja, vsebnik predpostavi privzeti kvalifikator @Default. Poleg tega ima vsako zrno tudi kvalifikator @Any. S podajanjem kvalifikatorja @Any na točki vstavljanja izničimo privzeti kvalifikator, ne da bi s tem omejili zrna, ki so upravičena do vstavljanja. To je še posebej uporabno, ko želimo iterirati čez vsa zrna z določenim tipom zrna, kot vidimo v naslednjem primeru [3]:

```
@Inject void iteriraj(@Any Instance<Service> storitve) {
    for (Service storitev: storitve) {...}
}
```

3.4.5 Kvalifikatorji s člani

Anotacije imajo lahko člane. Člani preprečujejo deklariranje prevelikega števila novih kvalifikatorskih anotacij. Če želimo, da vsebnik ignorira člana, ga anotiramo z anotacijo @Nonbinding.

Predpostavimo, da želimo implementirati več implementacij vmesnika NacinPlacila. Namesto kreiranja velikega števila kvalifikatorjev, ki predstavljajo različne načine plačila, jih lahko agregiramo v eno anotacijo @Placilo s članom, kar prikazuje naslednji primer:

```
@Qualifier
@Retention(RUNTIME)
@Target(METHOD, FIELD, PARAMETER, TYPE)
public @interface Placilo {
    PlacilnaMetoda value();
}
```

Nato lahko ob podajanju kvalifikatorja izberemo eno izmed možnih vrednosti članov oziroma plačilne metode [3, 4]:

```
private @Inject @Placilo(KARTICA) NacinPlacila nacinPlacilaKartica;
```

3.4.6 Alternative

Alternative so zrna, katerih implementacija je specifična za določen odjemalni modul ali namestitveni scenarij. Zrna spremenimo v alternative tako, da jih anotiramo z anotacijo @Alternative. Privzeto so alternativna zrna onemogočena. Omogočimo jih s pomočjo deskriptorja arhiva zrn beans.xml in jih s tem naredimo dostopne za instanciranje in vstavljanje. Naslednja koda iz beans.xml prikazuje omogočanje alternativnega zrna NacinPlacilaTestni:

<alternatives>
<class>si.fri.testna.NacinPlacilaTestni</class >
</alternatives>

Omogočanje velja samo za zrna v tem arhivu. Če pride do dvoumnosti v točki vstavljanja, jo vsebnik poskuša rešiti z iskanjem omogočene alternative med zrni, katera je lahko vstavljena. Če je na voljo natanko ena omogočena alternativa, bo ta vstavljena [3, 4].

3.4.7 Reševanje neizpolnjenih in dvoumnih odvisnosti

Algoritem Močno tipizirana resolucija odpove, če ne more identificirati natanko enega zrna za vstavljanje. Za reševanje neizpolnjene odvisnosti bodisi:

- ustvarimo zrno, ki implementira tip zrna in ima vse kvalifikatorske tipe točke vstavljanja,
- prepričamo se, da je zrno v classpathu modula s točko vstavljanja,
- s pomočjo datoteke beans.xml eksplicitno omogočimo alternativno zrno, ki implementira tip zrna in ima pravilne kvalifikatorske tipe.

Za popravljanje dvoumnih odvisnosti bodisi:

- za razlikovanje med dvema implementacijama tipa zrna naredimo nov kvalifikator,
- eno zrno onemogočimo z anotacijo @Alternative,

- eno izmed implementacij premaknemo v modul, ki ni v classpathu modula s točko vstavljanja,
- s pomočjo beans.xml onemogočimo eno izmed dveh alternativnih zrn.

Pomembno je, da imamo unikatno točko vstavljanja. Vendar lahko se tudi zgodi, da imamo opcijsko ali večvrednostno točko vstavljanja. Tej moramo spremeniti tip na Instance. Več o tem v poglavju *3.4.9 Pridobivanje kontekstne instance s programskim poizvedovanjem* [3].

3.4.8 Odjemalni posredniki

Posredovalni vzorec zagotavlja vmesnik za nadzorovanje dostopa do nekega drugega objekta. Pri tem sodeluje subjekt (definira skupni vmesnik za pravi subjekt in posrednika), pravi subjekt (definira objekt, ki ga predstavlja posrednik) in posrednik (nadzoruje dostop do pravega subjekta). Odjemalec pridobi referenco na posrednika in z njim rokuje enako kot s pravim subjektom [1, 7]. Opisani vzorec CDI odlično izkorišča z odjemalnimi posredniki (client proxies).

Odjemalci vstavljenega zrna imajo neposredne reference do instance zrna le, če je zrno odvisni objekt (@Dependent). To je uporabno v primeru, ko ima zrno vezano na doseg seje, neposredno referenco na zrno, vezano na doseg aplikacije. Zaradi učinkovitejše uporabe pomnilnika, se na določen interval sejni kontekst serializira na disk. Vendar instanca zrna z dosegom aplikacije ob tem ne sme biti serializirana, saj lahko kadarkoli pridobi to referenco, zato ni potrebe, da bi jo shranili. Pomembno je, da vsebnik s posredniškim objektom posreduje vse vstavljene reference do zrna. Odjemalni posredniki so odgovorni za zagotavljanje, da je instanca zrna, katere metoda se proži, tista instanca, ki je povezana s trenutnim kontekstom. Prav tako omogočajo zrnom, vezanim na kontekste, kot so sejni konteksti, da se lahko serializirajo na disk, ne da bi pri tem rekurzivno serializirali ostala vstavljena zrna [3].

Zaradi omejitev javanskega jezika, vsebnik ne more posredovati razredov, ki nimajo neprivatnega konstruktorja brez parametrov, razredov, deklarira-

nih s ključno besedo final ali z metodo deklarirano s final, polj in primitivnih tipov. To velja za zrna, ki nimajo dosega @Dependent [3, 4]. Če se točka vstavljanja tipa X izkaže kot odvisnost, katere ni mogoče posredovati, lahko bodisi tipu X dodamo konstruktor brez parametrov, lahko spremenimo tip točke vstavljanja v Instance<X> ali napišemo vmesnik Y, ki ga vstavljeno zrno implementira in spremenimo točko vstavljanja v Y. V primeru, da noben ukrep ne uspe, spremenimo doseg vstavljajočega zrna na @Dependent [3].

3.4.9 Pridobivanje kontekstne instance s programskim poizvedovanjem

Vstavljanje ni vedno najbolj priročen način za pridobivanje kontekstnih referenc. To se zgodi v naslednjih primerih:

- tipi zrna ali kvalifikatorji se spreminjajo v času izvajanja aplikacije,
- zaradi odvisnosti glede na namestitev se lahko zgodi, da nobeno zrno ne ustreza tipu in kvalifikatorjem,
- želimo iterirati čez vsa zrna določenega tipa.

V omenjenih primerih lahko aplikacija pridobi instanco določenega tipa zrna s pomočjo vmesnika Instance. Nato metoda get() vmesnika vrne kontekstno instanco zrna [3, 4]:

```
@Inject Instance<NacinPlacila> virNacinPlacila;
NacinPlacila nacinPlacila = virNacinPlacila.get();
```

Če moramo ob pridobivanju instance podati tudi kvalifikatorje, lahko to storimo z anotiranjem točke vstavljanja ali s podajanjem kvalifikatorja metodi select() vmesnika Instance. Določanje kvalifikatorja na točki vstavljanja je veliko enostavnejše, saj metoda get() vmesnika Instance samodejno ustvari kontekstno instanco s pravim kvalifikatorjem. Takšna rešitev je statična. Po drugi poti (metoda select() vmesnika Instance) lahko kvalifikator podamo

dinamično. Najprej točki vstavljanja podamo kvalifikator @Any, s čimer izničimo privzeti kvalifikator @Default [3]:

@Inject @Any Instance<NacinPlacila> virNacinPlacila;

Nato moramo pridobiti instanco našega kvalifikatorskega tipa. Ker so anotacije vmesniki, ne moramo napisati oziroma uporabiti operatorja new. Zato nam CDI dopušča pridobivanje instance kvalifikatorja tako, da ustvarimo podrazred pomožnega razreda AnnotationLiteral:

abstract class KarticaKvalifikator
extends AnnotationLiteral<Kartica> implements Kartica {}

Sedaj lahko podamo kvalifikator metodi select() razreda Instance:

NacinPlacila n = virNacinPlacila.select(new KarticaKvalifikator()).get();

Vmesnik Instance iz java.lang.Iterable deduje tudi funkcionalnost iteracije čez množico zrn z določenim tipom in kvalifikatorji [4].

3.4.10 Objekt InjectionPoint

Nekatere vrste odvisnih objektov (z dosegom @Dependent), morajo vedeti nekaj o objektih oziroma točkah vstavljanja, v katere so vstavljeni [3]. Odvisna zrna lahko vstavijo instanco InjectionPoint in dostopajo do potrebnih metapodatkov pripadajočega objekta ali točke vstavljanja. Pomembnejše metode vmesnika so: getType() in getQualifiers() (vrneta tip oziroma kvalifikatorje točke vstavljanja), getBean() (vrne zrno, ki definira točko vstavljanja) in getMember() (vrne objekt Field, Method ali Constructor, glede na to, ali gre za vstavljanje atributa, metode ali parametrov konstruktorja) [4]. Omenjeno funkcionalnosti lahko na primer uporabimo pri kategoriji beleženja za Logger, ki je odvisna od razreda, ki si jo lasti. Naslednji primer kode je ranljiv za preoblikovanje [3]:

Logger log = Logger.getLogger(NasRazred.class.getName());

V nasprotju s tem, naslednja proizvajalna metoda s pomočjo objekta InjectionPoint omogoča vstavljanje objekta JDK Logger, ne da bi ji ob tem bilo potrebno podajati kategorijo beleženja:

```
@Produces Logger ustvariLogger(InjectionPoint i) {
    return Logger.getLogger(i.getMember().getDeclaringClass().getName());
}
```

Sedaj za vstavljanje zadostuje že naslednja koda:

```
@Inject Logger log;
```

3.5 Dosegi in konteksti

Po specifikaciji CDI doseg določa [3, 4]:

- kdaj se ustvari nova instanca zrna s tem dosegom,
- kdaj se izbriše katerakoli obstoječa instanca zrna s tem dosegom,
- katere vstavljene reference se nanašajo na katerokoli instanco zrna s tem dosegom.

Če bi na primer imeli zrno z dosegom seje TrenutniUporabnik, bi vsa zrna, klicana v kontekstu iste seje, videla enako instanco TrenutniUporabnik. Instanca objekta bo samodejno ustvarjena takrat, ko bo TrenutniUporabnik prvič potreben v seji in samodejno uničena, ko bo seja končana.

Entitete JPA ne ustrezajo omenjenemu modelu, saj imajo svoj lasten življenjski cikel in identifikacijski model, ki se ne sklada z modelom, uporabljenim v CDI. Zato je priporočljivo, da se entitete JPA ne obravnavajo kot zrna CDI. Zagotovo bomo naleteli na težave, če bomo entitetam dodelili kateri drugi doseg od privzetega @Dependent [2, 3].

3.5.1 Kreiranje novih dosegov

CDI zajema razširljiv kontekstni model, kar dopušča ustvarjanje novih dosegov tako, da kreiramo novo anotacijo tipa dosega. To storimo s pomočjo anotacije @ScopeType:

```
@ScopeType
@Retention(RUNTIME)
@Target(TYPE, METHOD)
public @interface NovDoseg {}
```

Da bo deklarirani doseg uporaben, moramo definirati objekt razreda Context, ki implementira definirani doseg. Ta naloga je navadno zelo tehničnega tipa, namenjena le za razvijanje ogrodja. Običajno bodo zadostovali že vgrajeni dosegi CDI [3].

3.5.2 Vgrajeni dosegi

CDI definira štiri vgrajene dosege, in sicer doseg zahtevka (@RequestScoped), doseg seje (@SessionScoped), doseg aplikacije (@ApplicationScoped) in doseg pogovora (@ConversationScoped). V vsaki spletni aplikaciji, ki uporablja CDI, ima vsaka servletna zahteva dostop do aktivnih dosegov zahteve, seje in aplikacije. Poleg tega ima tudi vsaka zahteva JSF dostop do aktivnih dosegov pogovora. Razširitev CDI lahko implementira podporo za doseg pogovora v drugih spletnih ogrodjih. Doseg zahteve in aplikacije sta aktivna:

- med proženjem oddaljene metode EJB,
- med proženjem asinhrone metode EJB,
- med časovnim nadzorom EJB,
- med dostavo sporočila sporočilnemu zrnu,
- med dostavo sporočila objektu, ki implementira MessageListener,
- med proženjem spletne storitve.

Če aplikacija poskuša prožiti zrno, ki nima dosega z aktivnim kontekstom, bo prišlo do napake. Poleg tega morajo biti vsa upravljana zrna z dosegom seje in pogovora serializirana, ker vsebnik na določen interval pasivira sejo HTTP [3].

3.5.3 Doseg pogovora

Doseg pogovora hrani stanje, povezano z uporabnikom sistema in se razteza čez več zahtevkov na strežnik. Zato delno spominja tudi na doseg seje, vendar je v nasprotju z njim:

- je razmejen (demarcated) eksplicitno s strani aplikacije,
- v aplikaciji JSF hrani stanje povezano s točno določenim zavihkom spletnega brskalnika (brskalniki si pogosto delijo domenske piškotke in s tem tudi sejni piškotek, kar ne velja za doseg seje).

Pogovorni kontekst hrani stanje, povezano s tem, na čimer trenutno dela uporabnik. Če uporabnik dela več stvari naenkrat, potem imamo več pogovorov. Pogovorni kontekst je aktiven med vsako zahtevo JSF. Večina pogovorov je uničenih na koncu zahtevka. Če bi pogovor moral hraniti stanje čez več zahtevkov, bi moral biti eksplicitno povišan v dolgo trajajoči pogovor.

Vmesnik Conversation

CDI ponuja vgrajeno zrno Conversation, ki je namenjeno nadzorovanju življenjskega cikla pogovora. Pridobimo ga lahko z naslednjim vstavljanjem:

@Inject Conversation pogovor;

Da povišamo pogovor, povezan s trenutno zahtevo, na dolgo trajajoči pogovor, pokličemo metodo begin(). Za uničenje trenutnega dolgo trajajočega pogovornega konteksta na koncu trenutne zahteve pokličemo metodo end().

Pogovorni kontekst se samodejno razširja (propagates) z vsakim zahtevkom JSF (JSP pošiljanje obrazcev) ali s preusmeritvijo, pri čemer se ne

razširja samodejno z zahtevki, ki ne prihajajo od JSF, kot je na primer navigacija preko povezave. Če dodamo enolični identifikator pogovora kot parameter zahteve, lahko dosežemo, da pogovor razširja tudi take zahtevke. Specifikacija CDI ima v ta namen parameter zahtevka, imenovan cid. Enolični identifikator pogovora lahko pridobimo iz objekta Conversation. V naslednjem primeru kode vidimo povezavo, ki razširja pogovor:

Dodaj izdelek

Vsebniku je zaradi varčevanja z viri kadarkoli dovoljeno uničevanje pogovorov in vseh stanj v njihovem kontekstu. Aplikacija bo to običajno storila na podlagi neke vrste časovnega nadzora. To je čas dovoljene neaktivnosti, preden je pogovor uničen. Podamo ga lahko s pomočjo metode setTimeout objekta Conversation, ki ji za argument podamo število milisekund. To je le namig vsebniku, saj lahko ignorira omenjeno nastavitev [3].

3.5.4 Singleton psevdo doseg

Poleg štirih vgrajenih dosegov, CDI podpira tudi dva psevdo dosega. Prvi je singleton psevdo doseg, ki ga podamo z anotacijo @Singleton. Singleton vzorec nam zagotavlja, da ima razred samo eno instanco, do katere je možno globalno dostopati [1]. Pri tem dosegu se pojavlja težava. Zrna z dosegom singleton nimajo posredniškega objekta, torej morajo imeti odjemalci neposredno referenco na njihovo instanco. Singleton instance se lahko tudi serializirajo. Omenjeni funkcionalnosti predstavljata težavo pri odjemalcih singleton zrn, ki se lahko serializirajo, kar pomeni, da se lahko instance singleton zrn s pomočjo serializacije odjemalcev duplicirajo. To posledično privede do tega, da tako zrno ni več singleton in bi ga lahko deklarirali s privzetim dosegom. Obstaja več načinov reševanja opisane težave:

- singleton zrno implementira writeResolve() in readReplace(),
- zagotovimo, da ima odjemalec samo začasno referenco na singleton zrno,

- odjemalcu podamo referenco tipa Instance<X>, kjer je X tip zrna singleton zrna.

Najboljša rešitev je uporaba dosega aplikacije, kar vsebniku omogoča posredovanje zrna in s tem samodejno rešitev problema serializacije [3].

3.5.5 Odvisni psevdo doseg

CDI zajema tudi tako imenovani odvisni psevdo doseg. Določa ga anotacija @Dependent. To je privzeti doseg za zrna. Zrna s tem dosegom ne potrebujejo posredniških objektov, ker imajo njihovi odjemalci neposredno referenco do njihovih instanc. Intancirana so takrat, ko je ustvarjen objekt, kateremu pripadajo in odstranjena, ko je odstranjen objekt, kateremu pripadajo. Vezana so na življenjski cikel objekta, v katerega so vstavljena [3, 4].

3.5.6 Kvalifikator @New

Vgrajeni kvalifikator @New omogoča pridobivanje odvisnih objektov. Razred mora biti obvezno veljavno upravljano ali sejno zrno, vendar ne nujno omogočeno zrno. To deluje tudi v primeru, če je želeni razred deklariran z drugačnim tipom dosega. Če na točki vstavljanja podamo anotacijo @New, potem vstavljeni objekt ne bo imel dosega deklariranega v definiciji razreda, ampak bo njegov življenjski cikel vezan na starševski razred [3].

Poglavje 4

Šibka sklopljenost in močno tipiziranje

Prva pomembna tema CDI je šibka sklopljenost (loose coupling). Spoznali smo že tri pomene doseganja šibke sklopljenosti. Alternative omogočajo polimorfizem v času namestitve, proizvajalne metode omogočajo polimorfizem v času izvajanja in upravljanje življenjskih ciklov kontekstov je ločeno od življenjskih ciklov zrn. Omenjene tehnike so namenjene omogočanju šibke sklopljenosti odjemalca in strežnika. Odjemalci niso več tesno povezani z implementacijo vmesnika in hkrati odjemalcem ni več potrebno upravljati življenjskega cikla implementacije. Ta pristop zrnom s stanjem omogoča interakcijo, kot če bi bila storitve. CDI je prva tehnologija in prva specifikacija na platformi Java EE, ki omogoča šibko sklopljenost z močnim tipiziranjem. Ravno tako ponuja tri pomembne sposobnosti, ki izpopolnjujejo šibko sklopljenost:

- prestrezniki ločujejo tehnične probleme od poslovne logike,
- uporaba dekoratorjev za ločevanje določene poslovne logike,
- dogodkovna obvestila ločijo povzročitelje dogodkov od odjemalcev.

Druga pomembna tema CDI je močno tipiziranje, zaradi katerega ne potrebujemo identifikatorjev, osnovanih na nizih. Prednost takega pristopa je

tudi v tem, da lahko vsak IDE ponudi samodejno izpolnjevanje (autocomplete), validacijo in preoblikovanje (refactoring), brez uporabe specifičnih orodij. CDI spodbuja razvoj anotacij, ker jih lahko ponovno uporabimo. Pomagajo nam tudi kategorizirati in razumeti kodo. Stereotipi to idejo še bolj izpopolnjujejo, saj enkapsulirajo različne vloge. Več o stereotipih v poglavju *4.5 Stereotipi*. V naslednjih poglavjih se bomo osredotočili na bolj komplekse funkcionalnosti CDI [3].

4.1 Proizvajalne metode

Proizvajalne metode (producer methods) nam pomagajo zaobiti določene omejitve, ki se pojavijo takrat, ko je za instanciranje objektov namesto aplikacije odgovoren vsebnik. So tudi najenostavnejši način za integracijo tistih objektov, ki niso zrna, v okolje CDI. Imajo vlogo vira vstavljajočih se objektov kjer [3]:

- vstavljajočim se objektom ni potrebno, da so instance zrna,
- konkretni tip zrna vstavljajočega se objekta se lahko spreminja med časom izvajanja,
- objekti potrebujejo inicializacijo po meri, ki je ne izvede konstruktor.

4.1.1 Doseg proizvajalnih metod

Privzeti doseg proizvajalne metode je odvisni doseg, kar pomeni, da bo poklicana vsakokrat, ko bo vsebnik vstavil katerikoli atribut, proizveden v njej. Takšno obnašanje spremenimo tako, da proizvajalni metodi dodamo antoacijo (na primer) @SessionScoped. V trenutku, ko bo proizvajalna metoda poklicana, bo vrnjeni objekt vezan na kontekst seje, zato ta ne bo več poklicana v isti seji.

Zelo pomembno je, da proizvajalna metoda ne podeduje dosega zrna, ki jo deklarira. Tukaj gre pravzaprav za dve različni zrni, natančneje za proizvajalno metodo in za zrno, ki jo deklarira. Doseg proizvajalne metode določa,

kako pogosto bo metoda poklicana in življenjski cikel objekta, katerega metoda vrne. Doseg zrna, ki deklarira proizvajalno metodo, določa življenjski cikel objekta, ki proži proizvajalno metodo [3].

4.1.2 Vstavljanje v proizvajalne metode

Vzemimo za primer naslednjo proizvajalno metodo:

```
@Produces @Kvalifikator public NacinPlacila getPlacilo() {
    switch (nacinPlacila) {
      case KARTICA: return new NacinPlacilaKartica();
      case CEK: return new NacinPlacilaCek();
    }
}
```

Pri kodi v zgornjem primeru se pojavlja težava, saj sta implementaciji NacinPlacila instancirani z uporabo operatorja new. Objekti, ki jih instancira aplikacija, ne morejo izkoriščati prednosti vstavljanja odvisnosti in ne morejo imeti prestreznikov. Temu se lahko izognemo tako, da za pridobitev instanc zrna uporabimo vstavljanje odvisnosti v proizvajalno metodo:

```
@Produces @Kvalifikator @SessionScoped
public NacinPlacila getPlacilo(NacinPlacilaKartica k, NacinPlacilaCek c) {
    switch (nacinPlacila) {
      case KARTICA: return k;
      case CEK: return c;
    }
}
```

Predpostavimo, da se vstavi objekt NacinPlacilaKartica. Če je NacinPlacilaKartica zrno z dosegom zahteve, potem ima proizvajalna metoda funkcionalnost povišanja trenutne instance (zaradi anotacije @SessionScoped) v instanco z dosegom seje. Vsebnik bo uničil vrnjeni objekt z dosegom zahteve, še preden bo seja zaključena, pri čemer bo referenca na objekt še vedno

ostala v dosegu seje. Te napake ne bo zaznal. To lahko rešimo na najmanj tri načine. Lahko spremenimo doseg implementacije NacinPlacilaKartica, vendar bi to bi vplivalo na druge odjemalce omenjenega zrna. Boljša možnost je sprememba dosega proizvajalne metode na @Dependent ali @RequestScoped. Najbolj pogosta rešitev je uporaba kvalifikatorske anotacije @New:

```
@Produces @Kvalifikator @SessionScoped
public NacinPlacila getPlacilo(@New NacinPlacilaKartica k,
                               @New NacinPlacilaCek c) {
    switch (nacinPlacila) {
      case KARTICA: return k;
      case CEK: return c;
    }
}
```

Predpostavimo zopet, da se vstavi objekt NacinPlacilaKartica. To pomeni, da bo ustvarjena nova odvisna instanca NacinPlacilaKartica, ki bo predana proizvajalni metodi, zatem jo bo ta vrnila in bo na koncu vezana na kontekst seje. Odvisni objekt ne bo odstranjen, dokler ne bo na koncu seje odstranjen objekt razreda, v katerem je deklarirana proizvajalna metoda [3].

4.1.3 Odstranjevalne metode

Nekatere proizvajalne metode vračajo objekte, ki potrebujejo eksplicitno odstranjevanje, kot je na primer zapiranje povezave JDBC:

```
@Produces @RequestScoped Connection povezava(User uporabnik) {
    return ustvariPovezavo(uporabnik.getId(), uporabnik.getGeslo());
}
```

Odstranjevalna metoda (disposer method) omogoča odstranjevanje objektov, ki jih vračajo proizvajalne metode in atributi. Imeti mora natanko en parameter, anotiran z @Disposes, kateri mora imeti enak tip in kvalifikatorje,

kot proizvajalna metoda. Definirana mora biti v istem razredu kot proizvajalna metoda. Naslednja odstranjevalna metoda zapre povezavo, katero je ustvarila zgornja proizvajalna metoda [3]:

```
void zapriPovezavo(@Disposes Connection povezava) {
    povezava.close();
}
```

Odstranjevalna metoda je samodejno poklicana, ko se konča kontekst (v zgornjem primeru je to po koncu zahtevka). Takrat parameter metode prejme objekt, katerega je ustvarila proizvajalna metoda. Če ima odstranjevalna metoda dodatne parametre, ti predstavljajo točke vstavljanja [3, 4].

4.2 Prestrezniki

Preden podrobneje raziščemo prestreznike, se je najprej potrebno vprašati, kakšen je prestrezni vzorec? Uporablja se, ko želimo spremeniti običajen življenjski cikel programa oziroma ogrodja, pri čemer je sprememba transparentna in se zgodi samodejno. Ostalemu delu sistema ni potrebno vedeti, da je bilo nekaj dodano ali spremenjeno, torej lahko deluje kot prej. Vzorec sestavlja implementacija vnaprej definiranega vmesnika za razširitve, odpremni (dispatching) mehanizem za registriranje prestreznikov in kontekstni objekti, ki omogočajo dostop do notranjega stanja ogrodja [5].

Funkcionalnost prestreznikov je definirana v specifikaciji Java Interceptors. Ta definira dve vrsti točk prestrezanja, in sicer prestrezanje poslovnih metod in prestrezanje povratnih metod življenjskega cikla. Specifikacija EJB poleg tega definira tudi prestrezanje metod s časovnim nadzorom. Prestreznik poslovnih metod prestreza proženja metod zrna s strani odjemalca zrna. Za implementacijo prestrezne metode uporabimo anotacijo @AroundInvoke [2, 3]. Prestreznik povratnih metod življenjskega cikla prestreza proženja povratnih metod življenjskega cikla s strani vsebnika. Za to uporabimo anotacije @PostConstruct (izvede se po vstavljanju odvisnosti in pred izvajanjem zrna), @PreDestroy (izvede se pred odstranjevanjem zrna iz vsebnika),

@PrePassivate (izvede se, preden gre zrno v stanje pasivnosti) in @PostActivate (izvede se po vrnitvi zrna iz stanja pasivnosti) [2]. Prestrezni razredi, katere definiramo z anotacijo @Interceptor, lahko prestrežejo tako poslovne metode kot tudi povratne metode življenjskega cikla. Prestreznik metode s časovnim nadzorom prestreza proženja metod EJB s časovnim nadzorom s strani vsebnika. Za omenjene metode uporabimo anotacijo @AroundTimeout. Prestrezniki lahko tudi izkoristijo vstavljanje odvisnosti [2, 3].

4.2.1 Prestrezniške vezave

Predpostavimo, da želimo deklarirati nekaj zrn, ki so transakcijska. V ta namen potrebujemo tip prestrezniške vezave, s katerim določimo želena zrna:

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Transakcijski {}
```

Nato tip prestrezniške vezave @Transakcijski vežemo na prestreznik:

```
@Transakcijski @Interceptor class TransakcijskiPrestreznik {
    @AroundInvoke
    public Object upravljanjeTransakcije(InvocationContext ctx)
                                            throws Exception { ... }
}
```

Nazadnje tip prestrezniške vezave @Transakcijski vežemo na zrno:

```
public @SessionScoped @Transakcijski
class NakupovalnaKosarica implements Serializable { ... }
```

Pri tem je zelo pomembno dejstvo, da TransakcijskiPrestreznik in NakupovalnaKosarica drug o drugemu nič ne vesta. Tip prestrezniške vezave lahko vežemo tudi na metodo. Če želimo vezati več prestreznikov na zrno, potem običajno uporabljamo kombinacijo tipov prestrezniških vezav [3].

4.2.2 Omogočanje in razvrščanje prestreznikov

Privzeto so vsi prestrezniki onemogočeni. Omogočimo jih s pomočjo deskriptorja beans.xml arhiva zrn. Ta aktivacija velja le za zrna v omenjenem arhivu:

```
<interceptors>
    <class>si.fri.aplikacija.TransakcijskiPrestreznik </class>
</interceptors>
```

Z deklaracijo XML rešimo dva problema:

- omogoča nam definiranje vrstnega reda vseh prestreznikov v sistemu, kar zagotavlja deterministično vedenje,
- dopušča nam omogočanje oziroma onemogočanje prestreznikov v času namestitve.

Vrstni red izvajanja prestreznikov je enak vrstnemu redu, v katerem so navedeni v deskriptorju beans.xml. Če v deskriptorju ne navedemo nobenega prestreznika, so vsi onemogočeni [3, 4].

4.2.3 Prestrezniške vezave s člani

Predpostavimo, da želimo anotaciji Transakcijski dodati nekaj informacij:

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Transakcijski {
    boolean requiresNew() default false;
}
```

CDI bo uporabil vrednost requiresNew za izbiro med dvema različnima prestreznikoma, in sicer TransakcijskiPrestreznik in TransakcijskiPrestreznikRequiresNew. Naslednja koda prikazuje deklaracijo prestreznika TransakcijskiPrestreznikRequiresNew [3]:

@Transakcijski(requiresNew=true) @Interceptor
public class TransakcijskiPrestreznikRequiresNew {...}

Nato lahko uporabimo prestreznik TransakcijskiPrestreznikRequiresNew:

@Transakcijski(requiresNew=true) public class NakupovalnaKosarica {...}

4.2.4 Dedovanje tipa prestrezniške vezave

Ena izmed omejitev javanskega jezika pri anotacijah je pomanjkanje anotacijskega dedovanja. Na srečo CDI ponuja rešitev za omenjen problem. Nek tip prestrezniške vezave lahko anotiramo z drugim tipom prestrezniške vezave (metaanotacija). Prestrezniške vezave so tranzitivne, kar pomeni, da lahko vsako zrno s prestrezniško vezavo deduje prestrezniške vezave deklarirane kot metaanotacije.

@TipPrestezniskeVezave1 @TipPrestezniskeVezave2
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface NovTipPrestreznikeVezave {}

Sedaj bo vsako zrno, anotirano z @NovTipPrestreznikeVezave, vezano na prestreznike, na katere sta vezani metaanotaciji @TipPrestezniskeVezave1 in @TipPrestezniskeVezave2 [3].

4.2.5 Uporaba @Interceptors

Anotacijo @Interceptors je uporabljena v upravljanih zrnih in specifikaciji EJB. Še vedno je podprta v CDI [3]:

@Interceptors(Prestreznik1.class, Prestreznik2.class)
public class NakupovalnaKosarica {}

Ta pristop ima naslednje slabosti:

- implementacija prestreznika je zakodirana v poslovni kodi,
- v času namestitve ni enostavno onemogočiti prestreznikov,
- vrstni red prestreznikov ni globalen, ampak je določen tako, kot so prestrezniki navedeni na nivoju razreda.

4.3 Dekoratorji

Dekoratorski vzorec nam omogoča dinamično dodajanje vedenja nekemu objektu v času izvajanja aplikacije. Pri tem sodelujejo komponenta (definira vmesnik za objekte, ki jim lahko dinamično dodamo vedenje), konkretna komponenta (definira objekt, kateremu lahko dodamo vedenje), dekorator (hrani referenco na objekt komponente in definira vmesnik, ki se sklada z vmesnikom komponente), ter konkretni dekorator (doda dodatno vedenje komponenti oziroma prepiše metode, ki jim je potrebno spremeniti vedenje) [1]. Podrazred doda vedenje v času prevajanja, kar pomeni, da sprememba vpliva na vse instance prvotnega razreda, zgoraj opisano ovijanje pa nam omogoča dodajanje vedenja v času izvajanja, za posamezne objekte [6].

CDI odlično izkorišča opisani dekoratorski vzorec. Prestrezniki predstavljajo odlično orodje za reševanje tehničnih problemov, kot so upravljanje transakcij in varnost. Vendar ne poznajo semantike dogodka, ki ga prestrezajo. Zato niso primerno orodje za reševanje poslovnih problemov. Nasprotno velja za dekoratorje. Dekorator je zrno, katero implementira tip, ki ga dekorira in je anotiran z @Decorator. Implementira metode dekoratorskega tipa, ki ga želi prestreči. S tem posledično pozna celotno semantiko vezano na ta vmesnik. Poglejmo si primer, v katerem se bolje se bolje obnese dekorator. Naslednji vmesnik predstavlja bančne račune:

```
public interface Racun {
    public void dvig(BigDecimal znesek);
    public void nakazilo(BigDecimal znesek);
}
```

Predpostavimo tudi, da več različnih zrn implementira vmesnik Racun. Pri tem imamo zahtevo, da mora sistem za katerokoli vrsto računa zabeležiti velike transakcije, kar predstavlja odličen primer za uporabo dekoratorja:

@Decorator public abstract class Dekorator implements Racun { ... }

V nasprotju z ostalimi zrni je dekorator lahko abstrakten razred, saj mu ni potrebno implementirati vseh metod dekoriranega vmesnika [3].

4.3.1 Delegiran objekt

Dekorator ima posebno točko vstavljanja, imenovano delegirana točka vstavljanja (delegate injection point), z enakim tipom zrna, kot zrna, ki jih dekorira in je anotirana z anotacijo @Delegate. Obstajati mora natanko ena delegirana točka vstavljanja. Dekorator je vezan na vsako zrno, ki ima enak tip zrna in vse kvalifikatorje deklarirane v delegirani točki vstavljanja. Naslednja delegirana točka vstavljanja določa, da je dekorator vezan na zrna z vmesnikom Racun:

@Inject @Delegate @Any Racun racun;

Dekorator lahko proži delegiran objekt (delegate object), kar ima enak učinek kot klic InvocationContext.proceed() iz prestreznika. Glavna razlika je, da dekorator lahko proži katerokoli poslovno metodo delegiranega objekta [3, 4].

4.3.2 Omogočanje dekoratorjev

Dekoratorji so privzeto onemogočeni. Omogočimo jih s pomočjo deskriptorja beans.xml arhiva zrn. Ta aktivacija velja le za zrna v omenjenem arhivu:

<decorators>
    <class>si.fri.aplikacija.Dekorator </class>
</decorators>

Prikazana deklaracija nam omogoča definiranje vrstnega reda izvajanja vseh dekoratorjev v sistemu, kar zagotavlja deterministično vedenje, obenem pa zagotavlja omogočanje in onemogočanje dekoratorjev v času namestitve [3].

4.4 Dogodki

Vstavljanje odvisnosti zagotavlja šibko sklopljenost tako, da omogoča implementaciji vstavljenega tipa zrna spreminjanje, bodisi v času namestitve ali v času izvajanja. Dogodki gredo še en korak naprej, saj zrnom dopuščajo interakcijo brez katerekoli odvisnosti v času prevajanja. Povzročitelji dogodkov sprožijo dogodke, ki jih vsebnik dostavi opazovalcem dogodkov. Dogodkovni objekt ni nič drugega kot instanca konkretnega javanskega razreda. Ne samo, da so povzročitelji dogodkov ločeni od opazovalcev, ampak velja tudi obratno. Poleg tega lahko opazovalci podajo kombinacijo selektorjev za zmanjšanje množice prejetih dogodkovnih obvestil. Lahko so obveščeni takoj ali pa določijo zakasnitev dostave dogodka do konca trenutne transakcije [3].

4.4.1 Opazovalci dogodkov

Opazovalni (observer) vzorec omogoča, da objekt, imenovan subjekt, hrani odvisnosti (imenovane opazovalci) in jih obvesti, če je prišlo do spremembe stanja (navadno s klicem ene izmed njihovih metod) [7]. Pri tem sodeluje več gradnikov. Subjekt zagotavlja vmesnik za priključitev in sprostitev opazovalnih objektov. Konkretni subjekt ob spremembi stanja pošlje obvestilo svojim opazovalcem. Opazovalec definira posodobitveni vmesnik za objekte, ki morajo biti obveščeni o spremembi subjekta. Konkretni opazovalec implementira posodobitveni vmesnik opazovalca, da lahko ohranja svoje stanje konsistentno s subjektom [1]. Opazovalni vzorec CDI odlično izkorišča.

V CDI je opazovalna metoda tista metoda zrna (oziroma konkretnega opazovalca), ki ima parameter anotiran z @Observes (dogodkovni parameter). Tip dogodkovnega parametra je opazovani tip dogodka (v spodnjem primeru Dokument). Če želimo zmanjšati množico opazovanih dogodkov, ima dogodkovni parameter lahko tudi kvalifikatorje. Opazovalna metoda ima lahko dodatne parametre, ki predstavljajo točke vstavljanja [3, 4]:

public void onUpdate(@Observes @Kvalifikator Dokument d, Param p){...}

4.4.2 Povzročitelji dogodkov

Povzročitelji dogodkov (event producers) sprožijo dogodke s pomočjo instance parameteriziranega vmesnika Event, ki jo pridobimo z naslednjim vstavljanjem:

@Inject @Any Event<Dokument> dokDogodek;

Dogodek se sproži s klicem metode fire() vmesnika Event, pri čemer je potrebno podati še dogodkovni objekt:

dokDogodek.fire(dokument);

Ta dogodek bo dostavljen vsem opazovalnim metodam z dogodkovnim parametrom, na katerega je možno pripisati dogodkovni objekt (v našem primeru Dokument) in nimajo nobenih kvalifikatorjev. Vsebnik pokliče opazovalno metodo in ji ob tem preda dogodkovni objekt kot vrednost dogodkovnega parametra.

Na dogodke lahko dodamo tudi kvalifikatorje. To storimo bodisi z anotiranjem točke vstavljanja Event ali s posredovanjem kvalifikatorjev metodi select() vmesnika Event. Prva rešitev je preprostejša:

@Inject @Kvalifikator Event<Dokument> posodobitevDokumentaDogodek;

Slabost anotiranja točke vstavljanja je, da ne moremo dinamično podati kvalifikatorjev. CDI omogoča pridobivanje instance kvalifikatorja tako, da zanj ustvarimo podrazred pomožnega razreda AnnotationLiteral. Tako lahko kvalifikator podamo metodi select() vmesnika Event [3]:

dokDogodek.select(new AnnotationLiteral<Kvalifikator>()).fire(dokument);

4.4.3 Pogojne opazovalne metode

Če v trenutnem kontekstu ni nobene instance opazovalca, jo bo vsebnik privzeto instanciral, da ji bo lahko dostavil dogodek. Lahko se zgodi, da želimo dogodke dostaviti samo obstoječim instancam opazovalca. To lahko storimo

s pomočjo pogojnih opazovalcev. Pri tem velja, da odvisno zrno ne more biti pogojni opazovalec, saj ne bi bilo nikoli poklicano. Pogojnega opazovalca določimo z dodajanjem receive=IF_EXISTS anotaciji @Observes [3]:

public void refreshOnPosodobitevDok(@Observes(receive=IF_EXISTS)
@Kvalifikator Dokument d) {...}

4.4.4 Dogodkovni kvalifikatorji s člani

Dogodkovni kvalifikator ima lahko anotacijske člane. Vrednost člana se uporablja za zmanjšanje množice opazovalcev:

public void prijavljenAdmin(@Observes @Vloga(ADMIN) Prijavljen p) {...}

Povzročitelj dogodka lahko s pomočjo anotacij na točki vstavljanja pri obveščevalcu dogodka statično določi vrednost člana dogodkovnega kvalifikatorja. Alternativno je lahko vrednost člana dogodkovnega kvalifikatorja dinamično določena s pomočjo povzročitelja dogodka. To storimo tako, da najprej ustvarimo abstrakten podrazred razreda AnnotationLiteral:

abstract class Vez extends AnnotationLiteral<Vloga> implements Vloga {}

Povzročitelj dogodka preda instanco omenjenega razreda metodi select() [3]:

```
dokDogodek.select(new Vez() {
    public void value() { return uporabnik.getVloga(); }
}).fire(dokument);
```

4.4.5 Transakcijski opazovalci

Transakcijski opazovalci prejmejo obvestilo o dogodku pred oziroma po končani fazi transakcije, v kateri je bil dogodek sprožen. Obstaja pet vrst transakcijskih opazovalcev, katere določimo z dodajanjem parametra during=X anotaciji @Observes, pri čemer je lahko vrednost spremenljivke X bodisi:

- IN_PROGESS (poklicani takoj (privzeta vrednost)),

- AFTER_SUCCESS (poklicani po končani fazi uspešne transakcije),
- AFTER_FAILURE (poklicani po končani fazi neuspešne transakcije),
- AFTER_COMPLETION (poklicani po končani fazi transakcije),
- BEFORE_COMPLETION (poklicani pred končanjem faze transakcije).

Transakcijski opazovalci so zelo pomembni v modelu objektov s stanjem, saj je stanje pogosto shranjeno za več časa kot traja ena sama transakcija [3].

4.5 Stereotipi

V mnogih sistemih uporaba arhitekturnih vzorcev povzroča množico ponavljajočih se vlog zrn. Stereotip nam omogoča identifikacijo takih vlog in deklariranje pogostih metapodatkov za zrna s to vlogo na enem mestu.

Stereotip je anotacija ozrnačena z @Stereotype in zajema več drugih anotacij. Lahko določa privzet doseg zrn, množico prestrezniških vezav in privzeto ime EL (če ime EL ni že eksplicitno definirano na zrnu s tem stereotipom). Lahko tudi določi, če je zrno alternativa. Z njegovo pomočjo lahko zrna podedujejo zgoraj navedene funkcionalnosti. Stereotipne anotacije so lahko dodane razredu zrna, proizvajalni metodi ali atributu. Poglejmo si primer definiranja stereotipa:

```
@RequestScoped
@PrestreznikaVezava1(requiresNew=true)
@PrestreznikaVezava2
@Named
@Alternative
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Stereotip {}
```

Stereotip uporabimo tako, da zrnu dodamo anotacijo @Stereotip. Naslednje zrno bo zaradi anotacije @Named dobilo ime EL stereotipniRazred [3, 4]:

@Stereotip public class StereotipniRazred { ... }

4.5.1 Stereotipno skladanje

Stereotipi lahko deklarirajo druge stereotipe, kar imenujemo stereotipno skladanje (stereotype stacking). To je uporabno v primerih, če imamo dva ločena stereotipa, ki sta sama po sebi pomembna, pri čemer sta lahko pomembna tudi v drugih situacijah, ko sta združena [3].

4.5.2 Vgrajeni stereotipi

Dva stereotipa, definirana v specifikaciji CDI, smo že spoznali. To sta @Interceptor in @Decorator. CDI poleg naštetega definira še @Model, ki se uporablja predvsem v spletnih aplikacijah JSF. Namenjen je zrnom, ki predstavljajo plast modela v vzorcu MVC [4].

4.6 Specializacija, dedovanje in alternative

Specializacijska, dedovalna in alternativna pravila je potrebno podrobneje preučiti, če želimo rešiti neizpolnjene oziroma dvoumne odvisnosti, ali se želimo izogniti klicem neželenih zrn. Specifikacija CDI podaja dve pravili oziroma scenarija, v katerih eno zrno razširja drugo [3, 4]:

1. drugo zrno v določenih namestitvenih scenarijih specializira prvo. V takih namestitvah drugo zrno popolnoma nadomesti prvega,
2. drugo zrno ponovno uporabi implementacijo prvega zrna in nima nobene povezave z njim. Ni obvezno, da je prvo zrno ustvarjeno za uporabo kot kontekstni objekt.

CDI privzeto predpostavlja drugi primer. Prvi primer je izjema in zahteva tudi več pozornosti. V določeni namestitvi lahko samo eno zrno naenkrat izpolni vlogo. To pomeni, da mora biti samo eno zrno omogočeno in ostala

onemogočena. V to sta vpletena dva modifikatorja in sicer @Alternative (alternative) in @Specializes (specializacije) [3].

4.6.1 Uporaba alternativnih stereotipov

Spoznali smo že, kako lahko omogočimo alternativo s tem, ko navedemo njen razred v deskriptorju beans.xml. Predpostavimo, da imamo veliko število alternativ v testnem okolju. Bilo bi veliko bolj priročno, če bi jih lahko omogočili vse naenkrat. Vzemimo za primer, da imamo stereotip @Testno, katerega smo naredili alternativnega. Nato testna zrna anotiramo s stereotipom @Testno, namesto z anotacijo @Alternative. Kot prikazuje naslednja koda iz beans.xml, nam to zelo olajša omogočanje testnih zrn. To lahko sedaj opravimo z eno vrstico kode v beans.xml, ne glede na število testnih alternativnih zrn [3]:

<alternatives>
<stereotype>si.fi.aplikacija.Testno </stereotype>
</alternatives>

4.6.2 Težava z alternativami

V nekaterih primerih omogočanje alternativne implementacije ne zadostuje, da onemogočimo privzeto implementacijo. Namreč, če ima privzeta implementacija kvalifikator, pri čemer ga alternativa nima, potem lahko še vedno s pomočjo kvalifikatorja vstavimo privzeto implementacijo. Edini način, da določeno zrno povsem prepiše neko drugo zrno v točki vstavljanja je, da implementira vse tipe zrna in deklarira vse kvalifikatorje drugega zrna. Če drugo zrno deklarira proizvajalno ali opazovalno metodo, potem tudi to ni dovolj dobro zagotovilo, da drugo zrno ne bo nikoli poklicano. CDI v ta namen ponuja posebno funkcionalnost, imenovano specializacija. To je pravzaprav način informiranja sistema, da želimo popolnoma zamenjati in onemogočiti določeno implementacijo zrna [3, 4].

4.6.3 Uporaba specializacije

Za preprečevanje napak ob zamenjavi implementacije prvega zrna z nekim drugim zrnom lahko drugo zrno bodisi:

- neposredno razširja razred prvega zrna,
- v primerih, ko je prvo zrno proizvajalna metoda, lahko neposredno prepiše le-to in potem eksplicitno deklarira, da specializira prvo zrno:

  public @Alternative @Specializes
  class NacinPlacilaKarticaTest extends NacinPlacilaKartica {...}

Specializacija temelji na dedovanju. Ker vsebnik informiramo, da je alternativno zrno ustvarjeno kot nadomestek privzete implementacije, alternativna implementacija samodejno deduje vse kvalifikatorje in imena EL privzete implementacije. Torej v našem primeru NacinPlacilaKarticaTest deduje kvalifikatorja @Default in @Kartica [3].

4.7 Viri komponentnega okolja Java EE

Specifikacija CDI uporablja izraz vir za katerokoli vrsto objektov v komponentnem okolju (component environment) Java EE. To so podatkovni viri JDBC, vrste, teme in tovarne povezav JMS, seje JavaMail in ostali transakcijski viri vključno s povezovalci JCA, upravljalci entitet in tovarne upravljalcev entitet JPA, oddaljeni EJB in spletne storitve. Med vstavljanjem virov komponentnega okolja in vstavljanjem odvisnosti v CDI, je najbolj opazna razlika, da se vstavljanje komponent okolja pri iskanju tipov z izpolnjujočimi pogoji, zanaša na imena, osnovana na nizih, pri čemer ni nobene prave konsistentnosti imen (ime JNDI, ime trajne enote, povezava EJB in podobno). Proizvajalni atributi so se izkazali kot odličen vmesnik za zmanjševanje omenjenih kompleksnosti v skupni model, saj so virom komponentega okolja omogočili sodelovanje v sitemu CDI, tako kot kateremukoli drugemu zrnu. Atributi imajo dvojen pomen, ker so lahko tarča vstavljanja komponentnega

okolja Java EE ali pa so deklarirani kot proizvajalni atribut CDI. Zaradi tega lahko definirajo preslikovanje iz imena, osnovanega na nizu komponentnega okolja, v kombinacijo tipa in kvalifikatorjev, uporabljenih v okolju močno tipiziranega vstavljanja [3].

4.7.1 Definiranje virov

Vire komponentnega okolja deklariramo z anotiranjem proizvajalnih atributov. To storimo z anotacijami vstavljanja komponentnega okolja, kot so: @Resource, @EJB, @PersistenceContext, @PersistenceUnit ali @WebServiceRef. Naslednja koda prikazuje definiranje vira trajnega konteksta [3, 4]:

@Produces @PersistenceContext(unitName="PodatkovnaBaza")
@Kvalifikator EntityManager em;

Atribut je lahko statičen in ne sme biti deklariran s ključno besedo final. Deklaracijo virov sestavljata dve informaciji [3]:

- metapodatki, potrebni za pridobitev reference na vir komponentnega okolja (ime JNDI, povezava EJB, ime trajne enote in podobno),
- tip in kvalifikatorji, ki bodo uporabljeni za vstavljanje reference v zrna.

Deklariranje virov je specifično glede na namestitev, zato je le-to smiselno v alternativnih razredih.

4.7.2 Močno tipizirano vstavljanje virov

Vire vstavljamo na povsem običajen način. Morda se zdi deklariranje dodatnih proizvajalnih atributov v primerjavi z vstavljanjem komponentnega okolja potratno, vendar se moramo zavedati, da bomo vire, kot je na primer EntityManager, uporabljali v mnogih zrnih. Torej veliko lažje in bolj močno tipizirano je za zgoraj deklarirani vir napisati [3]:

@Inject @Kvalifikator EntityManager em;

kot pa:

@PersistenceContext(unitName="PodatkovnaBaza") EntityManager em;

Poglavje 5

CDI in ekosistem Java EE

CDI je integriran v samo jedro platforme Java EE. Zasnovan je tako, da s pomočjo SPI zagotavlja integracijske točke v platformo Java EE in s tem posledično lahko deluje s tehnologijami izven platforme. Torej SPI postavlja CDI kot temelj za nov ekosistem prenosnih razširitev ter integracijo z obstoječimi okvirji in tehnologijami. Zaradi tega CDI olajša uporabo tehnologij, ki še niso del platforme Java EE [3].

5.1 Namestitev in pakiranje

Vsebnik mora ob zagonu aplikacije najprej izvesti iskanje zrn. To poteka tako, da vsebnik najprej določi obstoječe arhive zrn in zrna v posameznih arhivih. Nato sledi določanje omogočenih alternativ, prestreznikov in dekoratorjev ter razvrščanje omogočenih prestreznikov in dekoratorjev. Za tem vsebnik zazna morebitne definicijske napake in namestitvene težave. Nazadnje sproži dogodke, ki omogočajo prenosnim razširitvam integracijo z življenjskim ciklom namestitve [4].

Zrna lahko pakiramo v arhivske datoteke jar, ejb jar in war. Toda arhiv mora biti arhiv zrn. To pomeni, da mora vsak arhiv, ki vsebuje zrna, vsebovati datoteko z imenom beans.xml v mapi META-INF classpatha ali v mapi WEB-INF spletnega korena (za arhive war). Ta datoteka je lahko

prazna. Zrna, nameščena v arhivih brez datoteke beans.xml, ni možno uporabljati v aplikaciji. V vgrajenem (embeddable) vsebniku EJB so zrna lahko nameščena na katerikoli lokaciji, kjer je možno namestiti Javanska strežniška zrna. Vsaka lokacija mora vsebovati datoteko beans.xml [3].

5.2 Prenosne razširitve

CDI izpostavlja množico SPI-jev, namenjenih uporabi prenosnih razširitev za CDI. Načrtovalci CDI so predvideli razširitve, kot so integracija s pogoni za upravljanje poslovnih procesov, integracija z zunanjimi (third-party) ogrodji in nove tehnologije, zasnovane na CDI. Prenosna razširitev se lahko integrira z vsebnikom s ponujanjem svojih lasnih zrn, prestreznikov in dekoratorjev vsebniku, z vstavljanjem odvisnosti v svoje objekte s pomočjo storitve vstavljanja odvisnosti, s ponujanjem implementacije konteksta za doseg po meri ali s stopnjevanjem oziroma prepisovanjem metapodatkov osnovanih na anotacijah z metapodatki nekega drugega vira [3].

5.2.1 Kreiranje razširitve

Pri kreiranju prenosne razširitve je najprej potrebno ustvariti razred, ki implementira vmesnik Extension. Nato moramo registrirati razširitev kot ponudnika storitev. To storimo z ustvarjanjem datoteke META-INF/services/javax.enterprise.inject.spi.Extension, ki vsebuje ime razširitvenega razreda. Razširitev ni zrno, ker jo vsebnik instancira med procesom inicializacije, preden katerokoli zrno ali kontekst obstaja. Po končanem procesu inicializacije je lahko vstavljena v druga zrna. Razširitve imajo lahko opazovalne metode [3].

5.2.2 Dogodki življenjskega cikla vsebnika

Med procesom inicializacije vsebnik zažene množico dogodkov, kot so: BeforeBeanDiscovery, ProcessAnnotatedType, ProcessInjectionTarget, ProcessProducer, ProcessBean, ProcessObserverMethod, AfterBeanDiscovery in Af-

terDeploymentValidation. Razširitve lahko opazujejo omenjene dogodke:

```
class Razsiritev implements Extension {
    void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) {...}
    <T> void processAnnotatedType(@Observes
                                  ProcessAnnotatedType<T> pat) {
      if (pat.getAnnotatedType().isAnnotionPresent(Ign.class)) pat.veto();
    }
    ...
}
```

Opazovalna metoda lahko vstavi objekt BeanManager. Poleg opazovanja, je razširitvam dovoljeno tudi spreminjanje vsebnikovega metamodela. V zgornjem primeru kode v opazovalni metodi processAnnotatedType povemo vsebniku, naj ignorira tip, če je anotairan z @Ign [3].

5.2.3 Objekt BeanManager

Temelj razširjanja CDI je objekt BeanManager. Vmesnik BeanManager nam omogoča programsko pridobivanje zrn, prestreznikov, dekoratorjev, opazovalcev in kontekstov. Vsako zrno ali komponenta Java EE, ki podpira vstavljanje, lahko pridobi instanco BeanManager z vstavljanjem:

```
@Inject BeanManager beanManager;
```

Komponente Java EE lahko pridobijo instanco BeanManager s pomočjo JNDI (ime "java:comp/BeanManager"). Poglejmo si nekaj vmesnikov, ki jih izpostavlja BeanManager [3].

Vmesnik InjectionTarget

Vmesnik InjectionTarget izpostavlja operacije za proizvajanje in odstranjevanje instanc komponent, vstavljanje njenih odvisnosti in proženje povratnih metod življenjskega cikla [4]. Poenostavlja vstavljanje zrn CDI v objekte,

ki niso pod nadzorom CDI. To stori s pomočjo metode inject(T, CreationalContext<T>), kateri podamo instanco komponente zunanjega okolja in objekt CreationalContext, ki ga potrebuje vsaka instanca. Priporočljivo je, da ogrodja prepustijo delo instanciranja objektov ogrodno kontroliranih objektov CDI. Na ta način bodo objekti, nadzorovani s strani ogrodja lahko izkoristili prednost vstavljanja v konstruktor. Če ogrodje zahteva uporabo konstruktorja s posebnim podpisom, potem mora ogrodje samo instancirati objekt tako, da sta podprti le vstavljanje v metodo in atribut [3].

Vmesnik Bean

Instance vmesnika Bean predstavljajo zrna. Za vsako zrno v aplikaciji obstaja instanca Bean, ki je registrirana z objektom BeanManager. Obstajajo tudi objekti Bean, ki predstavljajo prestreznike, dekoratorje in proizvajalne metode. Vmesnik Bean izpostavlja vse stvari, ki smo jih obdelali v poglavju *3.2 Anatomija zrn* [3]. To stori z metodami getTypes(), getQualifiers(), getScope(), getName(), getStereotypes(), ki vračajo (kot že ime pove) tipe zrn, kvalifikatorje, tip dosega, ime EL in stereotipe zrna. Poleg tega vsebuje še metode getBeanClass(), isAlternative(), isNullable(), getInjectionPoints() [4]. Vmesnik Bean prenosnim razširitvam ponuja podporo za nove vrste zrn, še nedefinirane s specifikacijo CDI. Lahko bi ga na primer uporabili, da bi objektom, upravljanim v drugih ogrodjih, omogočili vstavljanje v zrna [3].

Vmesnik Context

Vmesnik Context podpira nove dosege za CDI in razširitve vgrajenih dosegov za nova okolja. Lahko na primer ustvarimo implementacijo vmesnika Context, s katero dodamo nek nov doseg v CDI, ali pa implementiramo Context, ki omogoči podporo dosegu pogovora v nekem zunanjem ogrodju [3].

Poglavje 6

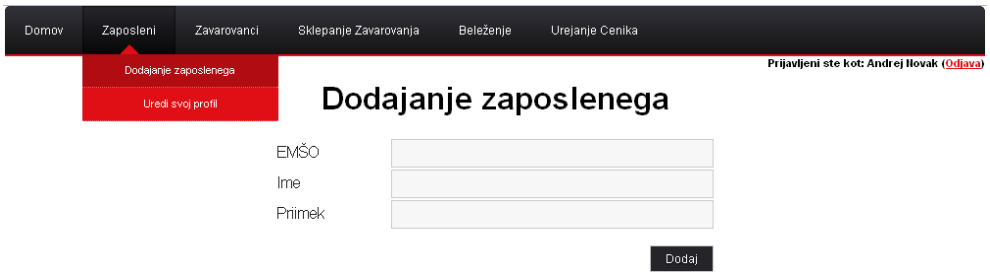
Praktična aplikacija

6.1 Razvojna orodja in tehnologije

Za razvojno okolje smo uporabili IBM Rational Application Developer for WebSphere Software verzije 8.0.3, ki uporablja strežnik WebSphere Application Server v8.0. Uporabniški vmesnik smo izdelali s tehnologijo JavaServer Faces 2.0, oblikovanje uporabniškega vmesnika spletne aplikacije pa smo izdelali s pomočjo knjižnice jQuery 1.7.1 programskega jezika JavaScript in s kaskadnimi stilnimi predlogami oziroma CSS. Podatkovno bazo smo zgradili s pomočjo orodja MySQL Workbench 5.2, s katerim smo tvorili skriptno datoteko SQL. Datoteko smo pognali s pomočjo orodja phpMyAdmin 3.3.2 in tako ustvaril podatkovno bazo. Poizvedbe smo opravljali s pomočjo JPA 2.0.

6.2 Implementacija

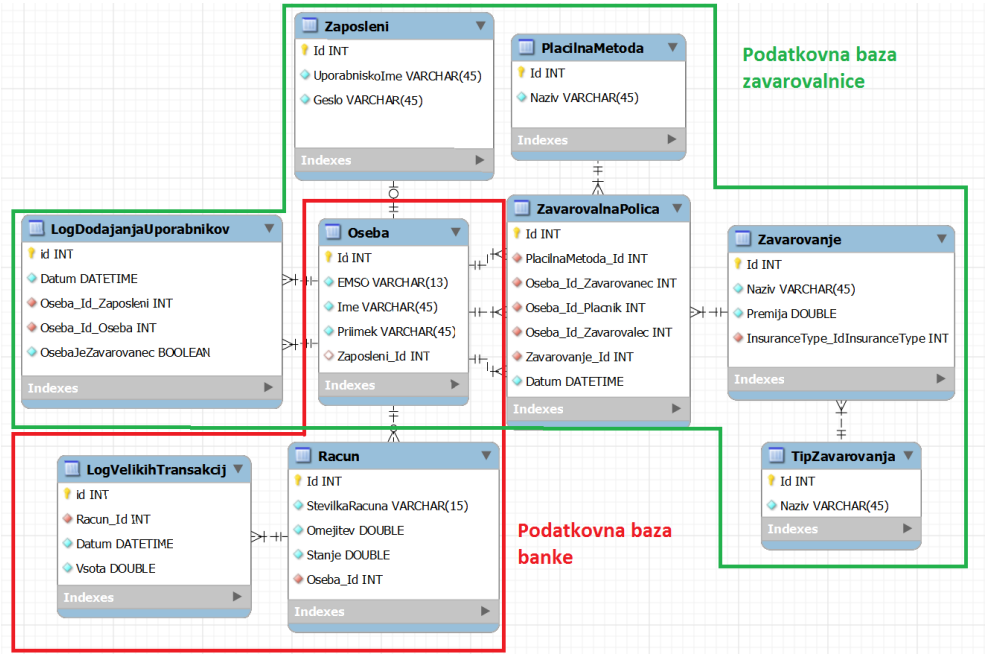
Za praktično implementacijo smo izdelali preprosto spletno aplikacijo, namenjeno zavarovalnicam. Ta omogoča dodajanje in urejanje zaposlenih ter zavarovancev, sklepanje zavarovanj, beleženje dodanih zaposlenih in zavarovancev ter urejanje cenika. Slika 6.1 prikazuje zaslonsko sliko obrazca dodajanja zaposlenega naše spletne aplikacije. Poleg tega aplikacija omogoča tudi opravljanje denarnih transakcij ob sklenitvi zavarovanja (TRR-ji so izmišljeni).



Slika 6.1: Zaslonska slika obrazca dodajanja zaposlenega spletne aplikacije

Za voljo primera sta podatkovni bazi zavarovalnice in banke (s pomočjo katere se opravlja transakcije) združeni, kot je prikazano na sliki 6.2. Zgradbo aplikacije lahko vidimo na sliki 6.3, ki prikazuje razredni diagram najpomembnejših zrn aplikacije. Pomembnejše komponente aplikacije prikazuje komponentni diagram na sliki 6.4.

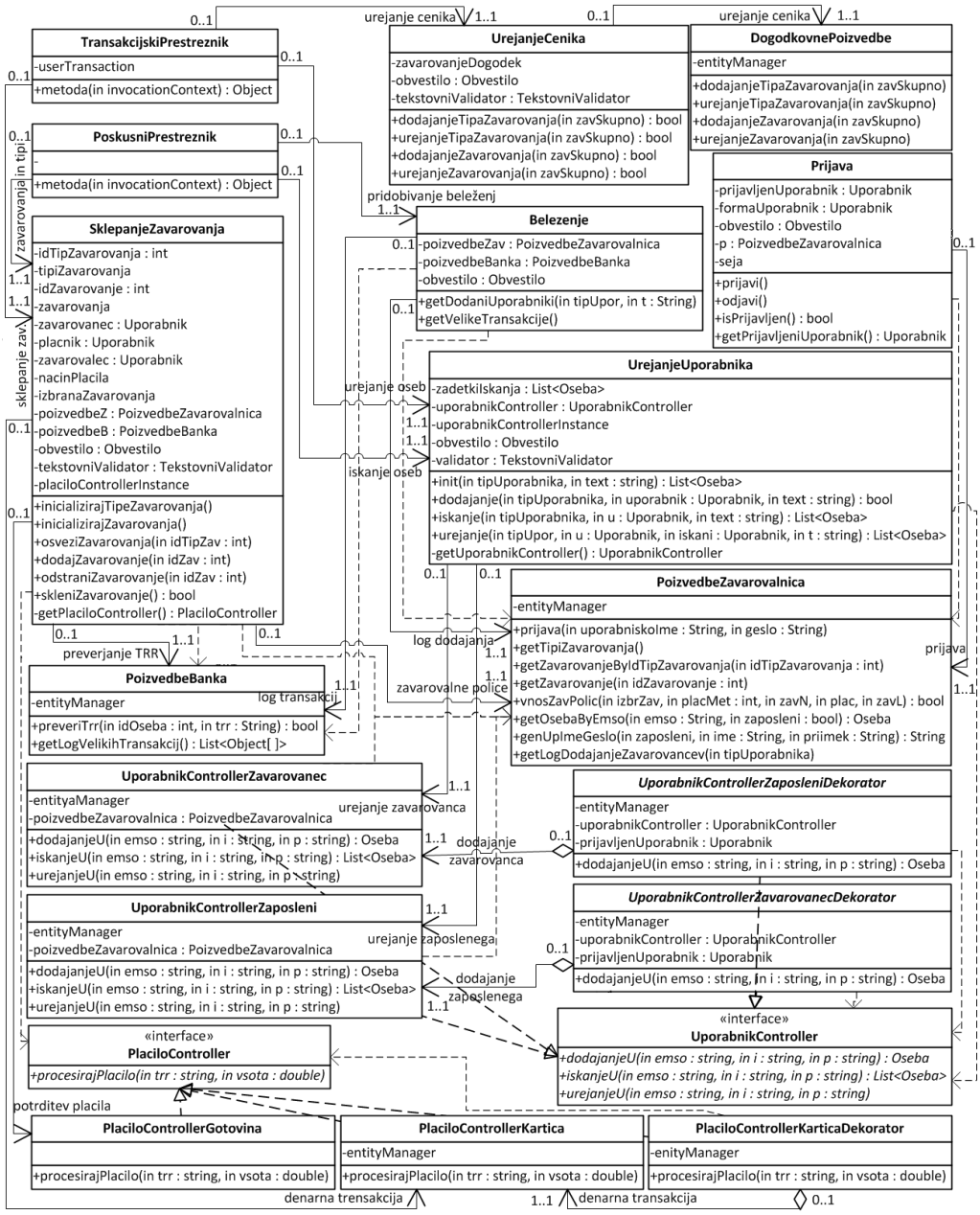
Aplikacija vsebuje več zrn s poizvedbami, namenjene zavarovalnici in banki. Zavarovalnica ima v ta namen zrna DogodkovnePoizvedbe (vsebuje opazovalne metode s poizvedbami za urejanje cenika, ki opazujejo dogodke tipa ZavarovanjeSkupno in imajo kvalifikator @Spremenljivo s članom TipSpremembe), PoizvedbeZavarovalnica (vsebuje splošno namenske poizvedbe, kot je na primer poizvedba za prijavo) ter implementaciji vmesnika UporabnikController. Ti sta UporabnikControllerZaposleni s kvalifikatorjem @Zaposleni in UporabnikControllerZavarovanec s kvalifikatorjem @Zavarovani. Vsebujeta poizvedbe namenjene urejanju uporabnika. Razlikujeta se v tem, da implementacija, namenjena zaposlenim, ob dodajanju zaposlenega letemu dodeli še uporabniško ime in geslo, medtem ko zavarovancu dodeli izmišljen transakcijski račun. Pri urejanju zaposlenega ustvari novo uporabniško ime in geslo glede na novo oziroma spremenjeno ime ali priimek. Poleg tega ima tudi banka več zrn namenjenih poizvedbam. Te so PoizvedbeBanka (vsebuje splošno namenske poizvedbe, kot je na primer preverjanje veljavnosti TRR) in implementaciji vmesnika PlaciloController. Ti sta PlaciloControllerGotovina s kvalifikatorjem @Gotovina in PlaciloControllerKartica s kvalifikatorjem @Kartica. Namenjeni sta obdelovanju plačil. Razlikujeta



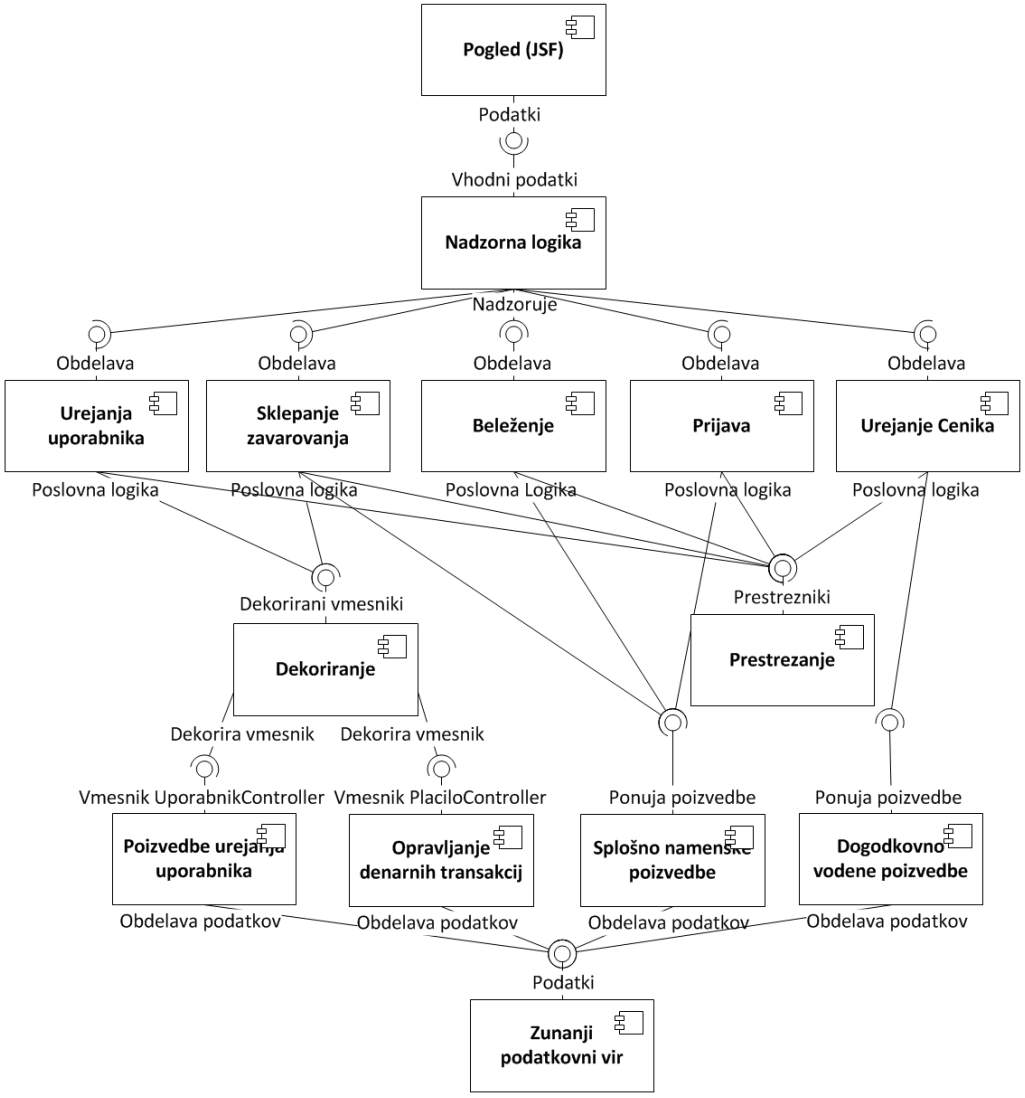
Slika 6.2: Podatkovna baza spletne aplikacije

se v tem, da implementacija, namenjena plačilu s kartico, opravi transakcijo, pri čemer implementacija, namenjena gotovini, samo potrdi plačilo oziroma vrne vrednost true. Aplikacija vsebuje tudi entitete JPA.

Implementirana so tudi zrna za hranjenje podatkov in pomožne logike. Zrno IzbranaZavarovanja hrani seznam izbranih zavarovanj ob sklepanju zavarovanja. Podatke o beleženju dodanih oseb hrani LogDodajanjaUporabnikovSkupno. Uporabimo ga za izpis beleženj v gradnik JSF dataTable. Natančneje, združuje atribute dveh entitet Oseba, in sicer prijavljenega uporabnika (zaposlenega, ki je dodal osebo) in dodano osebo (zavarovanca ali zaposlenega). Z istim razlogom LogVelikihTransakcijSkupno združuje atribute entitet Racun in LogVelikihTransakcij. Zrno ZavarovanjeSkupno združuje atribute entitet TipZavarovanja in Zavarovanje. Z njegovo pomočjo smo iz obrazcev za urejanje cenika prenesli podatke o novih vrednostih zavarovanja oziroma njegovem tipu. Zrno NacinPlacila hrani način plačila, ki ga uporabnik izbere na obrazcu ob sklepanju zavarovanja. Za obveščanje uporabnika



Slika 6.3: Razredni diagram najpomembnejših zrn aplikacije



Slika 6.4: Komponentni diagram

smo uporabili zrno Obvestilo. TekstovniValidator vsebuje pomožno logiko namenjeno validaciji vnosnih polj obrazcev. Iz obrazcev aplikacije posredujemo tudi podatke o osebah, ki zadevajo več entitet JPA. Te so Oseba, Zaposleni in Racun. Potrebne atribute le-teh združuje zrno Uporabnik.

Proizvajalce smo implementirali v razreda OstaliViri in ViriKomponentnegaOkolja. V prvem izmed naštetih smo z uporabo proizvajalnih metod izpostavili objekta HttpSession in HttpServletResponse. V aplikaciji imamo tudi več obrazcev, na katerih smo želeli ohraniti stanje vnosnih polj. Kot smo že omenili, je zrno Uporabnik zadolženo za posredovanje podatkov o osebah. V ta namen smo izpostavili več zrn Uporabnik, pri čemer je bilo vsako zadolženo za hranjenje podatkov različnega obrazca. Vsaki implementaciji proizvajalne metode smo dodelili različno ime EL, doseg seje in različno vrednost članka kvalifikatorja @Uporabniski. Uporabili smo tudi anotacijo @New, s pomočjo katere smo v omenjenih proizvajalnih metodah pridobili novo odvisno instanco zrna Uporabnik, ki jo je proizvajalna metoda vezala na doseg seje. S tem smo povišali doseg osnovne implementacije zrna Uporabnik, saj ima ta doseg zahtevka. V zrnu ViriKomponentnegaOkolja smo ustvarili proizvajalni atribut, ki izpostavi objekt UserTransaction, in proizvajalno metodo, ki izpostavi objekt EntityManager. Vira smo pridobili s pomočjo anotacij @PersistenceUnit in @Resource, ki temeljita na imenu, osnovanem na nizu. S pomočjo opisanih proizvajalcev, smo dosegli preslikovanje omenjenih imen v močno tipizirano vstavljanje odvisnosti. Za proizvedeno zrno EntityManager smo implementirali odstranjevalno metodo.

Aplikacije vsebuje tudi dva prestreznika, in sicer PoskusniPrestreznik (zadolžen za izvajanje bloka try-catch) ter TransakcijskiPrestreznik (zadolžen za opravljanje transakcije s pomočjo JTA). Prestreznika v primeru napake vrneta vrednost null. Na želeno zrno oziroma metodo smo ju vezali s pomočjo tipov prestrezniških vezav. Za vezavo prestreznika PoskusniPrestreznik smo uporabili tip prestrezniške vezave @Poskusni, za vezavo prestreznika TransakcijskiPrestreznik pa @Transakcijski.

Poslovna logika aplikacije se izvaja v več zrnih. Eno izmed takih je Pri-

java. Vanj smo vstavili odvisnost na zrna Uporabnik, Obvestilo, PoizvedbeZavarovalnica in HttpSession. Vsebuje metode prijavi, odjavi in isPrijavljen (vrne vrednost true, če je uporabnik prijavljen). Poleg tega vsebuje proizvajalno metodo getPrijavljeniUporabnik, ki izpostavlja zrno Uporabnik s kvalifikatorjem @Prijavljeni. Ta predstavlja trenutno prijavljenega uporabnika.

Naslednje zrno, ki vsebuje poslovno logiko, je UrejanjeUporabnika. Vanj smo vstavili odvisnost na zrna Obvestilo, TekstovniValidator in implementaciji vmesnika UporabnikController. Za vstavljanje omenjenih implementacij smo najprej na podlagi kvalifikatorjev @Zaposlen in @Zavarovan naredili podrazreda pomožnega razreda AnnotationLiteral, in sicer ZaposlenKvalifikator in ZavarovanKvalifikator. Želeno implementacijo vmesnika smo vstavili s pomočjo programskega poizvedovanja oziroma s pomočjo vmesnika Instance, kateremu smo podali kvalifikator z metodo select():

```
private @Inject @Any Instance<UporabnikController> uc;
private UporabnikController getUporabnikController(TipUporabnika tu) {
    if (tu == TipUporabnika.ZAPOSLENI) {
      return uc.select(new ZaposlenKvalifikator()).get();
    } else {
      return uc.select(new ZavarovanKvalifikator()).get();
    }
}
```

Tako smo lahko s pomočjo metode getUporabnikController pridobili želeno implementacijo vmesnika UporabnikController na povsem dinamičen način. Zrno vsebuje še metode init (inicializira seznam oseb za obrazec urejanja le-teh), dodajanje (doda osebo), iskanje (vrne zadetke iskanja oseb glede na EMŠO, ime in priimek) in urejanje (urejanje osebe). V metodah dodajanje in urejanje smo dosegli transakcijsko obnašanje tako, da smo ju s pomočjo tipa prestrezniške vezave @Transakcijski vezali na prestreznik TransakcijskiPrestreznik. Na podoben način smo dosegli tudi izvajanje bloka try-catch metode iskanje, in sicer s pomočjo tipa prestrezniške vezave @Poskusni.

Zrno SklepanjeZavarovanja je namenjeno izvajanju denarnih transakcij ter shranjevanju sklenjenih zavarovanj. Vanj smo vstavili odvisnosti na zrna NacinPlacila, več implementacij zrna Uporabnik (izpostavljene z zgoraj opisanimi proizvajalnimi metodami), IzbranaZavarovanja, PoizvedbeZavarovalnica, PoizvedbeBanka, Obvestilo, TekstovniValidator in implementaciji vmesnika PlaciloController. Omenjeni implementaciji smo pridobili na podoben način, kot smo pridobili želeno implementacijo vmesnika UporabnikController. Najprej smo na podlagi kvalifikatorjev @Gotovina in @Kartica naredili podrazreda pomožnega razreda AnnotationLiteral, in sicer GotovinaKvalifikator in KarticaKvalifikator. Nato smo s pomočjo programskega poizvedovanja vstavili želeno implementacijo vmesnika PlaciloController, zopet na močno tipiziran način. To smo storili z metodo getPlaciloController. Zrno poleg tega vsebuje še metode inicializirajTipeZavarovanja, inicializirajZavarovanja, osveziZavarovanja (glede na izbran tip zavarovanja na obrazcu s pomočjo zahteve AJAX posodobi seznam zavarovanj), dodajZavarovanje (izbrano zavarovanje na obrazcu doda v seznam izbranih zavarovanj) in odstraniZavarovanje (odstrani želeno zavarovanje). Omenjene metode so s pomočjo tipa prestrezniške vezave @Poskusni vezane na prestreznik PoskusniPrestreznik. Metoda skleniZavarovanje izvede denarno transakcijo in shrani sklenjena zavarovanja. Vezana je na prestreznik TransakcijskiPrestreznik.

Zrno Belezenje je namenjeno pridobivanju seznamov zrn, ki hranijo beleženja. Vanj smo vstavili odvisnost na zrna PoizvedbeZavarovalnica, PoizvedbeBanka in Obvestilo. Vsebuje metodi getDodaniUporabniki (vrne seznam zrn LogDodajanjaUporabnikovSkupno oziroma seznam beleženj dodanih oseb) in getVelikeTransakcije (vrne seznam zrn LogVelikihTransakcijSkupno oziroma seznam beleženj transakcij večjih od 10.000 €).

Zadnje zrno s poslovno logiko je UrejanjeCenika. To zrno je namenjeno dodajanju in urejanju zavarovanj ter njihovih tipov. Vanj smo vstavili odvisnosti na zrna Obvestilo, TekstovniValidator in vmesnik Event. Ta proži dogodke z opazovalci tipa ZavarovanjeSkupno s kvalifikatorjem @Spremenljivo. Opazujejo jih opazovalne metode zgoraj opisanega razreda DogodkovnePoi-

zvedbe. Za proženje dogodkov smo najprej na podlagi kvalifikatorja s članom @Spremenljivo naredili podrazred pomožnega razreda AnnotationLiteral, in sicer SpremenljivoKvalifikator. Nato smo instanci vmesnika Event predali ustrezno vrednost člana (TipSprememebe) omenjenega kvalifikatorja:

```
private @Inject @Any Event<ZavarovanjeSkupno> zavarovanjeDogodek;
zavarovanjeDogodek.select(new SpremenljivoKvalifikator() {
    public TipSpremembe value() {
      return TipSpremembe.DODAJANJE_TIPA_ZAVAROVANJA;
    }
} ).fire(zavarovanjeSkupno);
```

Na sproženi dogodek se odziva opazovalna metoda dodajanjeTipaZavarovanja zgoraj opisanega zrna DogodkovnePoizvedbe:

```
public void dodajanjeTipaZavarovanja(@Observes @Spremenljivo(TipSpreme
    mbe.DODAJANJE_TIPA_ZAVAROVANJA) ZavarovanjeSkupno zs) {...}
```

Zrno UrejanjeCenika vsebuje metode dodajanjeTipaZavarovanja, urejanjeTipaZavarovanja, dodajanjeZavarovanja in urejanjeZavarovanja (istoimenske metode kot zrno DogodkovnePoizvedbe). Naštete metode uporabljajo dogodke na podoben način, kot je prikazano v zgornji kodi. Vezane so na prestreznik TransakcijskiPrestreznik.

Vsa opisana zrna s poslovno logiko smo vezali na doseg seje. To pomeni, da so ustvarjena takrat, ko jih prvič potrebujemo in odstranjena na koncu seje. Pomembno je tudi dejstvo, da smo vsa zrna s poslovno logiko (razen Prijava) vstavili v istoimenska nadzorna zrna s pripono "Controller". To so BeleženjeController, SklepanjeZavarovanjaController, UrejanjeCenikaController, UrejanjeZaposlenegaController in UrejanjeZavarovancaController. To smo storili zaradi atributa action oznak commandButton in commandLink specifikacije JSF, s pomočjo katerega smo podali želeno metodo zrna, ki je bila s pomočjo omenjenih gradnikov nato sprožena. Težava je v tem, ker atribut action zahteva za argument metodo tipa void. Omenili smo tudi, da oba prestreznika v primeru napake vračata vrednost null. Zato smo najprej poklicali

metode nadzornih zrn tipa void, ki so nato poklicale istoimenske metode zrn s poslovno logiko. Tako smo lahko v primeru vrnjene vrednosti null oziroma v primeru napake prikazali ustrezno obvestilo. Z opisanimi zrni smo tudi izboljšali preglednost aplikacije.

V aplikaciji smo uporabili tudi dekoratorje. Dekoratorja UporabnikControllerZaposleniDekorator in UporabnikControllerZavarovanecDekorator smo uporabili za dekoriranje metode dodajanjeUporabnika zgoraj opisanega vmesnika UporabnikController. S pomočjo omenjenih dekoratorjev smo ob vsakem dodajanju osebe zabeležili prijavljenega uporabnika (zaposlenega), dodano osebo (zaposlenega ali zavarovanca) in datum.

```
@Decorator public abstract class ZapDek implements UporabnikController {
    @Inject @Delegate @Zaposlen UporabnikController uc;
    @Inject @Prijavljeni Uporabnik prijavljeniUporabnik;
    public Oseba dodajanjeUporabnika(String emso, String ime,
                                     String priimek) {
      Oseba oseba = uc.dodajanje(emso, ime, priimek);
      // ...beleženje prijavljenega uporabnika, dodane osebe in datuma...
```

V delegirani točki vstavljanja smo s pomočjo kvalifikatorja določili želeno implementacijo vmesnika UporabnikController. V zgornjem primeru smo uporabili kvalifikator @Zaposleni, zaradi česar je bila vstavljena implementacija UporabnikControllerZaposleni. Prijavljenega uporabnika smo pridobili s pomočjo vstavljanja zrna Uporabnik, ki ga izpostavlja zgoraj opisano zrno Prijava. Pri tem smo uporabili kvalifikator @Prijavljeni. Na podoben način smo realizirali tudi dekorator za beleženje denarnih transakcij večjih od 10.000 €. To smo dosegli z dekoratorjem PlaciloControllerKarticaDekorator, ki prestreza metodo procesirajPlacilo zgoraj opisanega vmesnika PlaciloController. V delegirani točki vstavljanja smo ga vezali samo na implementacijo PlaciloControllerKartica, kajti ob gotovinskem plačilu se ne izvede nobena transakcija.

Ugotovili smo, da je CDI pri razvoju spletne aplikacije prinesel mnoge prednosti. Kot prvo omenimo proizvajalce. Z njihovo pomočjo smo izpo-

stavili objekta HttpSession in HttpServletResponse, s čimer smo se izognili ponovnemu pridobivanju le-teh iz objekta FacesContext. Poleg tega smo z njihovo pomočjo izpostavili več različnih zrn tipa Uporabnik. Za razliko od prvotne implementacije zrna Uporabnik, smo jim dodelili drugačno ime EL, doseg in kvalifikator, s čimer smo se izognili implementaciji novih zrn z omenjenimi spremembami. Z njihovo pomočjo smo izvedli tudi preslikavo vstavljanja objektov UserTransaction in EntityManager, osnovanega na nizu, v močno tipizirano vstavljanje. S tem nam ob vstavljanju ni bilo več potrebno podajati anotacij @PersistenceUnit in @Resoruce ter pripadajočih imen. Z uporabo anotacij dosega smo nadzorovali življenjski cikel in vidnost instanc zrn. Vsebnik je ustvaril novo instanco zrna, ko smo jo prvič potrebovali in jo odstranil na koncu življenjskega cikla dosega. Poleg tega je delil instance med vsemi zrni, ki so se izvajala v istem kontekstu. Velika pridobitev specifikacije predstavljajo tudi kvalifikatorji. Z njihovo pomočjo smo vstavljali želeno implementacijo vmesnikov UporabnikController in PlaciloController. Zato nam v točki vstavljanja ni bilo potrebno podajati implementacije vmesnikov, ker bi s tem ustvarili močne odvisnosti med odjemalcem in implementacijo. Namesto tega smo podali vmesnik in kvalifikator @Any. Za dinamično pridobivanje želene implementacije smo nato uporabili programsko poizvedovanje, s čimer smo zmanjšali statičnost aplikacije. Kvalifikatorje smo uporabili tudi za ločevanje zrn tipa Uporabnik, ki smo jih izpostavili v zgoraj opisanih proizvajalnih metodah. Uporabili smo jih tudi za vezavo povzročiteljev in opazovalcev dogodkov. Šibko sklopljenost aplikacije smo povečali tudi s pomočjo tipov prestrezniških vezav, kajti PoskusniPrestreznik in TransakcijskiPrestreznik nista imela nobenih neposrednih odvisnosti z zrni, na katere sta bila vezana. S pomočjo dekoratorjev smo dodali dodatne funkcionalnosti metodi dodajanjeUporabnika vmesnika UporabnikController oziroma procesirajPlacilo vmesnika PlaciloController. To smo storili v času izvajanja aplikacije. S tem smo se izognili pisanju podrazredov implementacij omenjenih vmesnikov. Prednost pred podrazredi je bila tudi, da je sprememba vplivala le na dekorirane objekte. Prestreznikom in dekoratorjem smo s pomočjo da-

toteke beans.xml določili globalni vrstni red izvajanja. Prav tako smo jih z uporabo le-te enostavno omogočili. Šibko sklopljenost aplikacije so povečali tudi dogodki. Metode zrna UrejanjeCenika, ki so sprožale dogodke, niso imele nobenih neposrednih odvisnosti z opazovalnimi metodami zrna DogodkovnePoizvedbe. Vse naštete prednosti CDI pripomorejo k izboljšavi strukture kode ter s tem posledično naredijo razvoj bolj hiter in učinkovit.

Pri razvoju spletne aplikacije so se pojavljale tudi težave. Presenetljivo nam je največjo težavo predstavljal razmislek, kako bomo aplikacijo oblikovali oziroma zasnovali, da bomo čim bolje izkoristili prednosti CDI. Za omenjeno težavo smo porabili tudi največ časa. Težavo je predstavljala tudi tehnologija JSF, ki je do pričetka implementacije nismo poznali, tako da smo morali preučiti tudi njene osnove.

Aplikacijo bi bilo možno tudi izboljšati. Kar nekajkrat se v programski kodi ponovita anotacija @Named in anotacija dosega (bodisi @RequestScoped ali @SessionScoped), ki bi ju lahko agregirali v stereotip. Prav tako bi lahko agregirali ponavljajoči se anotaciji @SessionScoped in @Named("novo ime"), kateri se pojavljata v zgoraj opisanih proizvajalnih metodah, ki izpostavljajo zrno Uporabnik. Anotacijo @Named("novo ime") bi lahko zamenjali z anotacijo @Named, saj bi ta pridobila privzeto ime EL proizvajalne metode, s čimer bi anotacijo lahko dodali v stereotip. V zrnu UrejanjeCenikaController po vsakem uspešnem dodajanju in urejanju zavarovanj ter njihovih tipov, osvežimo seznama z omenjenimi elementi. To bi lahko storili tudi s pomočjo transakcijskega opazovalca. Dogodkovni parameter bi deklarirali s kvalifikatorjem @Observes(during=AFTER_SUCCESS), s čimer bi zagotovili, da se posodobitev zavarovanj in njihovih tipov izvede po uspešno opravljeni transakciji urejanja le-teh. Pri tem bi moral imeti dogodkovni parameter opazovalne metode tip ZavarovanjeSkupno, kot ostale opazovalne metode zgoraj opisanega zrna DogodkovnePoizvedbe, ki opravljajo urejanje zavarovanj in njihovih tipov. Razlika je v tem, da dogodkovni parameter ne bi smel imeti nobenega kvalifikatorja, s čimer bi dosegli, da bi bila opisana opazovalna metoda poklicana ob vsakem urejanju.

Poglavje 7

Sklepne ugotovitve

Pri implementiranju spletne aplikacije je CDI prinesel mnoge prednosti. Ob vstavljanju implementacije nekega vmesnika, nam v točki vstavljanja ni bilo potrebno podati razreda z implementiranim vmesnikom, ker bi s tem ustvarili močne odvisnosti med odjemalcem in implementacijo ter izničili pomen vmesnika. Namesto tega smo v točki vstavljanja podali vmesnik ter kvalifikatorje. Še več, ni nam bilo potrebno ustvariti kvalifikatorjev za vsako implementacijo, saj smo uporabili kvalifikatorje s člani. Podajali smo jih dinamično, s čimer smo zmanjšali statičnost programske kode.

S proizvajalnimi metodami in atributi smo dosegli preslikovanje vstavljanja virov komponentnega okolja Java EE, ki temelji na imenih, osnovanih na nizih, v močno tipizirano vstavljanje specifikacije CDI. Vire, ki zahtevajo eksplicitno odstranjevanje, smo odstranili s pomočjo odstranjevalnih metod – zopet na močno tipiziran način. Če smo želeli nekemu zrnu dodeliti drugačen doseg, ime EL ali drugačno inicializacijo, nam ni bilo potrebno ustvarjati novih razredov, saj smo razred z želenimi spremembami izpostavili s proizvajalno metodo. Uporabili smo tudi anotacijo @New in z njo v proizvajalno metodo vstavili novo odvisno instanco, posledično vezano na doseg le-te.

Življenjski cikel zrn in vidnost instanc smo nadzorovali s pomočjo dosega. Vsakemu zrnu smo določili želen doseg. Instanciranje in upravljanje življenjskega cikla vstavljenih zrn je ob vstavljanju prevzel vsebnik.

Za vezavo prestreznikov smo uporabili tipe prestreziških vezav. S tem smo povečali šibko soplpljenost aplikacije, ker prestreznik in prestreženi razred nista imela neposrednih odvisnosti. Izognili smo se uporabi anotacije @Interceptors, ki prestreznike posameznih zrn zakodira v programsko kodo. Z datoteko beans.xml smo določili globalni vrstni red prestreznikov. Ta nam je tudi poenostavila omogočanje oziroma onemogočanje prestreznikov.

S pomočjo dekoratorjev smo dinamično dodali vedenje želenemu objektu. V ta namen nam ni bilo potrebno ustvarjati podrazredov, ampak smo namesto tega s pomočjo kvalifikatorjev želen dekorator vezali na razred. S tem smo dodali dodatne funkcionalnosti v času izvajanja, namesto v času prevajanja, kot je značilno za podrazrede. Za razliko od podrazredov, je sprememba vplivala le na določene objekte. Z dekoratorjem smo lahko tudi poklicali katerokoli metodo vmesnika. Implementirani dekoratorji so bili abstraktni, zato jim ni bilo potrebno implementirati vseh metod dekoriranega vmesnika. Njihov globalni vrsti red izvajanja smo določili z datoteko beans.xml.

Šibko sklopljenost aplikacije smo povečali tudi s pomočjo dogodkov, ker so določeni deli aplikacije komunicirali brez katerekoli odvisnosti v času izvajanja. S pomočjo tipov zrn in kvalifikatorjev smo povezali opazovalce in povzročitelje dogodkov na močno tipiziran način.

V implementirani spletni aplikaciji bi lahko uporabili tudi druge funkcionalnosti CDI. V testnem okolju bi bilo mogoče izkoristili alternative in specializacijo. V bolj obsežni aplikaciji, bi se na določenih zrnih skoraj zagotovo ponavljali dosegi zrn, množice prestreziških vezav, privzeta imena EL ter alternative, ki bi jih lahko agregirali v stereotip. Za dogodke, sprožene v transakcijah, bi lahko uporabili transakcijske opazovalce. Če bi vstavljali odvisne objekte, ki bi morali vedeti nekaj o objektih ali o točkah vstavljanja, v katere bi bili vstavljeni, bi lahko uporabili objekt InjectionPoint. Prenosne razširitve bi uporabili za integracijo z zunanjimi ogrodji.

Skozi diplomsko delo se je izkazalo, da CDI izboljšuje strukturo programske kode in seveda s tem posledično naredi razvoj bolj učinkovit in hiter, aplikacijo oziroma programsko kodo pa bolj prilagodljivo na spremembe.

Literatura

[1] E. Gamma, R. Helm, R. Johnson in J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley, 1994, pogl. 3.5, 4.4, 4.7, 5.7

[2] M. B. Jurič, Postopki razvoja programske opreme: gradivo predmeta, Fakulteta za računalništvo in informatiko, 2011

[3] (2012) Weld - JSR-299 Reference Implementation. Dostopno na: http://docs.jboss.org/weld/reference/latest/en-US/html/index.html

[4] (2011) Contexts and Dependency Injection for the Java EE platform. Dostopno na: http://docs.jboss.org/cdi/spec/1.1.EDR1/pdf/cdi-spec.pdf

[5] (2012) Interceptor pattern. Dostopno na: http://en.wikipedia.org/wiki/Interceptor_pattern

[6] (2012) Decorator pattern. Dostopno na: http://en.wikipedia.org/wiki/Decorator_pattern

[7] (2012) Observer pattern. Dostopno na: http://en.wikipedia.org/wiki/Observer_pattern

[8] (2012) Java EE version history. Dostopno na: http://en.wikipedia.org/wiki/Java_EE_version_history