UNIVERZA V MARIBORU

Fakulteta za elektrotehniko, računalništvo in informatiko

DIPLOMSKO DELO

Maribor, junij 2003 Damijan Rebernak

Univerza v Mariboru

**Fakulteta za elektrotehniko,
računalništvo in informatiko**

Diplomsko delo univerzitetnega študijskega programa

Tehnologije za hiter razvoj strežniških javanskih zrn

Avtor: Damijan Rebernak, dipl. inž. rač. in inf.
Študijski program: Univerzitetni, Računalništvo in informatika
Smer: Programska oprema
Mentor: doc. dr. Janez Brest
Somentor: red. prof. dr. Viljem Žumer

Avtor: Damijan Rebernak, dipl. inž. rač. in inf.

Naslov: Tehnologije za hiter razvoj strežniških javanskih zrn

UDK: 004.41:004.738.5.05(043.2)

Ključne besede: strežniška javanska zrna, EJB vsebnik, java, odprti kod, poslovna programska oprema

Število izvodov: 4

Zahvala

Iskreno se zahvaljujem vsem, ki so mi kakorkoli pomagali.

UDK: 004.41:004.738.5.05(043.2)

Ključne besede:
strežniška javanska zrna, EJB vsebnik, java, odprti kod, poslovna programska oprema

Tehnologije za hiter razvoj strežniških javanskih zrn

Povzetek

Razvoj programske opreme je vedno veljal za zelo zahtevno opravilo. Izboljšave strojne in programske opreme ter vedno večja razširjenost računalniških sistemov je vzpodbudilo informatizacijo v mnogih podjetjih, kar je bistveno povečalo uporabniške zahteve. Vedno pomembnejša uporabniška zahteva je hiter razvoj programske opreme. To je glavni razlog, da se vedno več programerskih podjetij ukvarja z razvojem metod in orodij za pohitritev razvoja programske opreme.
V diplomskem delu je predstavljena arhitektura J2EE (*Java 2 Platform Enterprise Edition*), ki je namenjena razvoju vmesnega nivoja programske opreme. Posebno pozornost smo posvetili strežniškim javanskim zrnom (*Enterprise Java Beans*), ki je osrednja tehnologije arhitekture J2EE. Predstavljena so tudi odprto kodna orodja za hiter, inkrementalen razvoj aplikacij za to arhitekturo. Praktičen del diplomske naloge predstavlja spletni informacijski sistem za vodenje izposoje raznega gradiva kot so knjige, zgoščenke, inventar itd. V praktičnem delu diplomske naloge smo predstavili pristope pri implementaciji J2EE aplikacije, pri implementaciji pa smo uporabili orodje XDoclet.

UDK: 004.41:004.738.5.05(043.2)

Keywords:
Enterprise Java Beans, EJB Container, Java, Open-Source, Enterprise Application

Technologies for Rapid Development of Enterprise Java Beans

Abstract

Software development has always been considered a very coplex task. Rapid hardware and software development in production processes has led to complete informatization of many companies. One of the most important user demands in modern informatization is rapid development of software products. This serves as a main reason why so many software companies are occupied with the development of efficient methods and tools to speed up and simplify the development process.

In this work we present the J2EE (*Java 2 Platform Enterprise Edition*) architecture which is intended for the development of business tier in information systems. In our discussion we focused specifically on Enterprise Java Beans (EJB), which can be viewed as a principal technology of J2EE. In addition, we present some open source-based tools for rapid and incremental development of J2EE applications. Practical part of this work is realized in a web-based information system for managing a simple virtual library. This is also the part where we present a concrete approach in the implementational phase of a J2EE application with the use of XDoclet tool.

Kazalo

Slike

Primeri

Tabele

Education is that which remains, if one has forgotten everything they learned in school.

– Albert Einstein –

1 Uvod

Zahteve uporabnikov po kvalitetni programski opremi se venomer povečujejo. Programska oprema, in s tem informacijski sistemi, mora biti čimbolj prilagojena režimu dela uporabnika. Na ta način vidno prispeva k dvigu kvalitete opravljenega dela, hkrati pa delo tudi olajša. Prav tako se od programske opreme pričakuje vedno večja podpora pri sprejemanju poslovnih odločitev. Kljub vse bolj obsežnim in zahtevnim uporabniškim zahtevam, se pričakovani čas od začetka razvoja do končnega produkta manjša. Potreba po vse hitrejšem razvoju poslovne programske opreme izvira iz dinamičnosti dela, kjer se aplikacije uporabljajo. Posledično se veča tudi kompleksnost in cena

razvoja programske opreme. Vedno dražji razvoj programske opreme nas vzpodbuja k temeljitemu razmisleku o metodah načrtovanja in implementacije programske opreme, prav tako je ključnega pomena tudi izbira tehnologij za načrtovanje in implementacijo. Težnja obeh vpletenih strani, razvijalcev in naročnikov programske opreme, je razvoj fleksibilne, zmogljive in cenejše programske opreme, saj si tako obe strani zmanjšata stroške. Na trgu se pojavlja vedno več tehnologij in orodij za hiter razvoj programske opreme. Sorazmerno večanju kompleksnosti uporabniških zahtev in programske opreme se veča tudi kompleksnost tehnologij in orodij za razvoj programske opreme, zato je izbira le-teh ključnega pomena za končni uspeh razvite programske opreme.

V diplomskem delu smo preučili in preiskusili metode, tehnologije in orodja za načrtovanje in razvoj programske opreme. Uporabniške zahteve po hitrem razvoju in prilaganju programske opreme njihovim potrebam se večajo. Izbira tehnologije za implementacijo programske opreme je lahko ključnega pomena, saj lahko razvijalcem precej olajša oz. zagreni življenje v času razvoja programske opreme. Da bi poenostavili in pohitrili razvoj poslovnih aplikacij, smo se odločili, da preučimo in preizkusimo *Java 2 Platform Enterprise Edition* (J2EE), ki postaja čedalje pogostejša arhitektura za razvoj poslovnih aplikacij v programskem jeziku java. J2EE predstavlja vmesni nivo razvite aplikacije. Aplikacijam zagotavlja razne storitve, kot so transakcijska podpora, zaščita, obstojnost podatkov, vodenje dnevnikov itd. Omenjene storitve so za razvijalca povsem transparentne in jih ni potrebno posebej vključevati v poslovno logiko posameznih komponent aplikacije. Preučili smo aplikacijske strežnike za podporo arhitekture J2EE ter tehnologije, ki spadajo v ta okvir. Posebej smo si ogledali tehnologije: javanska strežniška zrna (EJB – *Enterprise Java Beans*), CMP – *Container Managed Persistence*, CMR – *Container Managed Relationships* ter tehnologije za razvoj spletnih aplikacij Java Servlets in JavaServer Pages (JSP). Razvoj aplikacij za arhitekturo J2EE je precej zahteven in obsežen, zato smo se pri implementaciji odločili uporabiti orodja, ki na podlagi podanih specifikacij generirajo strežniška javanska zrna. Na ta način se razvoj precej pohitri, prihranimo pa si tudi marsikatero napako, ki bi bila

posledica površnosti razvijalca. Čeprav na tržišču obstaja kar precej orodij za generiranje strežniških javanskih zrn, smo se odločili za uporabo orodij razvitih in dostopnih po principu odprtega koda (*Open Source*).

Predstavljene pristope pri načrtovanju programske opreme in orodja za generiranje strežniških javanskih zrn, opisane v diplomskem delu, smo uporabili pri razvoju spletnega informacijskega sistema za vodenje izposoje raznega gradiva kot so zgoščenke, knjige, inventar itd. Razvit spletni informacijsi sistem služi tudi kot predstavitev možnosti, ki jih razvijalcem nudijo opisana orodja ter arhitektura J2EE. Pridobjeno znanje in izkušnje nameravamo uporabiti pri načrtovanju in implementaciji spletnega informacijskega sistema za projektno vodenje proizvodnje v strojegradnji, ki ga za podjetje Lestro Ledinek d.o.o. razvijamo v Laboratoriju za računalniške arhitekture in jezike.

Diplomsko delo je razdeljeno na pet poglavij. V poglavju 2. opišemo arhitekturo J2EE ter nekaj najzanimivejših tehnologij v okviru te platforme. Posamezni pristopi pri razvoju aplikacij za arhitekturo J2EE so prikazani tudi na primerih. V poglavju 3. opišemo problematiko razvoja večjih programskih sistemov za arhitekturo J2EE. Sledi pregled orodij za generiranje strežniških javanskih zrn. V predzadnjem poglavju (4.) opišemo pristope, ki smo jih uporabili pri razvoju spletnega informacijskega sistema *Spletna izposojevalnica*. Diplomsko delo zaključuje 5. poglavje, kjer povzamemo znanje, ki smo ga zajeli v celotni diplomski nalogi.

Every time I learn something new it pushes some old stuff out of my brain!

– Homer Simpson –

2

J2EE - *Java 2 Platform Enterprise Edition*

V poglavju je opisana arhitektura J2EE in pripadajoče tehnologije. V uvodu se bralec seznani z arhitekturo ter spozna tehnologije, ki jo sestavljajo. Nadaljevanje poglavja je namenjeno opisu aplikacijskega strežnika, ki je jedro arhitekture, bralec pa se seznani tudi z najzanimivejšo tehnologijo arhitekture (javanska strežniška zrna – EJB). Poglavje zaključuje praktičen primer, ki podaja celoten postopek razvoja J2EE aplikacije.

2.1 Uvod

Metodologija razvoja poslovnih aplikacij se spreminja. Opažamo prehod izvajanja vedno več poslovnih aplikacij iz klasičnega namiznega okolja (osebni računalniki) na stran strežnikov. Večina poslovne logike se izvaja na strežniku, aplikacije na odjemalčevi strani pa so ponavadi zadolžene le za ustrezen prikaz in vnos podaktov. Na ta način se pohitri razvoj in nameščanje programske opreme saj se večina sprememb pri nadgradnji programske opreme opravi le na omejenem številu računalnikov (strežniki), poenostavi se tudi zagotavljanje varnosti uporabnikovih podatkov. Omenjene lastnosti poslovne programske opreme, predvem hitrost odziva poslovnih aplikacij na spremembe na trgu, so ključnega pomena za uporabnika, saj poslovne aplikacije težko sledijo hitrim spremembam v poslovnem svetu, primerna programska oprema pa je lahko pomembna prednost pred vse večjo konkurenco. Arhitektura J2EE (*Java 2 Enterprise Edition*) [**?**, **?**, **?**] je bila razvita z namenom, da razvijalcem olajša razvoj kompleksne poslovne programske opreme in s tem zmanjša čas ter stroške razvoja. Razvijalcem omogoča komponentno orientirano načrtovanje, razvoj in namestitev (*deployment*) programske opreme, porazdeljen model razvitih aplikacij, ponovno uporabnost že razvitih komponent, enoten model zaščite, vgrajene mehanizme za izmenjavo podatkov, ki temeljijo na XML-u, transakcijsko podporo komponentam aplikacij itd.

2.2 Kaj je J2EE?

Arhitektura J2EE *Java 2 Platform Enterprise Edition* je z leti postala standardno razvojno okolje za razvoj porazdeljenih večnivojskih poslovnih aplikacij v programskem jeziku java. Aplikacije razvite s to arhitekturo so sestavljene iz posameznih komponent, le-te pa so lahko po posameznih nivojih nameščene tudi na različnih računalnikih. Komponenta J2EE je samostojna enota, ki opravlja določene operacije in skupaj z ostalimi komponentami tvori celotno aplikacijo. Te komponente so razvite v programskem jeziku java, njihov razvoj pa je povsem identičen razvoju ostalih komponent oz. programov

v programskem jeziku java. Razlika, glede na *"navadne"* javanske komponente (*Java Beans*) oz. razrede, je le v tem, da so J2EE komponente razvite v sladu z javno dostopnimi specifikacijami [**?**] ter nameščene v aplikacijski strežnik, ki upravlja in nadzoruje njihovo izvajanje. Prednost J2EE komponent je v zasnovi, ki omogoča visok nivo ponovne uporabnosti razvitih komponent.

V času razvoja je mogoče na več načinov zasnovati arhitekturni model razvijajoče aplikacije. Tako poznamo t.i. enonivojske, dvonivojske, trinivojske in večnivojske arhitekturne modele aplikacij. Enonivojska arhitektura je najbolj enostavna, predstavlja jo aplikacija, ki se izvaja zgolj na enem računalniku. Pri dvonivojskih in večnivojskih aplikacijah pa že govorimo o porazdeljenih aplikacijah, saj aplikacije uporabljajo vire oz. izvajajo rutine, ki se ne nahajajo na zgolj enem računalniku. Dvonivojsko arhitekturo imenujemo tudi odjemalec/strežnik (*client/server*) arhitektura. Strežnik skrbi ponavadi za izvajanje poslovne logike in hrambo podatkov, medtem ko odjemalec skrbi za vnos in ustrezen prikaz podatkov. Pri trinivojski arhitekturi se aplikacija deli na nivo predstavitve podatkov, vmesni nivo ali nivo poslovne logike in podatkovni nivo. Posamezni nivoji so v večini primerov predstavljeni na različnih računalnikih. Arhitekturni modeli aplikacij so natančneje opisani v [**?**].
Arhitektura J2EE predvideva štirinivojski arhitekturni model razvitih aplikacij, kjer so komponente razdeljene na naslednje nivoje:

- nivo odjemalca (*Client-tier*) – izvajnje na strani odjemalca,

  V arhitekturi J2EE poznamo dve vrsti odjemalcev, in sicer spletni odjemalci in *"navadne"* J2EE aplikacije. Spletne odjemalce predstavljajo različni označevalni (*markup*) jeziki kot so HTML, XML, WML itd. Te dokumente generirajo komponente spletnega nivoja. Drugi del spletnega odjemalca predstavljajo spletni brskalniki, katerih glavna naloge je vnos in ustrezen prikaz podatkov. Spletne odjemalce imenujemo tudi šibki odjemalec (*thin client*). Šibki odjemalci ponavadi ne dostopajo do podatkovnega nivoja in ne vsebujejo nikakršne poslovne logike.

Odjemalci J2EE aplikacije se za razliko od spletnega odjemalca izvajajo na odjemalčevi strani. Te aplikacije ne vsebujejo poslovne logike, do poslovnega nivoja dostopajo z uporabo strežniških javanskih zrn. Poseben primer spletnega odjemalca je *Applet*, javanski program, ki dostopa do poslovnega nivoja ter ga izvaja spletni brskalnik.

- spletni nivo (*Web-tier*) – izvajanje na strani strežnika,

  Spletne komponente J2EE arhitekture so Java Servlet-i in JavaServer Pages (JSP) [?, ?, ?]. Servleti imajo natančno predpisan vmesnik. Namenjeni so strežni uporabnikovih zahtev (ponavadi te zahteve posreduje spletni brskalnik). Glede na zahtevo dinamično generirajo izhodni dokument (HTML, XML, WML itd.). JSP strani so s posebnimi oznakami dopolnjene spletne strani za predstavitev na spletu. Omogočajo integracijo javanskega izvornega koda v dokumente namenjene objavi v spletu in jim na ta način omogočijo prikaz dinamičnih vsebin. Vsaka JSP stran se pred izvajanjem prevede v servlet ter se na ta način tudi izvede. Razlika med servleti in JSP stranmi je torej le v načinu razvoja. Za obdelavo uporabniških zahtev in posredovanje le-teh strežniškim javanskim zrnom lahko spletne komponente vsebujejo tudi javanska zrna (*Java Beans*).

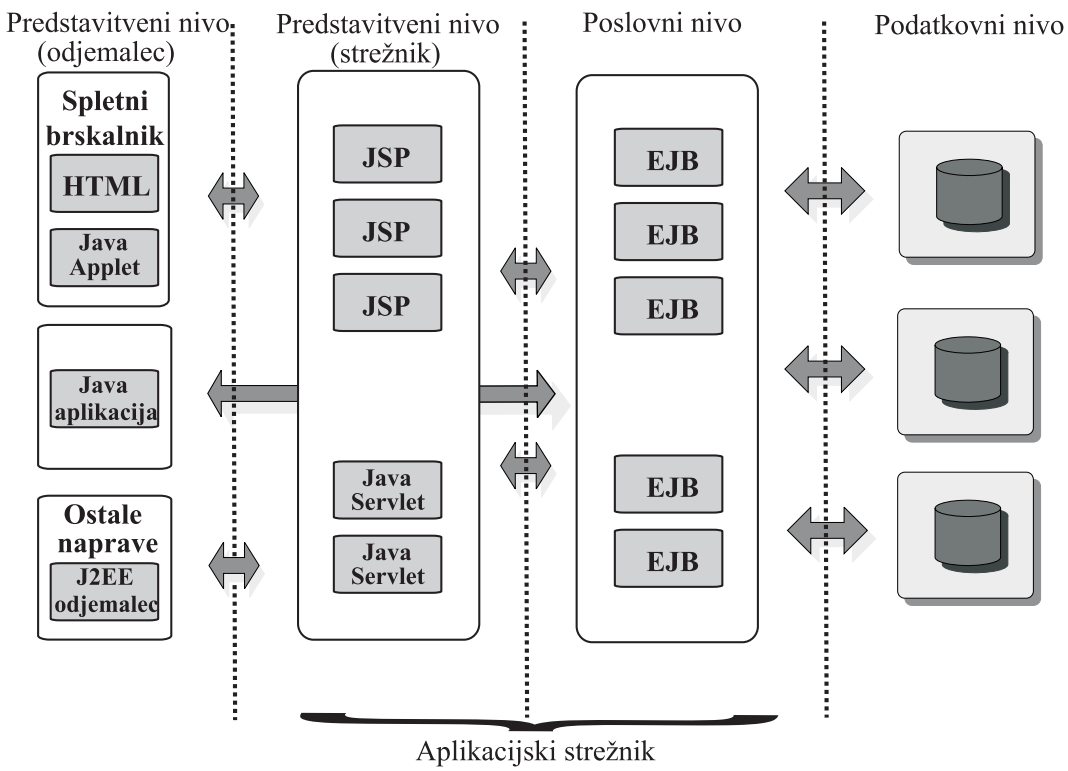
- poslovni nivo (*Business-tier*) – izvajanje na strani strežnika in

  Poslovna logika je implementirana s strežniškimi javanskimi zrni (*Enterprise Java Beans*), ki jih izvaja aplikacijski strežnik. Strežniška javanska zrna komunicirajo z vsemi ostalimi nivoji J2EE aplikacije, npr. podatke, ki jih uporabnik vnese v obrazec na spletu, je potrebno obdelati ter shraniti v podatkovno bazo. Več o strežniških javanskih zrnih bomo govorili v poglavju 2.4.

- nivo shranjevanja podatkov ali podatkovni nivo – samostojen podatkovni strežnik.

  Podatkovni nivo predstavlja ponavadi sistem za upravljanje s podatkovno bazo (SUPB – *DBMS*). Lahko se izvaja na ločenem strežniku, ali pa je del aplikacijskega strežnika.

Posamezne nivoje J2EE aplikacije prikazuje slika 2.1.



Slika 2.1: Nivoji J2EE aplikacije.

2.2.1 Standardne tehnologije J2EE

Tehnologije J2EE temeljijo na standardnem javanskem razvojnem okolju (*Java 2 Platform Standard Edition*), tako da so nekatere J2EE tehnologije tudi del standardnega javanskega razvojnega okolja. Razvijalci programske opreme imajo pri razvoju na voljo celoten aplikacijski vmesnik tehnologij standardenga razvojnega okolja ter tehnologije in aplikacijske vmesnike namenjene le za razvoj aplikacij za arhitekturo J2EE:

- Enterprise Java Beans (EJB),

  Specifikacije strežniških javanskih zrn [?] natančno določajo način razvoja le-teh ter način komunikacije EJB vsebnika in komponent. Strežniška

javanska zrna so osrednja tehnologija J2EE arhitekture, zato sledi v poglavju 2.4 natančnejši opis te tehnologije.

- Java Remote Method Invocation (RMI) in RMI-IIOP,

  RMI je prvoten način komunikacije med distribuiranimi objekti v programskem jeziku java in omogoča klice metod oddaljenega objekta. RMI-IIOP je razširitev RMI in omogoča integracijo arhitekture CORBA (*Common Object Request Broker Architecture*). J2EE aplikacije zahtevajo uporabo RMI-IIOP za dostop do komponent oz. za klic oddaljenih metod. To zagotavlja komponentam neodvisnost od protokola za komunikacijo med strežnikom in odjemalcem.

- Java Naming and Directory Interface (JNDI),

  Aplikacijski vmesnik JNDI uporabljajo komponente in aplikacijski strežnik za imenovanje in dostop do oddaljenih virov. Oddaljen vir je lahko komponenta, podatkovna baza, podatki itd.

- Java Database Connectivity (JDBC),

  JDBC je aplikacijski vmesnik za dostop do relacijskih podatkovnih virov. Omogoča izvajanje SQL stavkov v javanskih komponentah. Aplikacijski vmesnik je sestavljen iz dveh nivojev. Prvi nivo predstavlja aplikacijski vmesnik, tega uporabljajo komponente za dostop do podatkovnih virov. Drugi nivo predstavljajo gonilniki, ki jih ponavadi razvije dobavitelj oz. proizvajalec podatkovnega vira.

- Java Transaction API (JTA) in Java Transaction Service (JTS),

  JTA je aplikacijski vmesnik med med upravljalcem transakcij ter ostalimi viri, ki so zajeti v porazdeljen transakcijski sistem: upravljalec virov, aplikacijski strežnik in transakcijska komponenta oz. aplikacija. JTS pa je implementacija upravljalca transakcij, ki podpira JTA.

- Java Messaging Service (JMS),

  Aplikacijski vmesnik JMS omogoča komponentam sihnono in asinhromo pošiljanje in sprejemanje sporočil. Komunikacija poteko preko strežnika, tako ni potrebe po dostopnosti obeh komponent v času pošiljanja/sprejemanja sporočil, kar je bistvena prednost pred sorodnimi mehanizmi za komunikacijo (RMI, RPC, CORBA itd.).

- Java Servlets,

  Java Servlet je komponentno usmerjena in arhitekturno neodvisna tehnologija za razvoj spletnih aplikacij. Servlet je zahteva/odgovor (*request/response*) usmerjena spletna komponenta, ki glede na zahtevo s strani odjemalca (spletni brskalnik) dinamično generira spletno vsebino. Servlet se izvaja na strežniku, izvajanje servleta pa upravlja servlet vsebnik.

- JavaServer Pages (JSP),

  JSP strani so s posebnimi oznakami dopolnjene statične strani za predstavitev vsebine na svetovnem spletu. JSP strani se ob prvi odjemalčevi zahtevi prevedejo v servlet in se na enak način tudi izvedejo.

- Java IDL,

  Java IDL je javanska implementacije CORBA-e. Omogoča integracijo javanskih komponent s komponentami razvitimi v drugih programskih jezikih. Dostop do eksternih CORBA objektov je omogočem po protokolu IIOP.

- Java Mail,

  V vedno več komponentah prihaja do zahtev po pošiljanju elektronske pošte. Aplikacijski vmesnik Java Mail omogoča pošiljanje elektronske pošte neodvisno od arhitekture oz. protokola.

- J2EE Connector Architecture (JCA),

  JCA omogoča dotop komponentam J2EE arhitekture do že obstoječih informacijskih sistemov. Razviti je potrebno gonilnik za dostop do obstoječega informacijskega sistema, JCA pa poskrbi za upravljanje poslovnega nivoja (transakcije, zaščita podatkov, ...).

- Java API for XML Parsing (JAXP) in

  JAXP je implementacija industrijskega standardnega aplikacijskega vmesnika SAX in DOM za razpoznavanje in transformacijo XML dokumentov.

- Java Authentication and Authorization Service (JAAS).

  JAAS je javanska implementacija standardnega avtorizacijskega modula PAM (*Pluggable Authentication Module*). Predpisuje način avtoriziranja in overjanja uporabnikov za dostop do posamezne komponente oz. izvajanje določene metode.

2.3 Aplikacijski strežnik

Aplikacijski strežnik je vmesnik med komponentami poslovnega nivoja in nizkonivojskimi arhitekturno odvisnimi rutinami, ki omogočajo izvajanje operacij poslovnega nivoja na določeni arhitekturi (platformi). Komponentam zagotavlja izvajalno okolje, zato je le-te pred izvajanjem potrebno namestiti (*deploy*) v aplikacijski strežnik, ki komponentam zagotavlja tudi razne storitve, za katere bi v klasični aplikaciji moral poskrbeti razvijalec programske opreme sam. Namestitveni proces vključuje tudi opis storitev, ki jih komponenta zahteva od aplikacijskega strežnika. Vsak J2EE aplikacijski strežnik mora natančno ustrezati specifikacijam [**?**], ki jih upravlja in objavlja podjetje SUN Microsystems. To dejstvo omogoča komponentam poslovnega nivoja popolno neodvisnost od uporabljenega aplikacijskega strežnika (*WORDA – Write Once, Run and Deploy Anywhere*). Seveda se lahko aplikacijski strežniki med sabo razlikujejo po načinu implementacije, programskem jeziku v katerem so bili razviti, in deloma tudi v uporabljenih tehnologijah za zagotavljanje storitev

komponentam poslovnega nivoja. Postopek namestitve komponent se zato glede na proizvajalca aplikacijskih strežnikov razlikuje. Ker smo praktičen del diplomskega dela implementirali z aplikacijskim strežnikom JBoss [?] in spletnim strežnikom ter servlet vsebnikom Tomcat [?], je večina prikazanih primerov uporabe arhitekture J2EE razvita za ta dva produkta.

Storitve, ki jih mora po specifikacijah zagotavljati vsak J2EE aplikacijski strežnik, so naslednje:

- podpora transakcijam,

  V času namešanja komponente v aplikacijski strežnik določimo metode oz. relacije med metodami, ki se naj izvedejo v sklopu ene transakcije.

- zagotavljanje varnosti,

  Komponentam oz. metodam komponent lahko določimo pravice dostopa, tako lahko do posameznih virov dostopajo le avtorizirani uporabniki.

- JNDI storitve in

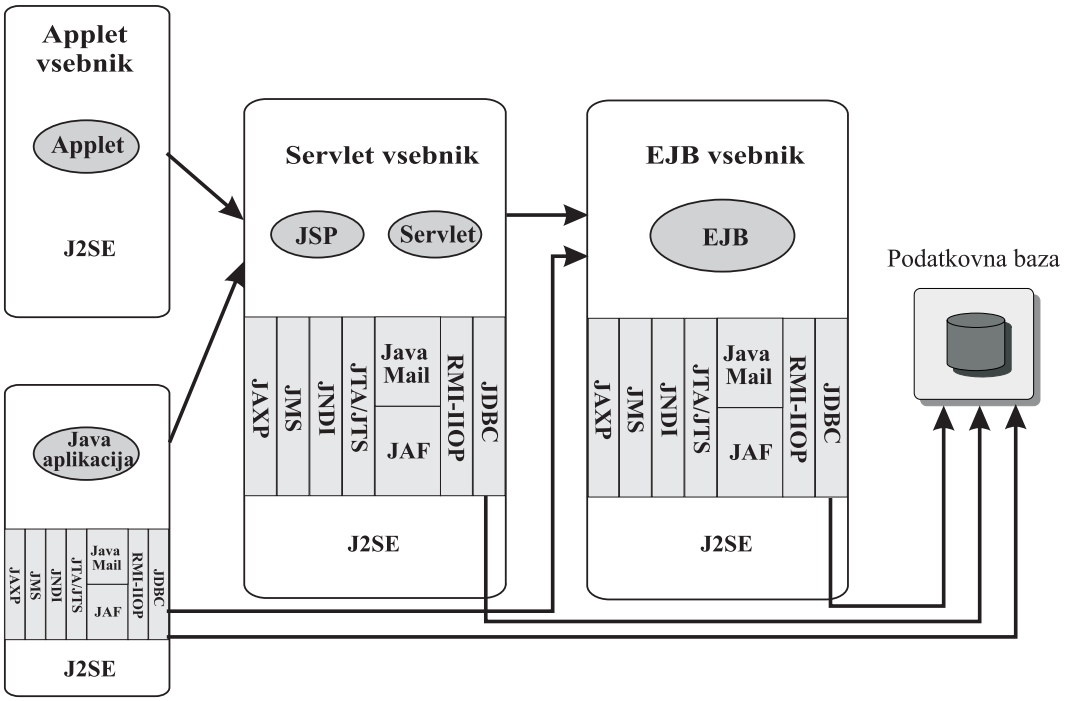
  Vsak vir v J2EE aplikaciji ima določeno unikatno ime, s katerim lahko dostopamo do posamezne komponente oz. vira. Ta pristop omogoča lokacijsko transparentnost razvitih aplikacij.

- upravljanje oddaljenih klicev.

  Nizkonivojske rutine za upravljanje oddaljanih klicev metod in rutine za dostop do ostalih virov so prepuščene aplikacijskemu strežniku. Klici metod J2EE komponent so tako identični klicem metod lokalnih komponent.

Komponentam šele v času nameščanja v J2EE aplikacijski strežnik in ne v času razvoja specificiramo storitve, ki jih mora komponenti zagotaviti aplikacijski strežnik, in način njihove uporabe. Ta pristop ima kar precej prednosti. Najbolj očitni sta olajšanje dela razvijalcem, saj se lahko osredotočijo le na

poslovno logiko ter ponovna uporabnost že razvitih komponent. Ista komponenta se lahko z različno konfiguracijo obnaša precej drugače in jo lahko tako uporabimo na različnih mestih ali celo v različnih aplikacijah.
Aplikacijski strežnik zagotavlja komponentam določene storitve, ki jih ni mogoče konfigurirati na nivoju komponent. Te storitve so naslednje: upravljanje življenjskega cikla strežniških in spletnih komponent, persistenca (obstojnost) podatkov strežniških javanskih zrn, upravljanje povezav do različnih podatkovnih virov in dostop do aplikacijskega vmesnika standardnega javanskega razvojnega okolja (J2SE) in ostalih tehnologij arhitekture J2EE (poglavje 2.2.1.). Celotno arhitekturo J2EE in pripadajoče tehnologije prikazuje slika 2.2.



Slika 2.2: Arhitektura J2EE.

Industrijska podpora arhitekturi J2EE narašča iz dneva v dan. Številni vodilni proizvajalci informacijske tehnologije vključujejo v svoje produkte tudi aplikacijske strežnike za arhitekturo J2EE oz. v svoje že obstoječe aplikacijske

strežnike vgrajujejo EJB vsebnik. Za primer naštejmo le nekaj teh aplikacijskih strežnikov: BEA Systems WebLogic, IBM Websphere, Orion, JBoss, JRun, JOnAs itd.

2.4 Strežniška javanska zrna (EJB)

Strežniška javanska zrna (*Enterprise Java Beans – EJB*) so, kot pove že ime, srežniške (*server-side*) javanske komponente. Komponente je potrebno razviti natančno po specifikacijah [**?**]. Tako razvite komponente lahko namestimo na EJB kompatibilen aplikacijski strežnik (EJB vsebnik), ki upravlja izvajanje komponent ter jim nudi določene storitve, o katerih smo že govorili. Strežniška javanska zrna vsebujejo zgolj poslovno logiko, kar razvijalcem precej olajša delo, saj se lahko posvetijo izključno razvoju poslovne logike. Poveča se tudi preglednost programskega koda, saj se izognemo prepletenosti poslovne logike in delov programskega koda, ki komponenti zagotavljajo transakcijsko podporo, zaščito podatkov, shranjevanje stanja komponente, komunikacijo itd.

Strežniška javanska zrna in celotna arhitektura J2EE je namenjena razvoju velikih distribuiranih poslovnih aplikacij, glavne prednosti EJB pred ostalimi tehnologijami so naslednje:

- visok nivo industrijske podpore,

  xEJB standard je "zrasel" na pobudo industrije in je s strani le-te tudi zelo dobro podprt. Vodilna podjetja na področje informacijske tehnlogije (Sun, IBM, Oracle, Sybase, Borland itd.) ponujajo EJB kompatibilne aplikacijske strežnike in orodja za razvoj strežniških javanskih zrn. Ker je EJB standard, lahko izbiramo med ponudnikom strežnikov. Prav tako pa je mogoč prehod od enega proizvajalca k drugemu, kar je do zdaj predstavljalo kar precejšen problem.

- razvoj strežniških javanskih zrn je omejen le na razvoj poslovne logike,

  Razvoj strežniškega javanskega zrna je omejen le na razvoj poslovne logike, ki predstavlja jedro vsake aplikacije. Upravljanje ostalih ostalih

storitev se prenese na aplikacijski strežnik. Razvijalec predstavitvenega nivoja aplikacije (grafični uporabniški vmesnik) se lahko tako posveti samo svojemu delu aplikacije, saj programski kod za predstavitev podatkov namreč ne vsebuje metod poslovne logike ali metod za zagotovitev ostalih storitev kot so: komunikacija s komponentami vmesnega nivoja, nadzor upravljanja komponent, zaščita, transakcije itd. Razvoj poslovne opreme se tako pohitri, kar bistveno zmanjša stroške razvoja.

- ponovna uporabnost že razvitih komponent,

  Strežniška javanska zrna omogočajo zelo visok nivo ponovne uporabnosti že razvitih (obstoječih) komponent. Že samo dejstvo, da komponenta vsebuje le poslovno logiko in dejansko "obnašanje" komponente dololočimo v fazi nameščanja, omogoča komponenti izvajanje v različnih okoljih.

- hiter razvoj komponent,

  Standardni aplikacijski vmesnik strežniških javanskih zrn ter vmesnikov za dostop do le-teh omogoča razvoj orodij za avtomatsko generiranje EJB komponent. Na tržišču je precej komercialnih produktov (JBuilder, Visual Age for Java, Rational Rose itd.), ki omogočajo vizualno načrtovanje strežniških javanskih zrn ter avtomatsko generiranje le-teh iz podanih specifikacij. Problem teh orodij je, da ponavadi nudijo podporo le za lastne aplikacijske strežnike, kar jim precej omejijo uporabo. Že razvoj srežniških javanskih zrn po klasični metodi (tipkanje celotnega programskega koda) je hitro, saj je celotni vmesni nivo implementiran na nivoju aplikacijskega strežnika.

- ločevanje med razvojem, nameščanjem (*deploy*) in administriranjem.

  Pri razvoju strežniških javanskih komponent je mogoče na enostaven način ločiti vloge med razvijalci, saj so posamezne faze razvoja med sabo striktno ločene. Posamezna vloga v razvoju programskega sistema je lahko implementirana s strani ene osebe. Zaželjeno pa je, da se vloge razdelijo na različne osebe ali celo na različne organizacije. Na kratko

povzamimo posamezne vloge razvijalcev v posameznih sklopih razvoja celotne poslovne programske opreme, pri tem pa se osredotočimo samo na del razvoja, administriranja in vzdrževanja.

- razvijalec strežniških javanskih zrn (*Enterprise Bean Developer*)

  Razvijalec strežniških javanskih zrn je odgovoren za razvoj komponent, ki predstavljajo poslovno logiko sistema in podajanje osnovnih specifikacij za postopek namestitve. Produkt je arhiv (*.jar*), ki vsebuje prevedene strežniške komponente.

- razvijalec spletnih komponent (*Web Component Developer*)

  Razvoj spletnih komponent (Java Servlet, JSP), statičnih spletnih strani (HTML, XML, WML itd.) ter ostalih komponent (Java Beans) je naloga razvijalca spletnih komponent. Produkt, ki ga poda v uporabo osebi za namestitev komponent, je spletni arhiv (*.war*). Ta vsebuje zgoraj naštete komponente ter specifikacije namestitvene datoteke.

- razvijalec aplikacijskega odjemalca (*J2EE Application Client Developer*)

  Odgovornost za razvoj komponent grafičnega uporabniškega vmesnika ter celotne aplikacije, ki jo izvajajo uporabniki, je na plečih razvijalcev aplikacijskega odjemalca. Komponente z vsemi pripadajočimi datotekami je potrebno v obliki arhiva (*.jar*) predati administratorju aplikacije.

- administrator aplikacije (*Application Deployer and Administrator*)

  Administrator aplikacije je oseba oz. organizacija, ki je odgovorna za ustrezno konfiguracijo in namestitev J2EE aplikacije ter administracijo računalniške infrastrukture aplikacijskega strežnika. Pri namestitvi mora administrator dosledno slediti navodilom razvijalcev posameznih sklopov komponent. Med naloge administratorja spada izdelava arhiva celotne J2EE aplikacije (*.ear*), konfiguriranje J2EE aplikacije za izvajanje na določenem aplikacijskem strežniku, verifikacija kompatibilnosti arhiva in J2EE specifikacij ter namestitev arhiva na aplikacijski strežnik.

Strežniška javanska zrna delimo na tri različne tipe, in sicer sejna strežniška zrna (*Enterprise Session Beans*), sporočilna strežniška javanska zrna (*Enterprise Message-Driven Beans*) in entitetna strežniška javanska zrna (*Enterprise Entity Beans*).

2.5 Sejna strežniška zrna

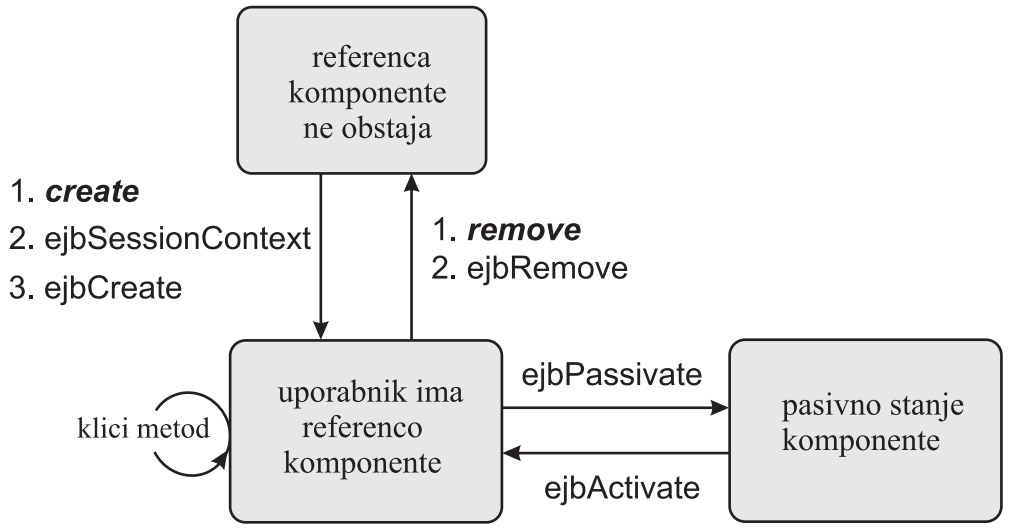
Sejno strežniško javansko zrno (*Sesssion Bean*) je komponenta, ki izvaja poslovno logiko za posameznega (enega) odjemalca J2EE aplikacije. Ime "Session Bean" je pri teh komponentah uporabljeno zato, ker je življensjki cikel komponente vezan le na odjemalca, ki to komponento uporablja. Te komponente "živijo" torej le, dokler jih odjemalec uporablja. Uporabljajo se za izvajanje poslovne logike ter za dostop do ostalih komponent in J2EE virov. Glede na upravljanje stanja in referenc komponent s strani EJB vsebnika ločimo dve vrsti teh komponent, in sicer sejno strežniško zrno s stanjem *Stateful Session Bean* in sejno strežniško zrno brez stanja *Stateless Session Bean*.

2.5.1 Sejno strežniško zrno s stanjem

Stanje posameznega objekta oz. komponente določajo vrednosti instančnih spremenljivk. V tem primeru določajo instančne spremenljivke stanje unikatno za določenega uporabnika, ki to zrno uporablja. Največkrat se ta zrna uporabijo v primeru, ko želimo ohraniti stanje med različnimi klici metod poslovne logike. Stanje zrna se ohrani samo tako dolgo, dokler uporabnik hrani referenco, in se ne hrani trajno. Prav tako se stanje zrna izgubi ob morebitnem izpadu EJB vsebnika.

Življenjski cikel sejnega strežniškega javanskega zrna s stanjem (*Stateful Session Beans*) se prične s klicem metote `create` s strani odjemalca. EJB vsebnik na odjemalčevo zahtevo reagira s klicem metode `setSessionContext` in `ejbCreate`, ki ustvari objekt. Zrno je sedaj pripravljeno na klice metod poslovne logike. Ko zrna ne želimo več uporabljati je potrebno poklicati metodo `remove`. Vsebnik ob klicu te metode, pokliče metode `ejbRemove`,

ki sproži uničenje objekta. V stanju pripravljenosti zrna lahko EJB vsebnik glede na frekvenco uporabe le to postavi v pasivno oz. aktivno stanje. Kot je razvidno iz opisanega in slike 2.3 ima odjemalec nadzor le nad ustvarjanjem in uničenjem reference.

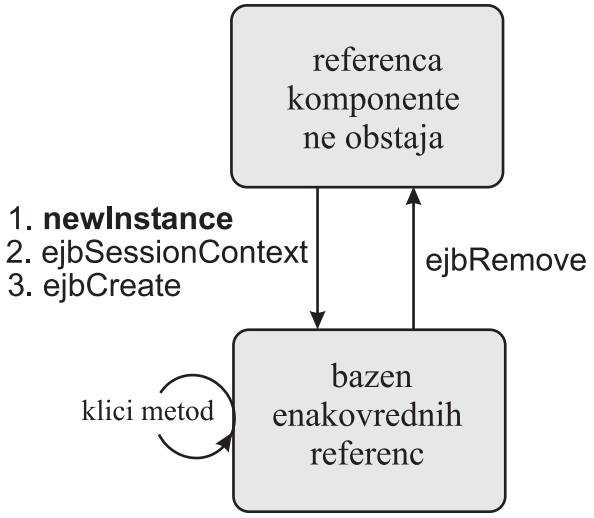


Slika 2.3: Življenjski cikel sejnega strežniškega javanskega zrna s stanjem.

2.5.2 Sejno strežniško zrno brez stanja

Za razliko od sejnega strežniškega javanskega zrna s stanjem, sejna strežniška javanska zrna brez stanja *Stateless Session Beans* ne vsebujejo stanja. Instančne spremenljivke zrna imajo vpliv le na en klic metode poslovne logike. Vsa zrna so torej identična in ni potrebe, da bi odjemalec ob več klicih metod vedno uporabljal isto referenco. Referenco si torej lahko deli več odjemalcev, kar zagotavlja hitrejše izvajanje aplikacij z več odjemalci.

Zrno se lahko v življenjskem ciklu nahaja samo v stanju pripravljenosti ali pa ga ni. Življenjski cikel sejna strežniška javanska zrna brez stanja je predstavljen na sliki 2.4.



Slika 2.4: Življenjski cikel sejnega strežniškega javanskega zrna brez stanja.

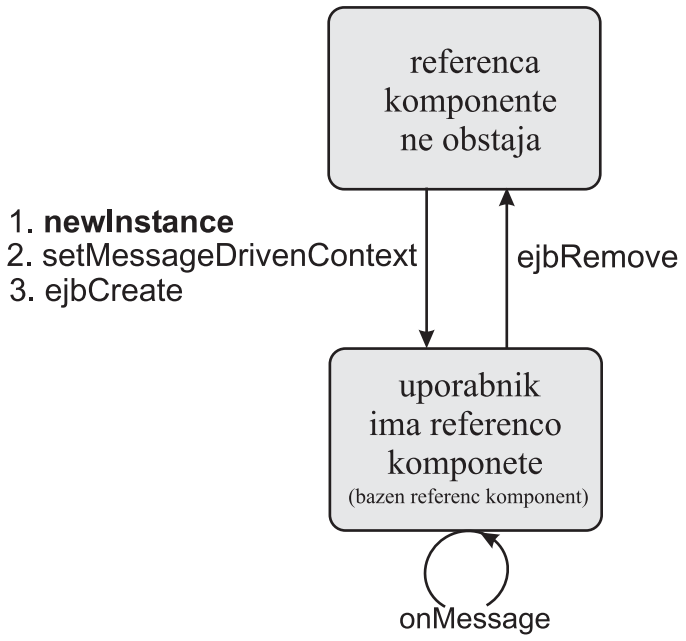
2.6 Sporočilna strežniška zrna

Sporočilne strežniške javansko zrno (*Message-Driven Bean*) temeljija na JMS (*Java Messaging Service*), kar komponentam omogoča pošiljanje in sprejemanje sinhronih in asinhronih sporočil. Podobno kot *Stateless Session* komponente ne vsebujejo stanja, ki je odvisno od posameznega odjemalca.

EJB vsebnik upravlja celoten življenjski cikel komponent (slika 2.5). Glede na potrebo po komponentah, ustvari vsebnik zalogo (*pool*) referenc, ki so na voljo odjemalcem.

2.7 Enitetna strežniška zrna

Stanje enititetnega strežniškega zrna (*Entity Bean*) je trajno, in se hrani v podatkovni bazi. Zrno predstavlja entiteto (tabelo) podatkovne baze. Vsaka referenca komponente predstavlja vrstico tabele podatkovne baze. EJB vsebnik zagotavlja konsistentno stanje komponente in podatkovne baze, prav tako pa zagotavlja obstojnost komponente tudi ob uničenju reference oz. ob morebitnem izpadu EJB vsebnika. Ker je stanje komponente odvisno od stanja v



Slika 2.5: Življenjski cikel sporočilnega strežniškega zrna.

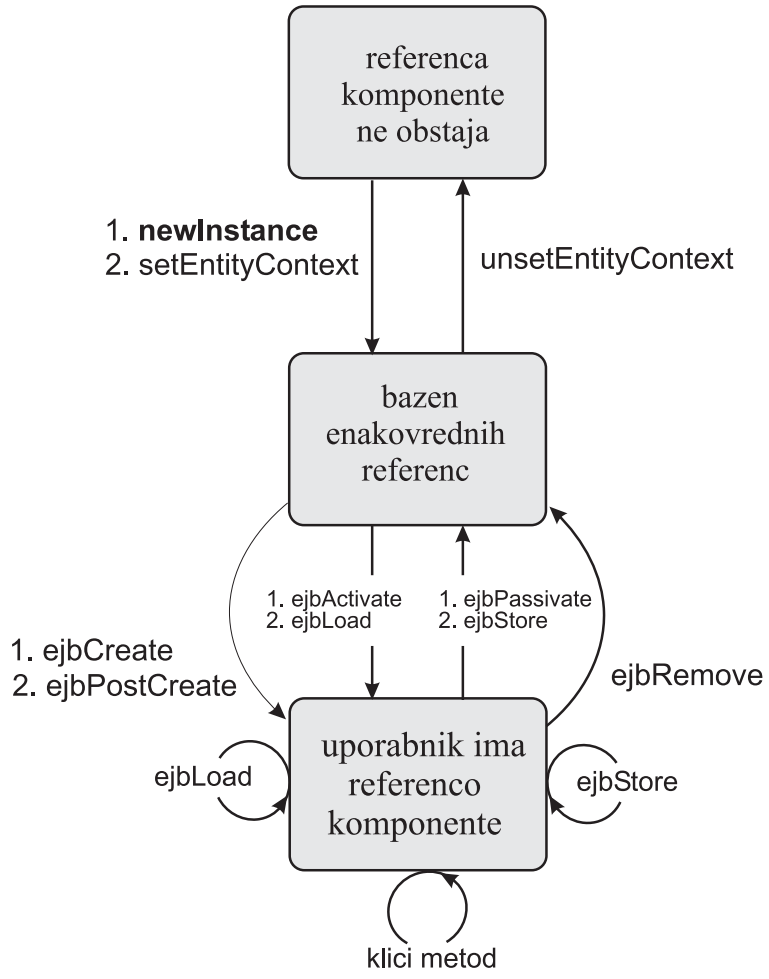
podatkovni bazi, je le-to neodvisno od posameznega odjemalca. Referenco komponente si tako lahko deli več odjemalcev.

Celotni življenjski cikel *Entity* komponente upravlja EJB vsebnik (slika 2.6). Ker stanje posamezne komponente ni odvisno od odjemalca temveč od stanja v podatkovnem viru, si lahko referenco komponente deli več odjemalcev. EJB vsebnik tako glede na povpraševanje po komponentah ustvari bazen enakovrednih referenc komponent, ki so na voljo odjemalcem.
Glede na frekvenco uporabe posamezne reference komponente lahko EJB vsebnik le-to postavi v aktivno oz. pasivno stanje.

2.7.1 *Bean Managed Persistence*

Rutine za dostop do podakovnih virov, povpraševanja in shranjevanje podatkov ter upravljanje relacij med posameznimi entitetami je pri *Bean Managed Entity* zrnu prepuščeno razvijalcu. Razvijalec ima torej dostop do po-

datkovnega vira, kar omogoča razvoj optimalnega programskega koda za upravljanje s podatki.



Slika 2.6: Življenjski cikel entitetnega strežniškega zrna.

Programski kod je optimiziran le za podatkovni vir, ki ga uporabljamo. Ta prednost pa se lahko kar hitro spremeni v slabost, in sicer ko želimo spremeniti izvajalno okolje komponente (drugi podatkovni vir, drugi EJB vsebnik, ...). V najslabšem primeru je lahko celotni izvorni kod zrna v tem primeru neuporaben in je potrebno razviti povsem novo zrno.

2.7.2 *Container Managed Persistence*

Za razliko od *Bean Managed Entity* zrna, *Container Managed Entity* zrno ne vsebuje programskega koda za dostop do podatkovnega vira. Upravljanje stanja je v celoti prepuščeno EJB vsebniku. Programski kod tako ne vsebuje metod za dostop do podatkovnega vira oz. za iskanje/ažuriranje podatkov. Komponenta je sestavljena zgolj iz metod, ki predstavljajo poslovno logiko. Prav tako je vsebniku prepuščeno upravljanje relacij med komponentami. Oboje definiramo šele v času nameščanja zrna in lahko brez posega v izvorni kod komponente tudi spremenimo. Kot je razvidno iz opisanega, je razvoj *Container Managed* zrna enostavnejši od *Bean Managed* entitetnega zrna, prav tako pa je razvito zrno neodvisna od EJB vsebnika oz. podatkovnega vira.

V nadaljevanju poglavja si bomo natančneje ogledali upravljanje obstojnosti strežniških javanskih zrn in relacij med njimi s strani EJB vsebnika. Osredotočili se bomo na CMP 2.0, ki je nadgradnja prejšnje verzije, prinaša pa tudi številne novosti in izboljšave. Opis implementacije temelji na aplikacijskem strežniku JBoss [**?**], ki smo ga uporabili pri implementaciji praktičnega dela diplomske naloge.

Kot smo že omenili, je upravljanje obstojnosti podatkov prepuščeno EJB vsebniku. Toda kako do podatkov oz. kako definirati katere podatke želimo upravljati z določeno komponento? To storimo preprosto z definicijo abstraktnih metod za dostop do podatkov (metode *get/set*). Ni potrebno definirati instančnih spremenljivk, prav tako pa je telo metod prazno (abstraktne metode). Vse ostale informacije o komponenti, EJB vsebniku posredujemo z namestitvenimi datotekami. Za implementacijo metod (*get/set*) poskrbi EJB vsebnik, prav tako po potrebi ustvari tabelo v podatkovni bazi. Vsebino le-te pa v življenjskem ciklu komponente tudi ustrezno spreminja, tako da je stanje komponente enako stanju v podatkovni bazi. Primer entitetnega zrna in preproste namestitvene datoteke sledi v poglavju 2.9.

Bistvena izboljšava v CMP 2.0 je upravljanje relacij med entitetnimi strežniškimi zrni (CMR – *Container Managed Relationships*). Razvijalcem ni potrebno pisati programskega koda za upravljanje relacij in za zagotavljanje

konstitentnega stanja komponent v relaciji in stanja v podatkovni bazi. Specifikacije CMP 2.0 podpirajo tri tipe relacij (ena-proti-ena, ena-proti-mnogo in mnogo-proti-mnogo). Omejitev pri uporabi CMR pa je vsekakor dejstvo, da lahko relacije definiramo samo na lokalni ravni (lokalni vmesniki). Relacije lahko ustvarimo torej samo med entitetnimi zrni, ki se izvajajo v istem javanskem navideznem stroju (JVM). Definicijo relacije med dvema entitetnima zrnima sestavljata dva koraka: definicija relacije v entitetnem strežniškem zrnu in definicija relacije v ustreznih namestitvenih datotekah (`ejb-jar.xml`, `jbosscmp-jdbc.xml`, ...). Podobno kot pri podatkih zrna, relacije definiramo z abstraktnima metodama (*get/set*). Zrno, ki se nahaja na drugi strani relacije in kardinalnost določimo s tipom, ki ga abstraktni relacijski metodi sprejmeta (*set*) oz. vračata (*get*). Pri relacijah ena-proti-ena oz. pri komponenti, ki dostopa do komponente na strani ena, relacijski metodi operirata z lokalnim vmesnikom komponente. V nasprotnem primeru (ena-proti-mnogo, mnogo-proti-mnogo) relacijski metodi operirata s kolekcijo podatkov (`java.lang.Collection` ali `java.lang.Set`). Podobno kot pri stanju entitetne komponente tudi za relacije velja, da je za implementacijo relacijskih komponent zadolžen EJB vsebnik.

Primer 2.1 Relacije med entitetnimi strežniškimi zrni.

```
public class Organizacija implements EntityBean {
    // dostopamo do zrna na strani mnogo
    public abstract Set getClan();
    public abstract void setClan(Set pOseba);
}

public class Oseba implements EntityBean {
    // dostopamo do zrna na strani ena
    public abstract Organizacija getOrganizacija();
    public abstract void setOrganizacija(Organizacija pOrganizacija);
}
```

Preprosti primer implementacije relacije ena-proti-mnogo je prikazan na primeru 2.1. Kot je razvidno iz primera smo implementirali dve entitetni zrni. Oseba je lahko član le ene organizacije, medtem ko lahko ima pozamezna organizacija več članov (ena-proti-mnogo). Kot smo že omenili definicija abstraktnih relacijskih metod ni dovolj. Kako se bo relacija dejansko preslikala, navedemo

Primer 2.2 Namestitvena datoteka `ejb-jar.xml`.

```
<ejb-relation >
  <ejb-relation-name>Organizacija_Oseba_1_to_n</ejb-relation-name>
  <ejb-relationship-role >
    <ejb-relationship-role-name>Organizacija_Oseba</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source >
        <ejb-name>Oseba</ejb-name>
      </relationship-role-source>
      <cmr-field >
        <cmr-field-name>organizacija</cmr-field-name>
        <cmr-field-type>java.util.Set</cmr-field-type>
      </cmr-field>
  </ejb-relationship-role>
  <ejb-relationship-role >
   <ejb-relationship-role-name>Oseba_Organizacija</ejb-relationship-role-name>
     <multiplicity>Many</multiplicity>
     <relationship-role-source >
       <ejb-name>Organizacija</ejb-name>
     </relationship-role-source>
       <cmr-field >
       <cmr-field-name>clani</cmr-field-name>
     </cmr-field>
  </ejb-relationship-role>
</ejb-relation>
```

v namestitvenih datotekah. Primer 2.2 navaja del namestitvene datoteke `ejb-jar.xml`, ki je namenjen definiranju relacij. EJB vsebnik JBoss pozna dva tipa preslikav relacij. Relacije se lahko preslikajo s pomočjo posebne relacijske tabele v podatkovni bazi oz. z uporabo tujega ključa. Način preslikave je opisan v posebni namestitveni datoteki `jbosscmp-jdbc.xml`.

Poizvedovalni jezik EJB QL

Do zdaj smo si pogledali upravljanje stanja entitetnih komponent. Seveda pa samo upravljanje stanja komponent ni dovolj, aplikacije oz. uporabniki aplikacij opravljajo razne poizvedbe (iskanja). Poizvedbe implementiramo v

posebnih metodah, ki se imenujejo *finder*. Te metode so tipa `ejbFindByXXX` in `ejbSelectXXX`. Razlika med njima je le v tem, da metode `find` vračajo le vmesnik zrna, metode `select` pa poljuben podatkovni tip. V izvornem kodu komponente navedemo le abstaktne definicije metod, definicijo posamezne metode pa navedemo v namestitvenih datotekah. Poizvedbe in ažuriranja se v podatkovnih bazah izvajajo z jezikom SQL (*Structured Query Language*). SQL ni povsem kompatibilen med različnimi ponudniki podatkovnih baz in zato ni primeren za razvoj *finder* metod. Zato so se v CMP 2.0 odločili, da za poizvedbe uporabijo EJB QL (*Enterprise Java Beans Query Language*). EJB QL je zelo podoben SQL jeziku le da ni tako obširen in je zato enostavnejši. Stavke EJB QL, ki jih definiramo v namestitveni datoteki, se v fazi nameščanja preslikajo v SQL stavke podatkovne baze, ki jo za shranjevanje stanja uporablja določena komponenta. Primer *finder* metode in del namestitvene datoteke `ejb-jar.xml`, ki namenjen definiranju poizvedb, je prikazan na primeru 2.3.

Primer 2.3 Primer *finder* metode in namestitvene datoteke ejb-jar.xml.

```
public class Oseba implements EntityBean {
    // finder metoda (iskanje osebe po imenu)
    public abstract Set findByIme(String ime) throws FinderException;
}

<query>
 <query-method>
   <method-name>findByIme</method-name>
   <method-params>
     <method-param>java.lang.String</method-param>
   </method-params>
 </query-method>
 <ejb-ql>
   <![CDATA[SELECT OBJECT(o) FROM Oseba o WHERE o.ime = ?1]]
 </ejb-ql>
</query>
```

Preslikava stavkov jezika EJB QL v SQL stavke je odvisna od uporabljene podatkovne baze, in jo definiramo na nivoju EJB vsebnika.

2.8 Vmesniki

Odjemalci lahko do strežniških komponent dostopajo le preko vmesnikov[1], ki definirajo uporabniški pogled na komponento. Tak pristop je nujen iz vsaj dveh razlogov; vsa kompleksnost poslovne logike je odjemalcu skrita, omogoča pa tudi neodvisnot razvoja poslovnega nivoja in nivoja predstavitve podatkov. Arhitektura z vmesniki omogoča transparentnost celo tako radikalnih sprememb kot je sprememba komponente iz *Bean Managed* v *Container Managed*. Prvi korak pri razvoju komponente je torej načrtovanje vmesnikov, pri tem moramo najprej razmisliti o načinu dostopa do komponente (lokalni/oddaljen).

Oddaljen dostop do strežniških komponent

Če želimo komponenti zagotoviti oddaljen dostop, je potrebno komponenti dodati vmesnika *Remote Interface* in *Home Interface*. Prvi definira vmesnik metod s poslovno logiko, medtem ko v drugem definiramo vmesnik metod za upravljanje življenjskega cikla komponente (`create, remove`), pri *Enitity* komponentah pa tudi do metod za iskanje (`findByPrimaryKey, ...`).
Značilnosti oddaljenega dostopa do komponent so naslednje:

- Možnost izvajanja na oddaljenem računalniku in znotraj drugega javanskega navideznega stroja (JVM). Seveda pa to ni nujno, odjemalec je lahko tudi druga strežniška komponenta, ki jo izvaja isti EJB vsebnik.
- Oddaljen vmesnik zagotavlja lokacijsko transparentnost strežniških komponent glede na odjemalca.
- Odjemalci komponente so lahko druge komponente, oddaljena aplikacija ali spletne komponente.

Za oddaljen dostop se odločamo predvsem, ko je odjemalec oddaljena aplikacija ali ko želimo komponenti zagotoviti lokacijsko transparentnost.

[1] Z vmesniki definiramo dostop do *Session* in *Entity* komponent. Poglavje se ne nanaša na *Message-Driven* komponente.

Lokalen dostop do strežniških komponent

Lokalen dostop je rezerviran le za komponente, ki jih izvaja isti EJB vsebnik oz. se izvajajo znotraj istega javanskega navideznega stroja. Odjemalci so lahko torej le spletne komponente in ostale strežniške komponente. Ker je dostop do komponente lokalen, lokacijska transparentnost ni zagotovljena. Za razvoj strežniške komponente z lokalnim dostopom je potrebno komponenti definirati vmesnika *Local Interface* in *Local Home Interface*. Funkcija obeh vmesnikov je podobna kot pri oddaljenem dostopu. Prvi definira vmesnik do metod s poslovno logiko, drugi pa do metod za upravljanje življenjskega cikla komponente.
Za lokalne vmesnike se odločamo predvsem zaradi pohitritve klicev metod poslovne logike, saj so oddaljeni klici metod lahko precej dragi (časovno). Lokalni vmesniki pa so nujni pri *Container Managed* komponentah, ki vsebujejo relacije do drugih komponent.

2.9 Primer J2EE aplikacije

Glede na to, da smo že kar precej besed namenili opisu tehnologije je prav, da si vse skupaj ogledamo tudi na praktičnem primeru. Primer prikazuje dve strežniški komponenti, in sicer *Session* komponento, ki izvaja poslovno logiko ter dostopa do *Entity* komponente. Da bi prikazali lokalen oz. oddaljen dostop do komponent, smo se odločili za oddaljen dostop do *Session* komponente. Ta pa preko lokalnih vmesnikov dostopa do metod poslovne logike *Entity* komponente. Za popolnejšo demonstracijo pa potrebujemo seveda še odjemalca, ki je v našem primeru oddaljena aplikacija. Podan zgled ne vsebuje delov, ki bi bili odvisni od posameznega EJB vsebnika in ga lahko namestimo na poljuben EJB vsebnik.
Najprej si poglejmo korake pri razvoju strežniških javanskih zrn oz. celotne J2EE aplikacije:

1. Definiranje in razvoj vmesnikov.

2. Razvoj strežniških komponent (poslovna logika).

3. Razvoj datoteke za namestitev (*deployment descriptor*).

4. Nameščanje komponent na aplikacijski strežnik oz. EJB vsebnik.

5. Implementacija odjemalcev (oddaljena aplikacija, JSP, Applet, ...).

Vsaka strežniška komponenta je sestavljena iz več datotek, zato je zaželjeno, da se pri razvoju komponent držimo standardnih imen komponent in vmesnikov. Tabela 2.1 prikazuje poimenovanja komponent, ki so uporabljena v našem primeru.

Tabela 2.1: Standardno poimenovanje sestavnih delov EJB aplikacije.

vmesnik/komponenta	sintaksa	primer
razred strežniškega javanskega zrna	<ime>Bean	OsebaBean
vmesnik *Home*	<ime>Home	OsebaHome
vmesnik *Remote*	<ime>	Oseba
vmesnik *LocalHome*	Local<ime>Home	LocalOsebaHome
vmesnik *Local*	Local<ime>	LocalOseba
jar arhiv komponent	<ime>JAR	OsebaJAR

Preden se resneje lotimo dela, si je potrebno komponente, vmesnike in datoteke za nameščanje organizirati v primerno strukturo. Ponavadi so komponente in vmesniki ločeni in se nahajajo v različnih imenikih. Potrebno pa je dodati tudi imenik `META-INF`, kjer se nahajajo datoteke za nameščanje.
Razvoja aplikacije, če vse skupaj implementira le ena oseba, se navadno najprej lotimo na najnižjem nivoju. V našem primeru je to *Entity* komponenta (`OsebaBean`), ki skrbi za vodenje podatkov o osebah. Če sledimo že naštetim korakom pri razvoju komponent, se najprej lotimo definiranja vmesnikov za dostop do komponente. Ker bomo do komponente dostopali le s *Session* komponento, torej le lokalno, potrebujemo le vmesnika *Local* in *LocalHome*.

Primer 2.4 Vmesnik *Local* komponente OsebaBean.

```
package zgled;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface LocalOseba extends EJBLocalObject {
    public abstract java.lang.Long getId() throws RemoteException;
    public abstract void setId(java.lang.Long id ) throws RemoteException;
    public abstract java.lang.String getIme() throws RemoteException;
    public abstract void setIme(java.lang.String ime ) throws RemoteException;
    public abstract java.lang.String getPriimek() throws RemoteException;
    public abstract void setPriimek(java.lang.String priimek ) throws RemoteException;
}
```

Primer 2.4 prikazuje definicijo lokalnega vmesnika za dostop do komponente OsebaBean. Vmesnik razširja vmesnik `javax.ejb.EJBLocalObject` in vsebuje zgolj vmesnik za metode poslovne logike. V našem primeru so to podatki o osebi (id, ime in priimek).

Vmesnik *LocalHome* (prikazan na primeru 2.5) vsebuje vmesnik za metode, ki služijo upravljanju življenjskega cikla komponente in metod za dostop do podatkov (`findByPrimaryKey, ...`) Entity komponent. Vmesnik razširja vmesnik `javax.ejb.EJBLocalHome`, ki ga morajo razširiti vsi lokalni vmesniki komponent.

Primer 2.5 Vmesnik *LocalHome* komponente OsebaBean.

```
package zgled;
import java.rmi.RemoteException;
import javax.ejb.*;
import java.util.*;

public interface LocalOsebaHome extends EJBLocalHome {
    public LocalOseba create(java.lang.Long pId, String pIme, String pPriimek) throws
CreateException;
    public LocalOseba findByPrimaryKey(java.lang.Long id) throws FinderException;
    public Enumeration findByIme(String ime) throws FinderException;
    public Collection findAll() throws FinderException;
}
```

Primer 2.6 *Entity* komponenta OsebaBean.

```
package zgled;
import javax.ejb.*;
import java.util.*;
public class OsebaBean implements javax.ejb.EntityBean
{
    public java.lang.Long id;
    public String ime;
    public String priimek;
    public java.lang.Long ejbCreate(java.lang.Long pId, String pIme, String pPriimek) throws
CreateException {
       this.id = pId;
       this.ime = pIme;
       this.priimek = pPriimek;
       return pId;
    }
    public void ejbPostCreate(Long pId, String pIme, String pPriimek) { }
    public void ejbLoad() { }
    public void ejbStore() { }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void setEntityContext(javax.ejb.EntityContext ctx) { }
    public void unsetEntityContext() { }
    public void ejbRemove() { }
    public java.lang.Long getId()
    { return id; };
    public void setId(java.lang.Long id )
    { this.id = id; }
    public java.lang.String getIme() // Ime osebe
    { return ime; };
    public void setIme(java.lang.String ime )
    { this.ime=ime; }
    public java.lang.String getPriimek() // Priimek osebe
    { return priimek; };
    public void setPriimek(java.lang.String priimek )
    { this.priimek = priimek; };
}
```

Entity komponenta OsebaBean razširja vmesnik `javax.ejb.Entity`. Implementira metode za upravljanje življenjskega cikla komponente (`ejbXXX`). Te motode po potrebi kliče EJB vsebnik in so uporabniku nedosegljive. Komponenta vsebuje tudi implementirane metode poslovne logike. V našem primeru smo se odločili za *Container Managed* komponento, zato implementacija metod za upravljanje stanja komponente ni potrebna. Telo teh metod (`setXXX in getXXX`) je prazno, saj te metode implementira in izvede EJB vsebnik. Izvorni kod komponente je prikazan na primeru 2.6. Za razliko od ravnokar opisane komponente, želimo *Session* komponenti omogočiti dostop iz oddaljenih aplikacij. Zatorej je potrebno komponenti definirati Remote in Home vmesnika. Prvi je prikazan na primeru 2.7 in definira vmesnik metod poslovne logike. Poslovno

Primer 2.7 Vmesnik *Remote* komponente Seja.

```
package zgled;
import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.*;

public interface Seja extends EJBObject {
    public void createOseba(java.lang.Long id, String ime, String priimek) throws
RemoteException, CreateException;
    public String getOseba() throws CreateException, FinderException, RemoteException;
    public String pozdrav() throws RemoteException;
}
```

logiko komponente predstavljata metodi `pozdrav` in `getOseba`. Metoda pozdrav vrne niz `"Dobro jutro!"`, medtem ko druga metoda preko lokalnih vmesnikov dostopa do komponente OsebaBean ter v obliki niza vrne vse entitete, ki se nahajajo v podatkovni bazi. Primer 2.8 prikazuje vmesnik *Home* komponente, ki definira vmesnik za metode življenjskega cikla komponente.

Primer 2.8 Vmesnik *Home* komponente Seja.

```
package zgled;
import java.rmi.RemoteException;
import javax.ejb.*;

public interface SejaHome extends EJBHome {
  Seja create() throws RemoteException, CreateException;
}
```

Komponenta SejaBean implementira vmesnik `javax.ejb.Session` in je dejanska implementacija že opisanih vmesnikov. V metodi ejbCreate smo implementirali dostop do komponente OsebaBean. Metodo ejbCreate pokliče vsebnik ob ustvarjanju reference komponente. Ta pristop smo lahko uporabili, ker smo implementirali Stateful Session komponento, referenco komponente ima torej samo odjemalec, ki jo je ustvaril. Metoda `createOseba` posreduje podatke metodi `OsebaBean.create`, ki ustvari entiteto oseba. Podatke o vseh osebah, ki so že shranjene v podatkovni bazi vrne metoda `getOseba`. Razlog, da metoda ne vrne podatkov v obliki množice referenc entitet je v tem, da oddaljena aplikacija ne more dostopati do komponente OsebaBean, saj ima ta na voljo le lokalne vmesnike. Izvorni kod komponente je prikazan na primeru 2.9.

Primer 2.9 *Session* komponenta Seja.

```
package zgled;
import java.rmi.*;
import javax.ejb.*;
import javax.naming.*;
import java.util.*;

public class SejaBean implements SessionBean {

    private LocalOsebaHome home;
    public void createOseba(java.lang.Long id, String ime, String priimek) throws
CreateException, RemoteException {
      home.create(id, ime, priimek);
    }
    public String getOseba() throws CreateException, RemoteException, FinderException {
      String result = "";
      Collection c = home.findAll();
      Iterator it = c.iterator();
      while (it.hasNext()) {
          LocalOseba os = (LocalOseba)it.next();
          result += os.getIme()+" "+os.getPriimek() +"\n";
      }
      return result;
    }
    public String pozdrav() {
      return "Dobro jutro!";
    }
    public void ejbCreate() {
       try {
           InitialContext ctx = new InitialContext();
           home = (LocalOsebaHome)ctx.lookup("local/OsebaBean");
           System.out.println(home);
       }
       catch (Exception ex) {
           System.out.println("Napaki pri referenciranju Entity komponente!");
           ex.printStackTrace();
       }
    }
    public void ejbRemove() { }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void setSessionContext(SessionContext sc) { }
}
```

Naslednji korak v razvoju J2EE aplikacije je razvoj datoteke za nameščanje (*deployment descriptor*). Ta korak je zelo pomemben in kar precej zapleten. Ponudniki EJB vsebnikov ponavadi zraven le-teh priložijo tudi orodje za nameščanje (*Deployment tool*), kar razvijalcem ponavadi zelo olajša delo. Datoteka za nameščanje (`ejb-jar.xml`) vsebuje podatke o vsaki komponenti in relacijah med njimi. Datoteka za nameščanje naše testne aplikacije je prikazana v primeru 2.10.

Primer 2.10 Namestitvena datoteka komponent OsebaBean in SejaBean.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar>
  <description>Prva EJB aplikacija</description>
  <display-name>Preprost primer</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>zgledSeja</ejb-name>
      <home>zgled.SejaHome</home>
      <remote>zgled.Seja</remote>
      <ejb-class>zgled.SejaBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
    <entity>
      <ejb-name>OsebaBean</ejb-name>
      <local-home>zgled.LocalOsebaHome</local-home>
      <local>zgled.LocalOseba</local>
      <ejb-class>zgled.OsebaBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Long</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field><field-name>id</field-name></cmp-field>
      <cmp-field><field-name>ime</field-name></cmp-field>
      <cmp-field><field-name>priimek</field-name></cmp-field>
      <primkey-field>id</primkey-field>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

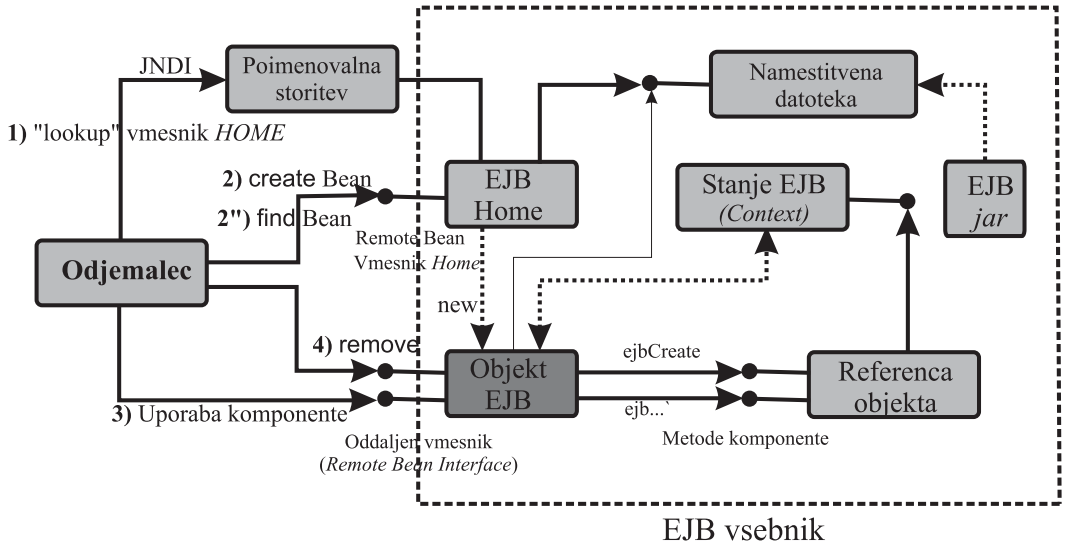
Primer 2.11 Odjemalec (oddaljena aplikacija).

```
import java.util.*;
import javax.ejb.*;
import zgled.*;
import javax.rmi.PortableRemoteObject;
import javax.naming.*;

public class Odjemalec {
    public static void main(String args[])  throws Exception {
    System.setProperty("java.naming.factory.initial",
              "org.jnp.interfaces.NamingContextFactory");
    System.setProperty("java.naming.provider.url", "localhost");
    InitialContext initialContext = new InitialContext();
    try {
        java.lang.Object objRef = initialContext.lookup("zgledSeja");
        SejaHome home = (SejaHome)PortableRemoteObject.narrow(objRef,
  SejaHome.class);
        System.out.println("Imam referenco komponente = "+home);
        Seja bean = home.create();
        System.out.println(bean.pozdrav());
        System.out.println("----------------------------------");
        bean.createOseba(new java.lang.Long(10), "Barbara", "Frešer");
        bean.createOseba(new java.lang.Long(3), "Janez", "Krajnc");
        bean.createOseba(new java.lang.Long(4), "Boris", "Novak");
        bean.createOseba(new java.lang.Long(5), "Evgen", "Burić");
        System.out.println(bean.getOseba());
      }
      finally {
          initialContext.close();
      }
    }
}
```

Vse opisane komponente in vmesnike prevedemo na običajen način (isto kot vsako drugo javansko komponento). Prevedene komponente nato vključimo v *jar* arhiv, ki na nato namestimo (*deploy*) v EJB vsebnik. Postopek namestitve je, prav tako kot namestitvena datoteka, na različnih EJB vsebnikih različen. V našem primeru, uporabljamo EJB vsebnik JBoss, je ta postopek sila preprost.

Arhiv s komponentami je potrebno prenesti v ustrezen imenik EJB vsebnika, ki je namenjen komponentam. Po namestitivi sproži vsebnik postopek verifikacije komponent. Preveri se skladnost komponent s sprecifikacijami strežniških zrn. Ob morebitnih napakah se namestitev ne izvede.
Odjemalec (primer 2.11), aplikacija, ki jo uporablja uporabnik, je v našem primeru oddaljena aplikacija. Le-ta preko oddaljenih vmesnikov dostopa do komponent poslovne logike komponente SejaBean. Na ta način pa posredno izvaja tudi poslovno logiko komponente OsebaBean. Kot je razvidno iz primera, dostopamo do komponente z unikatnim imenom, ki ga določimo v namestitveni datoteki. Prav tako aplikacija ne vsebuje mehanizmov za komunikacijo, za le-te poskrbi EJB vsebnik.
Celoten potek izvajanja J2EE aplikacije, oz. koraki za dostop do strežniških javanskih zrn so prikazani na sliki 2.7.



Slika 2.7: Potek izvajanja J2EE aplikacije.

2.10 J2EE in spletne aplikacije

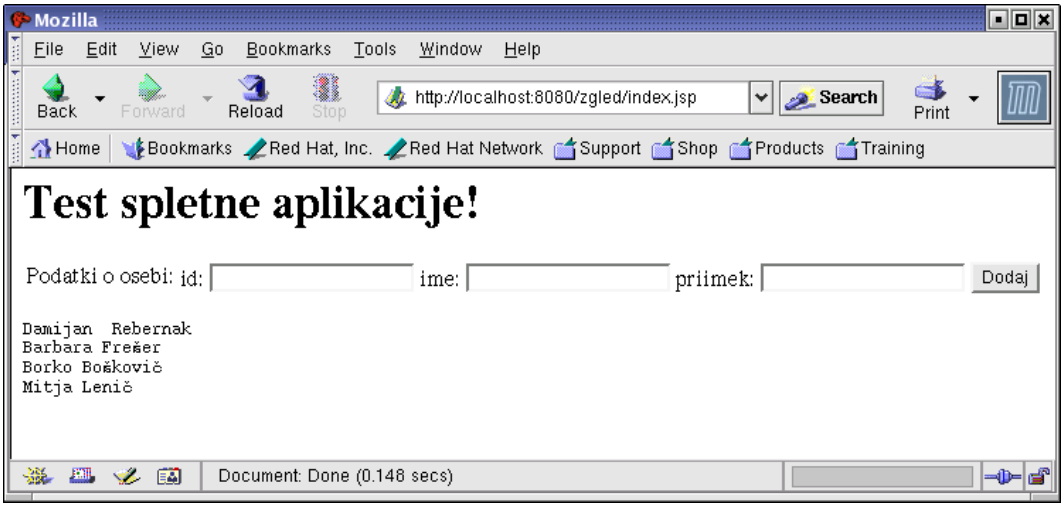
Spletne aplikacije postajajo vedno pogostejši pojav na področju poslovnih aplikacij in informacijskih sistemov. Zanimive so predvsem zaradi dosegljivosti,

saj so dostopne praktično povsod, kjer je omogočen dostop do svetovnega spleta. Pomembna prednost pa je tudi uporaba spletnega brskalnika kot odjemalca, saj le-tega večina uporabnikov že pozna in se tako čas učenja oz. privajanja uporabnikov na novo aplikacijo skrajša. Namestitev aplikacije pri vsakem uporabniku posebej zatorej ni potrebna, kar olajša postopek namestitve celotne aplikacije, administracijo in njen nadaljni razvoj oz. vzdrževanje. Vsaka sprememba aplikacije, poslovne logike ali odjemalcev (predstavitveni nivo), je tako za uporabnike povsem transparentna. Seveda pa ima vsaka stvar tudi svoje slabe lastnosti. Slabost spletnih aplikacij je predvsem v zmožnosti predstavitve podatkov oz. razvoja uporabniškega vmesnika. Pri tem smo omejeni na predstavitev v formatu HTML in na zmožnosti spletnega brskalnika, ki ga uporablja določen uporabnik. Pri tem pa naletimo še na problem spletnih brskalnikov, ki ne podpirajo identičnih standardov.
Ne gleda na vse prednosti, ki smo jih našteli (več o spletnih aplikacijah in arhitekturah le-teh si lahko bralec prebere v [**?**, **?**]), ne moremo mimo vprašanja ustreznosti uporabe oz. kdaj za spletno aplikacijo na nivoju vmesnega nivoja uporabiti arhitekturo J2EE? Postavitev in kasneje administracija celotnega sistema z arhitekturo J2EE je kar precej zahtevna in glede na to, da je večina obstoječih spletnih aplikacij namenjenih predvsem za pregled podatkov (stanja) in za določene nezahtevna ažuriranja in transakcije, je zastavljeno vprašanje na mestu. Kdaj torej za spletno aplikacijo uporabiti arhitekturo J2EE?
Po mojem mnenju je arhitektura J2EE ne samo primerna, ampak idealna rešitev za razvoj spletnih informacijskih sistemov.

2.10.1 Primer – spletni odjemalec

Spletni odjemalci J2EE aplikacije so komponente spletnega nivoja, to so v Servleti in JSP strani, o katerih smo že govorili. Tako kot vsaka komponenta ima tudi spletna komponenta dostop do komponent poslovnega nivoja, torej strežniških javanskih zrn. Dostop do komponent (lokalen/oddaljen) pa je

odvisen od EJB vsebnika in Servlet/JSP vsebnika, ki ju pri razvoju uporabljamo. Dostop do oddaaljenih vmesnikov ni vprašljiv v nobenem primeru, pazljivi pa moramo biti pri uporabi lokalnih vmesnikov entitetnih komponent, ki jih upravlja EJB vsebnik. Če te komponente vsebujejo relacije, lahko preko lokalnih vmesnikov do njih dostopamo le, če se komponente izvajajo znotraj istega javansakega navideznega stroja. Pred razvojem se moramo prepričati, če se EJB vsebnik in Servlet/JSP vsebnik izvajata v istem javanskem navideznem stroju. Primer 2.12 prikazuje preprosto JSP stran. Stran vsebuje obrazec, kamor lahko uporabnik vpiše podatke o osebi. Podatki se nato posredujejo komponenti OsebaBean, ki jih shrani v podatkovno bazo. V primeru sta prikaza oba pristopa za dostop do strežniških komponent. Do komponente SejaBean dostopamo preko lokalnih in do komponente OsebaBean preko lokalnih vmesnikov. Naslednji korak pri razvoju spletne komponente je implementacija namestitvene datoteke in namestitev komponente. V namestitveni datoteki je potrebno navesti URL (*Uniform Resource Locator*), s katerim bodo komponente spletnega nivoja dostopne uporabnikom. Postopek namestitve spletnih komponent je identičen kot pri ostalih komponentah. Komponente dodamo v *jar* arhiv k ostalim komponentam, nato pa vse skupaj namestimo v EJB vsebnik. Na slik 2.8 je prikaz rezultata izvajanja JSP strani.



Slika 2.8: Rezultat izvajanja JSP strani.

Primer 2.12 Spletni odjemalec.

```
<%@ page language="java" import="java.util.*"%>
<%@ page import="zgled.*"%>
<%@ page import="javax.rmi.PortableRemoteObject"%>
<%@ page import="javax.naming.*"%>
<HTML>
<H1>Test spletne aplikacije!</H1>
<FORM action="index.jsp" method="POST">
<TABLE>
 <TR>
  <TD>Podatki o osebi: </TD>
  <TD>id: <INPUT type="edit" name="id" value=""></TD>
  <TD>ime: <INPUT type="edit" name="ime" value=""></TD>
  <TD>priimek: <INPUT type="edit" name="priimek" value=""></TD>
  <TD><INPUT type="submit" name="gumb" value="Dodaj"></TD>
 </TR>
</TABLE>
</FORM>
<PRE>
<%
InitialContext initialContext = new InitialContext();
InitialContext ctx = new InitialContext();
try {
    LocalOsebaHome localHome = (LocalOsebaHome)ctx.lookup("local/OsebaBean");
    if ("POST".equals(request.getMethod())) {
    localHome.create(new Long(request.getParameter("id")), request.getParameter("ime"),
    request.getParameter("priimek"));
    }
    java.lang.Object objRef = initialContext.lookup("zgledSeja");
    SejaHome home = (SejaHome)PortableRemoteObject.narrow(objRef, zgled.SejaHome.class);
    Seja bean = home.create();
    out.println(bean.getOseba());
    out.println("</PRE>");
   }
finally {
    initialContext.close();
    ctx.close();
}
%>
</HTML>
```

3

If your development tools don't suck, you're not on the bleeding edge of technology.

Orodja za hiter razvoj strežniških javanskih zrn

Poglavje opisuje tehnologije za hiter razvoj strežniških javanskih zrn in celotnih aplikacij, temelječih na J2EE arhitekturi. V uvodu opišemo motivacijo za uporabo teh orodij. Glede na to, da so vsa opisana orodja razvita po principu odprtega koda, se dotaknemo tudi filozofije razvoja programske opreme po principu odprtega koda. Opis tehnologij za hiter razvoj strežniških javanskih zrn obsega tri orodja, ki pri razvoju uporabljajo različne pristope. Opisana orodja so uporabljena pri razvoju praktičnega dela diplomske naloge.

3.1 Uvod

Že preprosti primer J2EE aplikacije, prikazan v poglavju 2.9, nakazuje kompleksnost razvoja aplikacij za J2EE arhitekturo. Vsaka strežniška komponenta je sestavljena iz več delov (datotek). Minimalno število datotek, ki jih mora napisati razvijalec komponente je štiri (komponenta, vsaj dva vmesnika in namestitvena datoteka). Pri kompleksnejših komponentah pa lahko ta številka zraste tudi do deset ali pa še več. Razvoj kompleksnejše in obsežnejše aplikacije tako kar hitro postane zelo težavno opravilo. Zlahka se izgubimo v množici datotek in razvoj se na ta način bistveno upočasni. Večina dobrih lastnosti okrog razvoja strežniških komponent bi se torej tu končala. Na srečo imajo strežniške komponente eno zelo dobro lastnost, razvite morajo biti natančno po specifikacijah, ki so neodvisne od EJB vsebnika, za katerega so razvite. Podobno kot komponente so tudi vmesniki komponent standardizirani. Ideja o pohitritvi razvoja komponent je torej na dlani. Glede na to, da je razvoj strežniških zrn standardiziran, odpira možnost avtomatskega generiranja komponent iz podanih specifikacij. Razvijalec se tako osredotoči na razvoj metod s poslovno logiko, za ostale metode in vmesnike pa poda zgolj specifikacije in se generirajo avtomatsko. Ideja o avtomatskem generiranju komponent in vmesnikov je zrasla skupaj s specifikacijami strežniških javanskih zrn. Na tržišču najdemo precej komercialnih, in tudi orodij, razvitih po principu odprte kode, ki omogočajo avtomatsko generiranje komponent. Komercialna orodja za razvoj strežniških javanskih zrn so ponavadi namenjena zgolj razvoju aplikacij za aplikacijski strežnik ta istega proizvajalca, tako da smo pri razvoju omejeni le na te produkte. Najbolj znana orodja za razvoj strežniških javanskih zrn so JBuilder, Visual Age for Java itd. Vedno resnejša alternativa komercialnim produktom so orodja razvita po principu odprtega koda. Teh je na spletu kar precej in imajo pred komercialnimi kar nekaj prednosti. Prva je seveda v smisli financ. Ta orodja so dostopna širši množici razvijalcev in na ta način se pri razvoju orodij upoštevajo izkušnje veče množice razvijalcev, kar pripomore k izboljšanju orodij. Druga prednost pa je vsekakor tudi dejstvo, da je na voljo izvorni kod orodij. Orodja je mogoče razširiti in jim na ta način

dodati manjkajoče funkcionalnosti.
V nadaljevanju poglavja bomo nekaj besed namenili filozofiji razvoja odprto kodnih programskih produktov. Nato sledi še pregled nekaterih tehnologij za razvoj strežniških zrn. Natančneje si bomo ogledali le orodje za generiranje strežniških javanskih zrn XDoclet [?].

3.2 Odprti kod

Produkti razviti po principu odprtega koda so v nasprotju s klasičnim pristopom razvoja programske opreme, kjer se razvite aplikacije skušajo prodati po visoki ceni in po možnosti veliko strankam, brezplačno na voljo vsakomur. Strategija razvoja programske opreme po principu odprtega koda je v zadnjih letih doživela pravi razcvet. Programska oprema z odprtim izvornim kodom postaja vedno večja konkurenca komercialnim programskim produktom in jih je na določenih področjih že prehitela. Zgodba o uspehu in najbolj znan primer produkta z odprtim izvornim kodom je seveda operacijski sistem Linux. Odprto kodna programska oprema pa se vedno bolj uveljavlja tudi na področju namiznih aplikacij. Produkti kot so Open Office, Mozila, Apache spletni strežnik so le najbolj znani primeri uspešnih aplikacij. Poglavitni razlog za uspešnost in razcvet teh programov je predvsem strategija deljenja programskega koda, ki vsakemu posamezniku omogoča uporabo razvitih produktov in možnost sodelovanja pri razvoju le teh. Učinkovitost razvoja se, glede na to, da pri razvoju sodeluje veliko število razvijalcev, ki prihajajo z različnih znanstvenih področij, precej poveča.
Razlika med lastniškimi programi in programi z odprtim izvornim kodom se iz dneva v dan manjša. Oboji imajo kar precej skupnih lastnosti, naštejmo najočitnejše med njimi:

- komercialno dostopno tehnično podporo,
- profesionalno dokumentacijo,
- izdaje novih verzij programske opreme po vnaprej določenem urniku in

- izvedljivo (*binary*) verzijo programske opreme za večino najpopularnejših operacijskih sistemov.

Kljub mnogim skupnim točkam pa lahko programi z odprtim izvornim kodom uporabniku v določenih primerih ponudijo celo več kot njihovi komercialni tekmeci. Najočitnejše prednosti teh produktov so naslednje:

- konfiguracija na nivoju izvornega koda,
- natančen vpogled v delovanje programskih produktov in možnost izboljšave oz. prilagoditev določenih operacij potrebam uporabnika in
- možnost preizkušanja programov v fazi implementacije ter možnost vplivanja na potek razvoja programkega produkta (ideje, pomoč pri implemenaciji).

Vedno pomembnejša lastnost poslovne programske opreme je varnost. Prva misel o programih z odprtim izvornim kodom bi bila, da programski kod, ki je "viden" vsakomur, ne more zadoščati strogim varnostnim merilom. Realnost pa kaže povsem dugo sliko, nekateri izmed najbolj varnih programskih sistemov je zasnovanih prav na programski opremi z odprtim izvornim kodom. Vedno več uporabnikom ne zaupa "zaprtim" lastniškim programom, ki ne dopuščajo vpogleda v izvorni kod, tako se uporabnik ne more prepričati o nivoju varnosti in zaščite. Za razliko od teh programskih produktov je mogoče produkte z odprtim izvornim kodom pregledati. Uporabnik se lahko prepriča o tem, da programska oprema zadostuje varnostim merilom ter izvorni kod ne vsebuje morebitnih šibkih točk kot so trojanski konji ali virusi oz., da se pomembne poslovne informacije morebiti preko spleta ne pošiljajo konkurenci. Večina programskih produktov z odprtim izvornim kodom je zaščitena z licencama [?, ?].

3.3 Orodje *Ant*

Ant [?] je orodje za inkrementalno prevajanje programskih projektov (aplikacij). Precej je podoben svojim predhodnikom, orodjem kot so *make, gnumake, nmake, jam itd.*, a za razliko od naštetih ni zasnovan na interpretacijski lupini operacijskega sistema. Razvit je v programskem jeziku java, tako da je popolnoma neodvisen od operacijskega sistema na katerem ga uporabljamo. Slabost naštetih orodij je predvsem zapletenost uporabe in zelo strikten zapis datotek z opisom navodil za opravila. Orodje Ant za svoje delovanje ne uporablja lupinskih ukazov operacijskega sistema oz. skript, temveč vsa opravila opišemo v eni datoteki (v formatu *XML*). Privzeta datoteka z opisom opravil je *build.xml*, lahko pa opravila zapišemo v datoteko s poljubnim imenom. Prednost zapisa opravil v formatu *xml* je predvsem enostavnost in platformna neodvisnosti zapisa. Filozofija zapisa datoteke z navodili je podobna kot pri že prej omenjenih orodjih, tako da je učenje precej enostavno. Definiramo lahko razne cilje (*target*) in odvisnosti med njimi, vsak cilj vsebuje opise opravil (*task*). Posamezna datoteka vsebuje le en projekt, vsebovati pa mora vsaj en (privzet) cilj, le-ta pa seveda oznake (*tag*) z opravili. Ant vsebuje veliko že definiranih opravil, če pa ta opravila slučajno ne zadoščajo potrebam pri razvoju, jim lahko dodamo nova. Pri razvoju oz. nadgradnji takšnih opravil je razvijalcem na voljo dokumentiran aplikacijski vmesnik in izvorni kod orodja. V tabeli 3.1 je opisano nekaj najpogosteje uporabljenih opravil orodja Ant.

Tabela 3.1: Najpogosteje uporabljena opravila v orodju Ant.

Opravilo	Razlaga
Javac	Prevajanje javanskega izvornega koda.
Jar (War, Ear)	Arhiviranje datotek (jar, war ali ear arhiv).
JavaDoc/JavaDoc2	Generiranje dokumentacije z orodjem JavaDoc.
Copy	Kopiranje datoteke ali celotnih imenikov v drugi imenik.
Exec	Izvajanje sistemskega ukaza.
Mkdir	Ustvarjanje imenika.
Echo	Izpis poljubnega besedila na standarden izhod.

Primer 3.13 Zgled datoteke "build.xml".

```
<?xml version="1.0"?>
<project name="zgled" default="deploy" basedir=".">
<property name="name"      value="zgled" />
<property name="src.dir"   value="zgled" />
<property name="web.dir"   value="web" />
<property name="build.dir" value="build" />
<property name="dist.dir"  value="dist" />
<property name="jboss.home" value="/home/damijan/jboss/jboss-3.0.0" />
<property name="deploy.dir" value="${jboss.home}/server/default/deploy" />
 <target name="init">
   <mkdir dir="${build.dir}" />
   <mkdir dir="${dist.dir}"  />
 </target>
 <target name="compile" depends="init">
   <javac srcdir="${src.dir}" destdir="${build.dir}" includes="*.java" debug="off" />
 </target>
 <target name="deploy" depends="clean, compile">
   <jar destfile="${dist.dir}/${name}-web.war">
     <fileset dir="${web.dir}" />
   </jar>
   <jar destfile="${dist.dir}/${name}.jar">
     <fileset dir="${build.dir}" />
   </jar>
   <ear destfile="${dist.dir}/${name}.ear" appxml="META-INF/application.xml">
     <fileset dir="${dist.dir}" />
   </ear>
   <copy file="${dist.dir}/${name}.ear" todir="${deploy.dir}"/>
 </target>
 <target name="clean">
   <delete dir="${build.dir}" />
   <delete dir="${dist.dir}" />
 </target>
</project>
```

Tabela vsebuje opis le najosnovnejših opravil, ki pa ji Ant premore okoli petdeset. Za podrobnejši pregled orodja si poglejmo datoteko *build.xml* (primer 3.13), ki smo jo uporabili pri prevajanju primera iz poglavja 2.9. Korenska oznaka datoteke *build.xml* mora biti vedno `project`, kateri določimo privzet

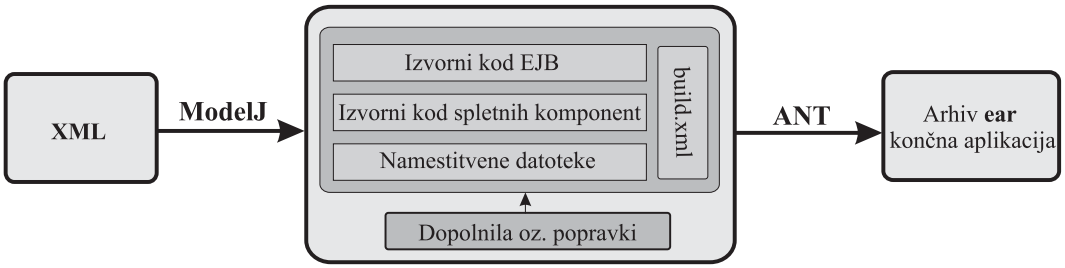
cilj. Prenosljivost izvornega koda med razvijalci je pri razvoju projektov zelo pomembna, zato se je potrebno pri definiranju opravil izogibati absolutnih poti. Da se izognemo absolutnih poti, oznaki `project` navedemo bazni imenik (`basedir`) projekta. Ostale lastnosti si definiramo z oznakami `property`, ki jih lahko kasneje v datoteki uporabimo, tako da ime lastnosti navedemo znotraj "`${`" in "`}`". Vsi cilji (`target`), ki vsebujejo določena opravila so navedena za tem, znotraj oznake `project`. Cilji so lahko med sabo v odvisnosti, npr. cilj *deploy* je v našem primeru odvisen od cilja *init*, kar pomeni, da se mora vedno pred izvedbo tega cilja, izvesti vsa opravila cilja *init*.
Orodje Ant seveda omogoča precej več kot kaže naš preprosti primer. Pri razvoju projektov v programskem jeziku java ga uporablja večina odprto kodnih projektov. Uporabljajo pa ga tudi nekateri komercialni produkti. Orodje smo uporabili pri razvoju praktičnega dela diplomske naloge, uporabljajo pa ga tudi orodja za hiter razvoj spletnih javanskih zrn, ki so opisana v nadaljevanju poglavja.

3.4 Orodje ModelJ

ModelJ [?] je razvojno orodje za generiranje spletnih J2EE aplikacij. Orodje na podlagi podanih specifikacij generira spletno aplikacijo, ki na vmesnem nivoju uporablja strežniška javanska zrna, predstavitveni nivo pa predstavljajo servleti in JSP strani. Specifikacije aplikacije podamo v obliki poslovnega modela aplikacije, ki ga lahko razvijemo v poljubnem programu za načrtovanje. Te specifikacije nato zapišemo (prepišemo) v XML datoteko, ki vsebuje podatke o entitetah in relacijah med njimi. Na podlagi teh specifikacij orodje generira izvorni kod strežniških javanskih zrn, spletno aplikacijo (funkcionalnost aplikacije je omejena zgolj na vpisovanje in in brisanje podatkov) ter namestitvene datoteke tako za strežniška zrna, kot za spletno aplikacijo. Namestitvene datoteke so prilagojene za namestitev na aplikacijski strežnik JBoss. Kot je razvidno iz primera 3.14 je specifikacijska datoteka precej enostavna, učenje uporabe orodja tako ne predstavlja bistvenega dodatnega stroška. Kot je razvidno iz primera so v specifikacijski datoteki navedene le

Primer 3.14 Primer specifikacijske datoteke orodja ModelJ.

```
<generator>
  <application-information>
    <short-name>primer</short-name>
  </application-information>
  <build-information>
    <app-server>jboss</app-server>
    <datasource>DefaultDS</datasource>
    <type-mapping>mySQL</type-mapping>
    <package>com.diploma.applications.primer</package>
  </build-information>
  <bean>
    <name>Artikel</name>
    <attribute>
      <primary-key>yes</primary-key>  <name>id</name>   <type>integer</type>
    </attribute>
    <attribute>
      <name>naziv</name>   <type>string</type>
     </attribute>
  </bean>
  <bean>
    <name>Oseba</name>
    <attribute>
      <primary-key>yes</primary-key>   <name>id</name>   <type>integer</type>
    </attribute>
    <attribute>
      <name>ime</name>   <type>string</type>
    </attribute>
    <attribute>
      <name>priimek</name>   <type>string</type>
    </attribute>
  </bean>
  <relationship>
    <parent-bean-name>Oseba</parent-bean-name>
    <parent-multiplicity>One</parent-multiplicity>
    <child-bean-name>Artikel</child-bean-name>
    <child-multiplicity>Many</child-multiplicity>
    <navigation>unidirectional</navigation>
  </relationship>
</generator>
```



Slika 3.1: Postopek generiranja spletne J2EE aplikacije z orodjem ModelJ.

entitete. Celotna aplikacija pa je seveda dosti obsežnejša. Generiran izvorni kod aplikacije je tako potrebno dopolniti s poslovno logiko, mehanizmi zaščite, transakcije itd. Popravki oz. dopolnitve generiranega koda obsegajo dopolnitve izvornega koda strežniških javanskih zrn, spletnih komponent in seveda namestitvenih datotek. Slabost orodja je predvsem v dejstvu, da so te dopolnitve potrebne ob vsaki spremembi na nivoju specifikacijske datoteke (torej ob ponovni uporabi orodja). Za primer navedimo število vrstic, ki se generirajo iz specifikacijske datoteke, navedene v primeru. Specifikacijska datoteka obsega 59 vrstic. Orodje na podlagi te datoteke generira 3304 vrstic javanske kode (strežniška zrna in spletne komponente), 689 vrstic JSP strani ter 1892 vrstic namestitvenih in konfiguracijskih datotek. Skupno torej 5885 vrstic izvornega koda. Potek razvoja spletne J2EE aplikacije z orodjem ModelJ je prikazan na sliki 3.1.

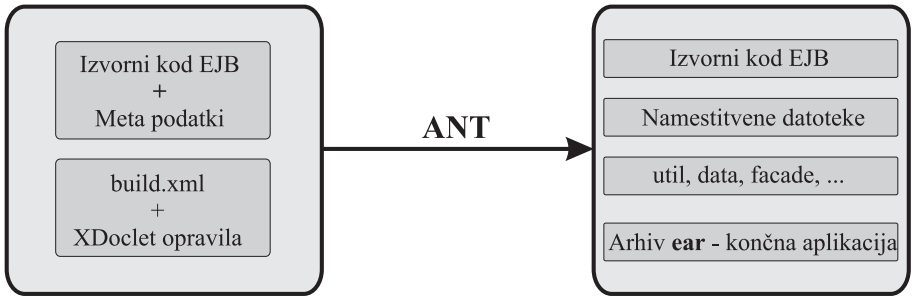
3.5 Orodje XDoclet

XDoclet [**?**] je odprto kodno orodje za generiranje strežniških javanskih zrn z atributno-orientiranim pristopom (*Attribute-Oriented Programming*). S pomočjo orodja se razvoj aplikacij za arhitekturo J2EE precej poenostavi. Razvijalcu strežniških javanskih zrn nudi orodje enoten pogled na komponento (celotna komponenta je predstavljena z eno datoteko – izvorni kod komponente). Ostalo "kramo", kot so vmesniki, namestitvene datoteke, podatkovni razredi,

pomožni razredi itd., generira orodje s pomočjo meta podatkov, ki jih razvijalec doda izvornemu kodu strežniškega zrna. Te meta podatke posredujemo orodju v obliki posebnih `JavaDoc` oznak. Prednosti tega pristopa so naslednje:

- Ni potrebno skrbeti za konsistentno stanje namestitvenih datotek in izvornega koda strežniškega zrna, saj se ob vsaki spremembi izvornega koda, namestitvene datoteke ponovno generirajo.
- Razvijalec J2EE aplikacije ob razvoju komponente spreminja le eno datoteko, za ostalo poskrbi orodje. Na ta način se razvoj pohitri, saj lahko komponento sestavlja tudi do deset datotek, ki jih je potrebno ohranjati v konsistentnem stanju.
- Orodje nudi podporo za večino vodilnih ponudnikov EJB vsebnikov.
- Če orodje ne zadostuje potrebam pri implementaciji projekta, je mogoče orodju na dokaj enostaven način razširiti funkcionalnost. Za tako početnje je razvijalcem na voljo izvorni kod in dokumentiran aplikacijski vmesnik orodja.

Orodje XDoclet je nadgradnja projekta EJBDoclet, ki razvijalcem omogoča le razvoj strežniških javanskih zrn. Orodje XDoclet pa to funkcionalnost razširja na mnoga druga področja. Tako lahko z orodjem na podlagi izvornega koda komponent in pripadajočih atributov generiramo izvorni kod komponente, lokalne in oddaljene vmesnike komponent, podatkovne (*Data* oz. *Value*) razrede, pomožne (*utility*) razrede ter lokalno in oddaljeno fasado (*Local/Remote Session Facade* [**?**]) strežniških zrn in XML namestitvene datoteke. Razvoj slednjih je zelo časovno zahteven, pa tudi pogost vir napak. Z XDoclet-om je mogoče generirati tudi namestitvene datoteke spletnih aplikacij, konfiguracijske datoteke za Struts [**?**] obrazce in akcije, dokumentacijo itd. Za uporabo orodja potrebujemo datoteko z opisom opravil orodja Ant, v katerem definiramo opravila, za katera želimo, da jih xdoclet izvede. Za generiranje preprosto poženemo orodje Ant. Postopek generiranja je prikazan na sliki 3.2.



Slika 3.2: Postopek generiranja strežniških zrn z orodjem XDoclet.

Oznake s katerimi določimo atribute določenega strežniškega zrna sledijo standardnemu `JavaDoc` formatu. Oznake so razdeljene v imenske prostore (*namespace*) in ime oznake (*tagname*), ki temu imenskemu prostoru pripadajo. Vsak imenski prostor nato vsebuje določene lastnosti (*properties*), ki so standardenaga formata (`ime=''vrednost''`). Trenutno orodje vsebuje naslednje imenske prostore:

ejb Standardni EJB atributi (neodvisni od EJB vsebnika).

jboss Atributi strežniškega zrna za nemestitev na JBoss EJB vsebnik.

weblogic Atributi strežniškega zrna za nemestitev na BEA WebLogic EJB vsebnik.

webSphere Atributi strežniškega zrna za nemestitev na IBM WebSphere EJB vsebnik.

orion Atributi strežniškega zrna za nemestitev na Orion EJB vsebnik (Oracle).

castor Atributi za generiranje preslikav za delovno okolje Castor.

mvcsoft Atributi strežniškega zrna za namestitev na upravljalnik obstojnosti MVCSoft EJB2.0.

soap Atributi za generiranje SOAP deskriptorjev.

struts Generiranje `struts-config.xml` datoteke.

web Generiranje `web.xml` datoteke za spletne aplikacije.

jsp Generiranje namestitvene datoteke za knjižnice oznak (*tag library*).

Podrobneje si bomo ogledali le imenska prostora `ejb` in `jboss`, ki sta namenjena razvoju strežniških javanskih zrn in namestitvenih datotek za EJB vsebnik JBoss. Preden pa se lotimo pregleda možnosti, ki nam jih ponuja orodje pri razvoju strežniških zrn si poglejmo preprost primer. Orodje na podlagi izvornega koda zrna (izvorni kod je prikazan na primeru 3.15) generira izvorni kod strežniškega zrna prikazanega na primeru 2.6, lokalna vmesnika s primera 2.4 in 2.5, pomožne rezrede ter lokalno in oddaljeno fasado komponente. Za boljšo predstavo delovanja orodja si poglejmo nekaj statističnih podatkov. Primer obsega 66 vrstic izvornega koda in meta podatkov. XDoclet na podlagi teh specifikacij generira 903 vrstice izvornega koda strežniških javanskih zrn in vmesnikov ter 197 vrstic namestitvenih datotek. Kot je razvidno iz primera, se XDoclet oznake pojavljajo na ravni razreda oz. na ravni metod (včasih pa celo na ravni konstruktorjev in instančnih spremenljivk).
Na nivoju razreda je potrebno orodju podati informacije o tipu komponente, ki jo implementiramo, kateri vmesniki naj se generirajo, katere pomožne razrede želimo (*Util, Facade, Data,* itd.), ... Lahko pa definiramo tudi poizvedovalne metode (*ejbFindByXXX, ejbSelectXXX*). Orodju na tem mestu podamo tudi informacije vezane na EJB vsebnik, ki ga uporabljamo. Seveda lahko podamo informacije za več EJB vsebnikov hkrati. Orodje tedaj generira namestitvene datoteke za več vsebnikov, kar končni aplikaciji omogoča nemoteno migracijo med temi vsebniki. Najpogosteje uporabljene oznake na nivoju razreda so naslednje:

@ejb:bean S to oznako navedemo osnovne informacije o razredu, kot so tip strežniškega zrna (`type`), ime (`name`), JNDI oznaka (`jndi-name`), katere vmesnike želimo (`view-type`) itd. Ta oznaka je edina, ki jo je potrebno navesti, vse ostale so opcijske.

Primer 3.15 Izvorni kod strežniškega zrna in XDoclet oznake.

```
package zgled;
import javax.ejb.*;
import javax.naming.*;
/** @ejb:bean name="OsebaBean" type="CMP" view-type="local" primkey-field="id" cmp-version="2.x"
   * @ejb:finder signature="Collection findAll()" role-name="Diploma" transaction-type="NotSupported"
   * @ejb.util
   * @ejb:facade
   * @ejb.remote-facade
   * @ejb:pk generate="false" class="java.lang.Long"
   * @ejb:finder signature="Collection findByIme(String pIme)" unchecked="true"
              transaction-type="Required" query="select object(o) from OsebaBean o where o.ime = ?1"
   * @jboss:table-name "OsebaBean"
   * @jboss:create-table "true"
   * @jboss:remove-table "true"   **/
public class OsebaBean implements javax.ejb.EntityBean {
    /** @ejb:create-method **/
    public Long ejbCreate(Long pId, String pIme, String pPriimek)throws CreateException {
        setId(pId); setIme(pIme); setPriimek(pPriimek);
        return pId;
    }
    public void ejbPostCreate(Long pId, String pIme, String pPriimek) { }
    /** @ejb:interface-method
      * @ejb:persistence
      * @ejb.pk-field   **/
    public abstract Long getId();
    /** @ejb:interface-method
      * @ejb:persistence  **/
    public abstract void setId(java.lang.Long id );
    /** @ejb:interface-method
       * @ejb:persistence
       * @ejb:ejb.facade-method   **/
    public abstract String getIme();
    /** @ejb:interface-method
        * @ejb:persistence  **/
    public abstract void setIme(String pIme);
   ...
   ...
}
```

@ejb:finder, @ejb:select Definicija poizvedovalne metode.

@ejb:pk Definicija primarnega ključa entitetnega strežniškega zrna.

@ejb:data-object (@ejb:value-object) Generiranje podatkovnih razredov entitetnega strežniškega javanskega zrna. Več o podatkovnih (*Data* oz. *Value*) razredih si lahko bralec prebere v [**?**].

@ejb:util Generiranje pomožnih razredov. Pomožni razredi vsebujejo metode, s katerimi referenciramo strežniška zrna (`getHome` in `getLocalHome`).

@ejb:facade, @ejb:remote-facade Generiranje lokalne in oddaljene fasade. Generirana zrna sledijo vzorcu *Session Facade Pattern*.

@ejb:transaction Definiranje transakcij za vse metode razreda. Oznaka se lahko "povozi" na nivoju metode.

@ejb:permission Določimo vloge (`role-name`), ki imajo pravico izvajati metode razreda.

@jboss:table-name Ime tabele, kjer se hrani stanje entitetnega strežniškega zrna.

@jboss:read-ahead Določimo (*Read Ahead*) strategijo upravlja strežniškega zrna. Oznako lahko pripnemo le entitetnemu CMP strežniškemu zrnu.

Na nivoju metod specificiramo atribute posamezni metodi. Najpogosteje določimo metodi, v katerem vmesniku naj bo "vidna", tip transakcije, obstojnost podatka (za *get/set* metode), relacije do drugih entitetnih zrn in informacije vezane na določen EJB vsebnik. Najpogosteje uporabljene oznake na nivoju metod so naslednje:

@ejb:persistence-field Metodama get/set določimo obstojnost.

@ejb:pk-field Zastavica za primarni ključ entitete.

@ejb:relation Metodi, ki sledita, predstavljata relacijo do drugega entitetnega strežniškega zrna.

@ejb:interface-method Metoda, ki sledi bo vključena v vmesnik(e) zrna.

@ejb:transaction Določimo transakcijsko obnašanje metode (`NotSupported`, `Supports`, `Required`, `RequieresNew`, `Mandatory`).

@ejb:permission Določimo dovoljenja za izvajanje metode.

@ejb:facade-method Označimo metodo za vključitev v fasado zrna.

@jboss:column-name Ime atributa na nivoju podatkovne baze.

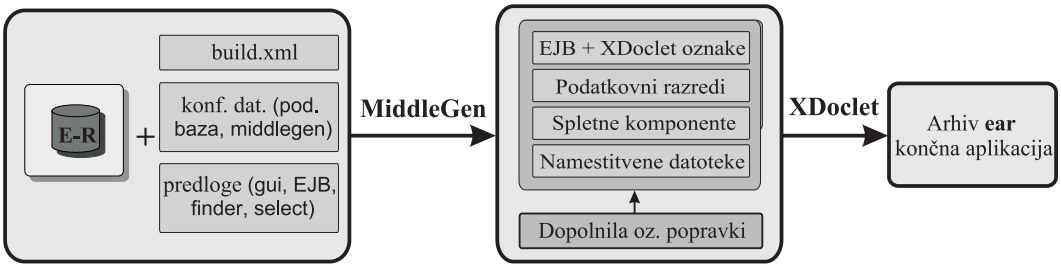
@jboss:relation-mapping Kakšen način preslikave naj se uporabi pri implementaciji relacije do drugega entitetnega zrna.

@jboss:relation-table Ime tabele, ki se naj uporabi pri preslikavi z relacijsko tabelo.

Natančnejše informacije o oznakah si lahko bralec prebere v dokumentaciji orodja [**?**].
Orodje je zelo primerno za implementacijo aplikacij za J2EE arhitekturo, saj razvijalcem prihrani veliko časa. Razvijalec strežniških zrn se z uporabo orodja pri razvoju srežniških zrn lahko v celotni posveti le načrtovanju (XDoclet oznake) in implementaciji poslovne logike. Čeprav je orodje precej bolj obširno od že opisanih orodij je dokaj preprosto, tako da učenje ne traja predolgo in ne prinaša prevelikega dodatnega stroška. Za razliko od že opisanih orodij, dopušča razvijalcem veliko več svobode pri implementaciji. Rezultat generiranja pa je dokončna aplikacija, ki je ni potrebno dopolnjevati. Kljub opisanim prednostim ima orodje kar precej slabosti. Kot prva slabost je vsekakor dobro poznavanje orodja Ant, ki je predpogoj za uporabo orodja. Pomanjkljivosti so opazne tudi pri generiranju fasade, ki ne obsega vseh metod, ki jih določimo v fasadno zrno. Prav tako pa je zelo moteče, da relacijske metode ne vračajo in sprejemajo podatkovnih objektov, tako da je to potrebno za vsako relacijo implementirati ročno. Seveda pa lahko te pomankljivosti odpravimo sami, saj nam je na voljo izvorni kod orodja.

3.6 Orodje MiddleGen

MiddleGen [?] je prosto dostopno orodje za generiranje izvornega koda strežniških javanskih zrn in spletnih J2EE komponent. Generiranje strežniških javanskih zrn in celotne spletne J2EE aplikacije poteka v dveh popolnoma ločenih korakih. V prvem koraku podamo orodju MiddleGen specifikacije, na podlagi katerih orodje generira izvorni kod J2EE komponent (EJB, srvleti, JSP) z dodanimi XDoclet oznakami. Druga faza generiranja končne aplikacije je tako popolnoma neodvisna od orodja MiddleGen in jo opravi orodje XDoclet. Končni produkt je spletna J2EE aplikacija, katere funkcionalnost je omejena na vpis podatkov in na določena osnovna iskanja. Generiranemu izvornemu kodu manjka torej poslovna logika, ki jo potrebno dodati "ročno", po koncu prve faze. Tukaj naletimo na prvi problem. Če želimo dopolniti obstoječi izvorni kod moramo zraven dobrega poznavanja orodja MiddleGen, dobro poznati tudi orodje XDoclet, kar pa lahko v določenih primerih predstavlja precejšen dodaten strošek (učenje uporabe obeh orodij).



Slika 3.3: Postopek generiranja spletne J2EE aplikacije z orodjem MiddleGen.

Orodje je primerno za posodobitev že obstoječih informacijskih sistemov, kakor tudi za razvoj novih. Za generiranje izvornega koda, ni potrebno orodju podati nobenih posebnih specifikacij. Orodju podamo le konfiguracijske datoteke s katerimi orodju omogočimo dostop do podatkovne baze, ki je na podatovnem nivoju razvijajoče aplikacije (v bistvu orodju podamo entitetno relacijski model). Vse ostale nastavitve, ki jih podamo orodju so opcijske. Na podlagi teh informacij, konfiguracijske datoteke v kateri navedemo informacije o EJB vsebniku ter podatkovni bazi in predlog (uporabniški vmesnik,

poizvedovalne metode, del izvornega koda strežniških zrn), generira orodje izvorni kod strežniških javanskih zrn, podatkovnih razredov, servletov in JSP strani. Izvorni kod vseh komponent, razen JSP, je namenjen nadaljni obdelavi s strani orodja XDoclet. Orodje XDoclet nato generira izvorni kod strežniških zrn, vmesnikov, ustrezne namestitvene datoteke ter *ear* arhiv primeren za namestitev na EJB vsebnik. Podrobnejši opis poteka razvoja spletne J2EE aplikacije z orodjem MiddleGen je prikazan na sliki 3.3.

Slika 3.4: Primer E-R modela za orodje MiddleGen.

Na sliki 3.4 je prikazan primer E-R modela, ki ga podamo orodju kot vhodne specifikacije. Na podlagi prikazanega primera generira orodje 744 vrstic izvornega koda strežniških zrn, 441 vrstic izvornega koda podatkovnih objektov, 2647 vrstic izvornega koda spletnih komponent in 1487 vrstic namestitvenih in konfiguracijskih datotek.

Orodje je primerno za razvoj manjših spletnih J2EE aplikacij, ki morajo biti razvite v zelo kratkem času. Za razvoj preprostih aplikacij, ki bistveno ne spreminjajo osnove, ki jo generira orodje, je orodje tudi enostavno za uporabo. Če želimo na primer narediti spletno aplikacijo z osnovno funkcionalnostjo lahko kot šablono vzamemo primer, ki je priložen orodju. Spremenimo le nekaj podatkov, ki se navezujejo na podatkovno bazo, zaženemo orodje in že imamo spletno aplikacijo. Če pa želimo z orodjem razvijati večje aplikacije, ki bistveno razširjajo osnovno funkcionalnost, kar jo resne aplikacije vsekakor, potem je za razvoj potrebno veliko znanja in izkušenj. Za resen razvoj aplikacij z orodjem MiddleGen je potrebno dobro poznati orodje samo. Predvsem način podajanja predlog za uporabniški vmesnik in izvorno kodo strežniških zrn. Ker

večino izvornega koda, ki predstavlja poslovno logiko aplikacije, podamo po zaključku prve faze je potrebno dobro poznati tudi orodje XDoclet in ogrodje za razvoj spletnih aplikacij Struts. Kot slabost orodja je potrebno omeniti še zelo slabo dokumentacijo, kar pa razvijalci obljubljajo, da bodo z naslednjo verzijo popravili.

3.7 Prednosti in slabosti

Prednost opisanih orodij je predvsem pohitritev razvoja J2EE aplikacij. Seveda pa to ni edina prednost uporabe teh orodij. Orodja so razvijalcem v veliko pomoč predvsem v fazi nameščanja oz. pisanja namestitvenih datotek, ki so pogost vir napak pri razvoju J2EE aplkikacij. Končni produkti so pri vseh opisanih orodjih praktično enaki (celotna oz. delna J2EE aplikacija), a se pristop pri načrtovanju oz. implementaciji med orodji kar precej razlikuje. Direktna primerjava orodij torej ni mogoča, saj je učinkovitost posameznega orodja odvisna od tipa razvijajoče aplikacije oz. načina načrtovanja, ki je bližje razvijalcu aplikacije. Od vseh orodij je prav gotovo najzahtevnejše XDoclet, ki zahteva od razvijalca visok nivo poznavanja J2EE tehnologij in orodja samega. Prav tako pa orodje zahteva največ vloženega truda, da pridemo do željenega rezultata. Slabost ostalih dveh orodij (MiddleGen in ModelJ) je predvem potreba po dopolnjevanju generirane aplikacije. Kar po eni strani zahteva visok nivo poznavanja orodja (generiranega izvornaga koda), po drugi pa dodatno delo ob vsaki spremembi na nivoju specifikacij aplikacije.
Opisana orodja so produkt odprto kodnih projektov (*Open Source Projects*) in so trenutno še v razvoju. V prihodnosti lahko pričakujemo (obljube avtorjev) odpravo večine naštetih problemov.

Experience is that marvelous thing that enables you recognize a mistake when you make it again.

– F. P. Jones –

4 Implementacija spletnega informacijskega sistema za vodenje izposoje

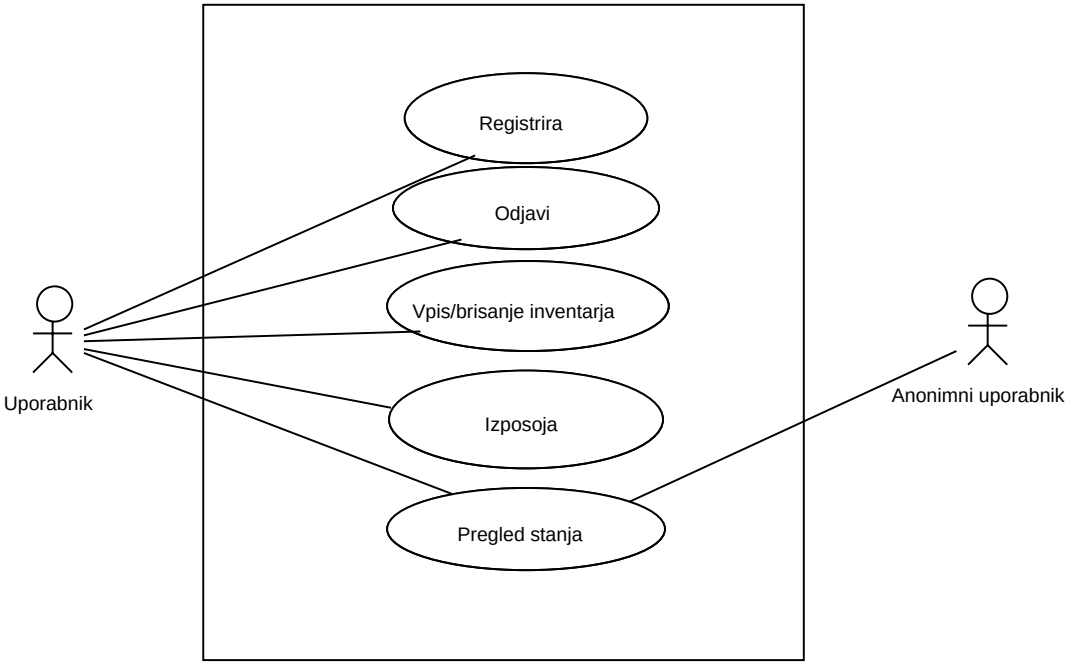
V poglavju je opisana implementacija praktičnega dela diplomske naloge, spletne J2EE aplikacije za vodenje izposoje in arhiv literature v digitalni obliki. Opisani so uporabljeni pristopi, tehnologije in arhitekturni model, ki smo ga uporabili pri implementaciji. Uvodoma so razložene specifikacije in omejitve aplikacije. Naslednja podpoglavja pa so osredotočena na opis implementacije posameznih nivojev spletne aplikacije.

4.1 Uvod

Do sedaj smo predstavljali tehnologije in orodja za pomoč pri načrtovanju in implementaciji programske opreme. Na predstavljenih primerih le s težavo ponazorimo prednosti in izrazno moč celotnega ogrodja. Ponavadi v praksi naletimo na različne ovire in probleme, ki niso samo tehnološke, ampak tudi filozofske narave. V ta namen si oglejmo razvoj celotnega produkta, od specifikacij do prvega končnega izdelka, na konkretnem primeru.
V Laboratoriju za računalniške arhitekture in jezike se že dalj časa soočamo s problemom vodenja evidence izposoje revij, knjig in ostalega inventarja. Za potrebe evidentiranja izposoje in za praktično demonstracijo opisanih tehnologij smo razvili spletni informacijski sistem za vodenje evidence izposojenega gradiva.

4.2 Specifikacija zahtev

V fazi specificiranja zahtev poskušamo pridobiti čimveč informacij o delovanju in funkcionalnosti, ki naj bi jo podpirala aplikacija. Zbiranje zahtev od naročnika lahko poteka na več načinov. Najbolj običajen način zbiranja uporabniških zahtev je pogovor z naročnikom. Le redko se zgodi, da bi od naročnika dobili natančne specifikacije, primerne za nadaljnje korake razvoja aplikacije. Pri pridobivanju informacij o delovanju aplikacije, oz. specifikacij lahko uporabimo različne pristope, formalne, kakor tudi neformalne. Med formalne tehnike zbiranja specifikacij spadajo diagrami uporabe (*Use-case diagrams*), ki so ena od diagramskih tehnik UML (*Unified Modeling Language*) [**?**]. Z diagrami uporabe določimo možne scenarije uporabe aplikacije. Na sliki 4.1 je prikazan primer diagrama uporabe, ki predstavlja uporabnikov pogled na delovanje spletnega informacijskega sistema za vodenje izpososoje. Prikazan diagram uporabe predstavlja splošen uporabnikov pogled. Seveda je potrebno diagrame uporabe razširiti na vse možne scenarije uporabe aplikacije.
Kot je razvidno iz diagrama uporabe, prikazanega na sliki 4.1 so bile zahteve s strani naročnika (mentor) naslednje:



Slika 4.1: Diagram uporabe za spletni informacijski sistem za vodenje izposoje.

- avtorizacija uporabnika,
- vpisovanje inventarja v podatkovno bazo,
- brisanje inventarja iz podatkovne baze,
- iskanje inventarja po raznih ključih (lastnik, tip inventarja, opis, ključne besede, ...),
- vodenje izposoje,
- razni izpisi (izposojeno gradivo, prepozno vrnjeno gradivo, ...),

Seveda je potrebno upoštevati dejstvo, da naročnik nikoli ne poda popolnih specifikacij, tako da je potrebno aplikacijo v fazi implementacije, pa tudi kasneje, spreminjati oz. prijagajati uporabnikovim zahtevam. Temu primerno je potrebno zastaviti arhitekturni model aplikacije.

4.3 Arhitekturni model spletne aplikacije

Izbira arhitekturnega modela aplikacije in tehnologij, ki jih uporabimo pri razvoju je ponavadi ključno vprašanje in vpliva na celoten potek razvoja. Izbira pravilnega pristopa lahko bistveno pripomore k lažji implementacji in kakovostnejšem produktu. Pri razvoju arhitekturnega modela aplikacije smo poskušali zadostiti naslednjim kriterijem:

- Modularnost aplikacije.

  Sama arhitektura J2EE je zanovana na komponentnem modelu. To dejstvo smo poskušali čimbolje izkoristiti in celotni sistem zasnovati tako, da so posamezne kompomente čimbolj neodvisne med sabo. Na ta način smo zagotovili visok nivo modularnosti aplikacije in zmožnosti nadgradnje oz. vgradnje morebitnih sprememb v osnovno funkcionalnost razvite aplikacije.

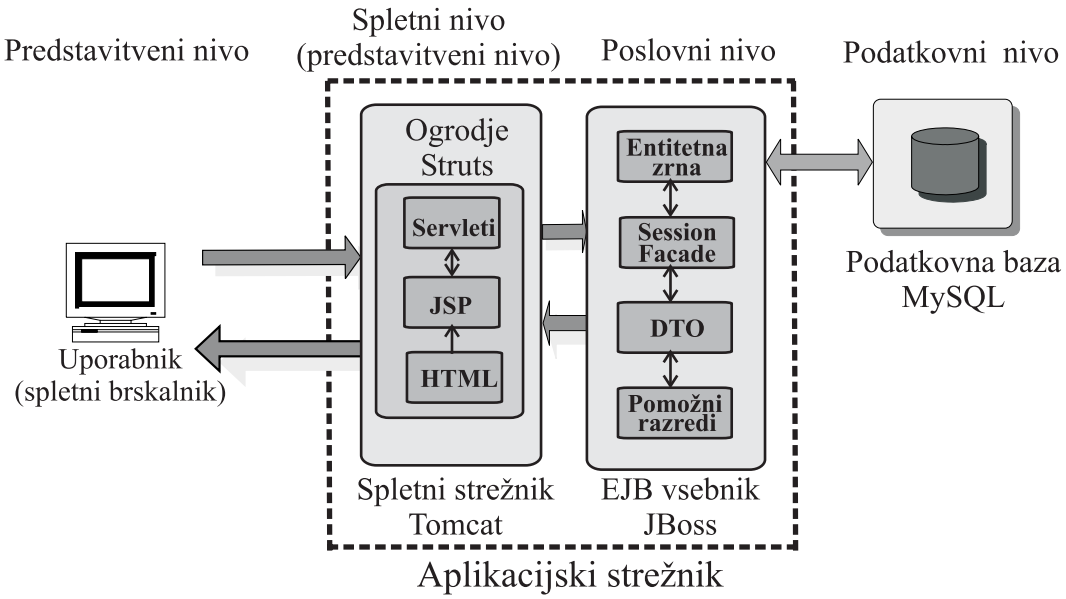
- Ponovna uporabnost razvitih komponent.

  Pri zasnovi komponent celotne aplikacije, ne samo strežniških javanskih zrn, smo težili k možnosti ponovne uporabe že razvitih komponent. Posamezne komponente, razvitega informacijskega sistema, lahko tako uporabimo v drugih aplikacijah.

- Uporaba primernih vzorcev za načrtovanje J2EE aplikacij.

  Pri zasnovi celotne aplikacije in implementaciji posameznih komponent smo sledili raznim, v strokovnem svetu dobro poznanim, vzorcem za načrtovanje J2EE aplikacij. Na ta način smo v našo rešitev vključili znanje in izkušnje mnogih razvijalcev.

Arhitekturni model razvite aplikacije je zelo podoben klasičnemu J2EE modelu, značilnem za vse J2EE aplikacije in je prikazan na sliki 4.2.
Arhitekturni model je sestavljen iz štirih nivojev. Na najnižjem, podatkovnem nivoju, se nahaja podatkovna baza, do katere imajo v našem primeru dostop le entitetna strežniška zrna poslovnega nivoja. Na poslovnem nivoju se izvaja



Slika 4.2: Arhitekturni model aplikacije Spletna izposojevalnica.

celotna poslovna logika, predstavljajo pa ga strežniška javanska zrna ter razne pomožne komponente kot so podatkovni objekti, lokalna in oddaljena fasada itd. Spletni nivo predstavljajo servleti, JSP strani in statične HTML strani. Pri razvoju spletnega nivoja smo uporabili zelo znan pristop za implementacijo uporabniških vmesnikov *Model-View-Controller* (MVC). Pri razvoju spletnega uporabniškega vmesnika smo v našem primeru uporabili odprto kodno implementacijo ogrodja za razvoj spletnih uporabniških vmesnikov, Struts [**?**].
Seveda ne smemo pazabiti na EJB in Servlet (JSP) vsebnik, ki upravljata izvajanje aplikacije. V našem primeru smo uporabili EJB vsebnik JBoss [**?**] z vgrajenim spletnim strežnikom Tomcat [**?**].
Predstavljena arhitektura zadošča že omenjenim kriterijem in jih na določenih mestih celo presega. Kot je razvidno iz slike, ki prikazuje arhitekturo aplikacije, so posamezni nivoji aplikacije skoraj povsem neodvisni od preostalega dela. Sprememba aplikacije na enem nivoju v večini primerov ne vpliva na preostale dele. Vsa orodja, ki smo jih uporabili pri razvoju aplikacije, so ravita v programskem jeziku java. Tako lahko celotno razvojno okolje aplikacije, kakor tudi izvajalno, brez dodatnih posegov prenesemo na drugi operacijski sistem.

4.4 Podatkovni nivo

Arhitekturni model in način implementacije aplikacije omogoča neodvisnost od uporabljene podatkovne baze. Podatkovno bazo bi lahko v našem primeru kadarkoli zamenjali. V našem primeru smo za hranjene podatkov uporabili podatkovno bazo MySql [?], ki postaja v zadnjem času najpogosteje uporabljena odprto kodna podatkovna baza. Odlike podatkovne baze MySql so predvsem hitrost, zanesljivost in preprostost uporabe in administriranja. To so ključni razlogi, da smo se odločili to podatkovno bazo uporabiti pri razvoju informacijskega sistema. MySql je relacijska podatkovna baza in za dostop do podatkov uporablja SQL (*Structured Query Language)*, ki je standard pri skoraj vseh ponudnikih podatkovnih baz.

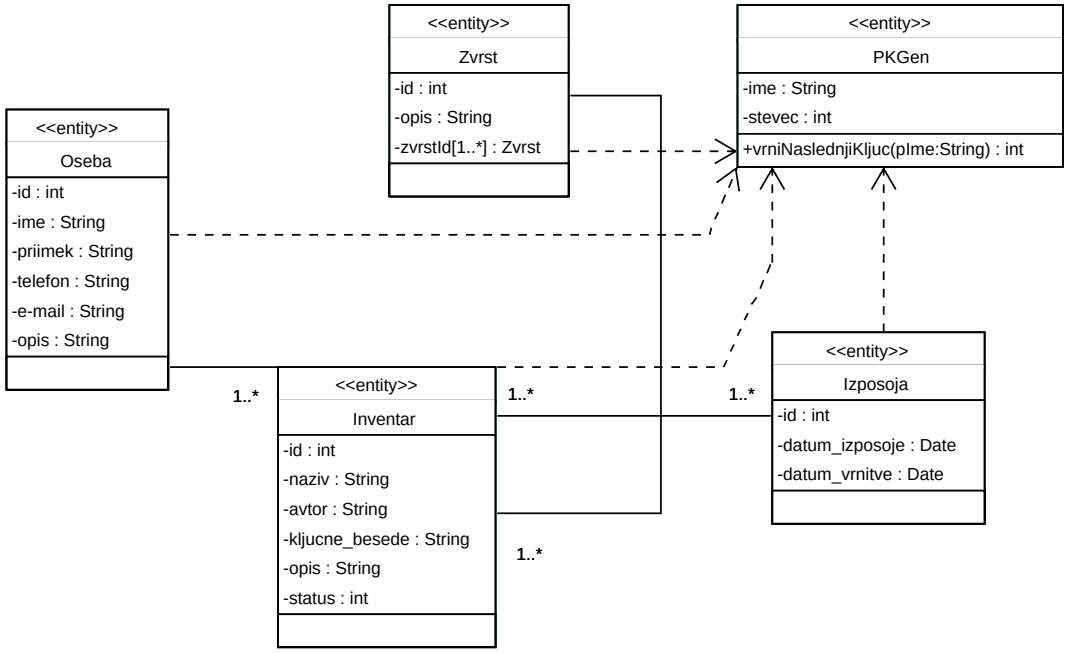
4.5 Poslovni nivo

Poslovni nivo je jedro vsake aplikacije. Implementacija poslovnega nivoja predstavlja ponavadi najtežji del implementacije celotne aplikacije. Razredi vmesnega nivoja vsebujejo izvorni kod poslovne logike, dostopov do podatkovne baze, mehanizmov zaščite, upravljanja transakcij itd. V našem primeru smo se kodiranju večine naštetih storitev izognili. Z uporabo J2EE arhitekture smo upravljanje večine teh storitev prenesli na EJB vsebnik ter se posvetili zgolj razvoju poslovne logike informacijskega sistema.
Poslovni nivo je v našem primeru sestavljen iz strežniških javanskih zrn, pomožnih razredov (podatkovni razredi, *Session Facade*, oddaljena fasada) ter razreda za generiranje unikatnih primarnih ključev posameznih entitetnih zrn.

4.5.1 Implementacija

Glede na to, da smo pri implementaciji poslovnega nivoja uporabili orodje XDoclet, smo se lahko osredotočili le na razvoj poslovne logike. Implementacija celotnega poslovnega nivoja informacijskega sistema zajema tako le razvoj strežniških javanskih zrn, načrtovanje vmesnikov in XDoclet oznak. Kot smo že večkrat poudarili, je izvorni kod strežniških zrn kombinacija javanskega koda

in XDoclet oznak. Javanski kod zrna predstavlja poslovno logiko, medtem ko XDoclet oznake služijo za generiranje vmesnikov, pomožnih razredov ter namestitvenih datotek z opisom storitev, ki so v domeni EJB vsebnika. Te storitve so naslednje: zaščita, transakcijska podpora, obstojnost entitetnih strežniških zrn ter upravljanje relacij med entitetnimi strežniškimi zrni. Prav tako z XDoclet oznakami definiramo tudi poizvedovalne metode (`findByXXX`, `ejbSelectXXX`).
Strežniška javanska zrna spletnega informacijskega sistema ter povezave med njimi so prikazane na sliki 4.3.



Slika 4.3: Razredni diagram informacijskega sistema.

Kot je razvidno iz slike, predstavlja celotni poslovni nivo zgolj pet entitetnih strežniških zrn. Pozoren bralec je na sliki sigurno opazil razred `PKGen`, ki nekako ne spada v okvir ostalih. To zrno je namenjeno za dodeljevanje in shrambo primarnih ključev ostalih entitetnih zrn. Zrno ima dva atributa `ime` (naziv zrna) in `stevec` (trenutno zadnji dodeljen primarni ključ zrna `ime`). V

zrnu je implementirana metoda `getNextCounter`, ki vrne unikatni primarni ključ in ažurira stanje atributa `stevec`. Ta metoda se kliče vedno ob ustvarjanju novega entitetnega zrna (metoda `create`). Na ta način je dodeljevanje primarnih ključev povsem transparentno, izognemo pa se tudi morebitnim napakam pri dodeljevanju (ponavljanje istega ključa).

Priloga A vsebuje izpis izvornega koda entitetnega strežiškega zrna `OsebaBean`. Razred upravlja s podatki o osebi (uporabniku informacijskega sistema). V tabeli 4.1 so prikazani podatki o datotekah in številu vrstic datotek, ki jih orodje XDoclet generira iz izvornega koda zrna `OsebaBean`.

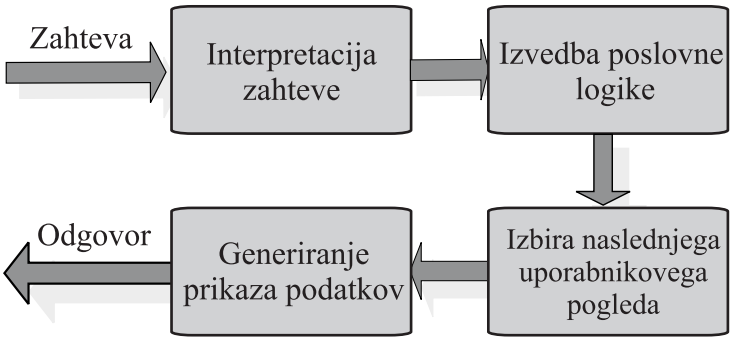
Tabela 4.1: Z orodjem XDoclet generirane datoteke.

Datoteka	Št. vrstic	Opomba
OsebaBean.java	188	izvorni kod
ejb/OsebaBeanCMP.java	112	CMP zrno
ejb/OsebaBeanFacadeSession.java	39	*Session Facade*
interfaces/OsebaBeanLocalHome.java	37	vmesnik *LocalHome*
interfaces/OsebaBeanLocal.java	53	vmesnik *Local*
interfaces/OsebaBeanHome.java	37	vmesnik *Home*
interfaces/OsebaBean.java	59	vmesnik *Remote*
OsebaBeanUtil.java	156	pomožni razred
OsebaBeanData.java	200	podatkovni razred
OsebaBeanFacadeRemote.java	118	oddaljena fasada
OsebaBeanFacadeLocalHome.java	23	vmesnik *LocalHome*
OsebaBeanFacadeLocal.java	32	vmesnik *Local*
OsebaBeanFacadeHome.java	23	vmesnik *Home*
OsebaBeanFacade.java	39	vmesnik *Remote*
OsebaBeanFacadeUtil.java	157	pomožni razred
web/OsebaBeanForm.java	131	Struts obrazec

4.6 Spletni nivo

Komponente spletnega nivoja so most med uporabnikom in poslovnim nivojem informacijskega sistema. Predstavljajo ga spletne komponente: servleti,

JSP in statične strani za predstavitev podatkov (HTML). Spletni nivo spletnega informacijskega nivoja je v bistvu predstavitveni nivo. Na tem nivoju se specificira uporabniški vmesnik, ki bo na voljo uporabniku. Načrtovanje in implementacija uporabniškega vmesnika je zelo pomemben korak pri implementaciji celotnega informacijskega sistema. Ponavadi se načrtovanje uporabniškega vmesnika omeji le na izgled aplikacije, kar pa je lahko velika napaka. Pri razvoju uporabniškega vmesnika smo imeli v mislih predvsem modularnost, zmožnost nadgradnje oz. možnost spreminjanja funcionalnost, brez večjih posegov v samo arhitekturo uporabniškega vmesnika in celotnega informacijskega sistema.
Spletni nivo J2EE aplikacije streže vsem HTTP zahtevam s strani uporabnikov. Naloge spletnega nivoja pri strežbi uporabnikovih zahtev povzema slika 4.4. Vsako uporabnikovo zahtevo je potrebno ustrezno interpretirati in glede na



Slika 4.4: Strežba uporabnikove zahteve v spletni aplikaciji.

tip uporabnikove zahteve izvesti določene metode poslovne logike. Na podlagi zahteve je potrebno izbrati nasledji uporabnikov pogled ter le tega ustrezno prikazati uporabniku.

4.6.1 MVC

Model–View–Controller (MVC) je arhitekturni model za načrtovanje in implementacijo uporabniških vmesnikov interaktivnih aplikacij. Uporabljen je v številnih novejših interaktivnih aplikacijah. Uporabniški vmesnik aplikacije je razdeljen na tri nivoje, katerim dodelimo specifične naloge:

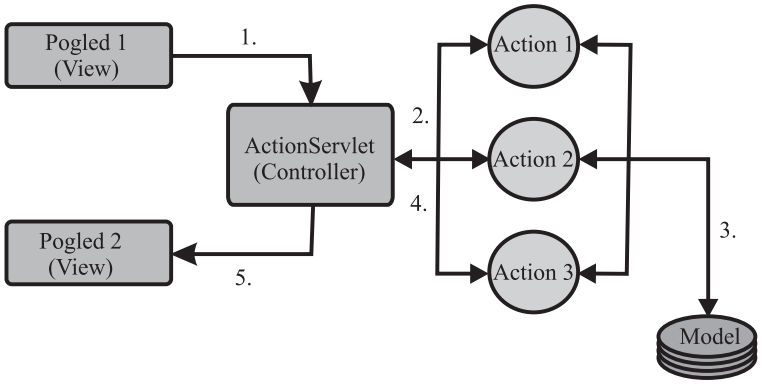
- 1. nivo (**Model**) predstavlja podatke in poslovno logiko predstavitvenega nivoja. Ta nivo komunicira s poslovno logiko aplikacije in služi kot most med predstavitvenim nivojem in nivojem poslovne logike.
- 2. nivo (**View**) dostopa do podatkov prvega nivoja in je odgovoren za ustrezen prikaz le teh.
- 3. nivo (**Controller**) definira obnašanje aplikacije. Uporabnikove zahteve razpošlje ustreznim komponentam in izbere naslednji uporabnikov pogled.

Razdelitev odgovornosti na tri nivoje zmanjša podvajanje izvornega koda (na predstavitvenem in poslovnem nivoju) in olajša vzdrževanje ter nadgradnjo uporabniškega vmesnika. Prav tako zagotavlja neodvisnost med poslovnim nivojem in nivojem predstavitve podatkov.

4.6.2 Ogrodje Struts

Ogrodje Struts [**?**, **?**] je odptro kodna implementacija arhitekture MVC za razvoj spletnih aplikacij. Ogrodje Struts je uporabljeno v številnih odprto kodnih projektih kakor tudi v komercialnih produktuh. Arhitektura MVC je v ogrodju Struts implementirana v celoti na strani strežnika, z uporabo JSP strani in servletov. Strežba uporabnikovega zahteve (slika 4.5) poteka v naslednjih korakih:

1. Uporabnik pošlje zahtevo.
2. Zahtevo prestreže `ActionServlet`, ki se odzove kot *Controller*. Na podlagi naslova od koder je prispela zahteva in konfiguracijske datoteke `struts-config.xml` pošlje zahtevo ustreznemu razredu za izvedbo poslovne logike.
3. Izvedba poslovne logike.



Slika 4.5: Strežba uporabnikove zahteve v ogrodju Struts.

4. Po izvedbi poslovne logike se rezultati izvajanja skupaj s ključem, ki predstavlja stanje izvedbe (uspešno, neuspešno, napaka, ...), posredujejo servletu `ActionServlet`. Ta, na podlagi ključa, rezultate posreduje ustreznim komponentam za prikaz podatkov.

5. Zahteva se zaključi, ko `ActionServlet` posreduje rezultate komponentam za prikaz podatkov.

4.6.3 Implementacija spletnega nivoja

Vsakemu entitetnemu strežniškemu zrnu aplikacije smo v fazi generiranja strežniških javanskih zrn, z orodjem XDoclet, zgenerirali tudi zazred `ActionForm`. Ta razred predstavlja podatke obrazca, ki jih s spletnim brskalnikom vnese uporabnik. Te razrede smo v fazi implementacije uporabniškega vmesnika razširili z osnovno poslovno logiko, ki obsega le preverjanje pravilnosti vpisanih podatkov. `Action` razrede, ki izvajajo poslovno logiko uporabniškega vmesnika, neposredno tudi poslovno logiko celotne aplikacije, ni mogoče generirati. Posamezen spletni obrazec je lahko namreč povezan z več `Action` razredi. Kateri se bo v danem trenutku uporabil je odvisno od uporabnikove zahteve. `Action` in `ActionForm` razrede smo dopolnili z XDoclet oznakami, na podlagi katerih orodje generira datoteko `struts-config.xml`. V datoteki so opisane povezave med razredi obrazcev (`ActionForm`) in razredi z akcijami (`Action`).

Vidni del uporabniškega vmesnika (predstavitev v spletnem brskalniku) predstavljajo JSP strani. Implementcija JSP strani z ogrodjem Struts omogoča ločevanje poslovne logike in grafičnega oblikovanja uporabniškega vmesnika. JSP strani tako vsebujejo le majhen odstotek javanskega koda. Grafično oblikovanje uporabniškega vmesnika lahko implementiramo v poljubnem orodju za razvoj in oblikovanje spletnih strani.
Primer implementacije opisanega postopka razvoja spletnega uporabniškega vmesnika je prikazan v prilogi B. Primer prikazuje razvoj spletnega uporabniškega vmesnika za vpis podatkov novega uporabnika informacijskega sistema.

4.7 Predstavitveni nivo

Predstavitveni nivo (odjemalec) informacijskega sistema predstavlja spletni brskalnik, ki ga pri uporabi informacijskega sistema uporabljajo uporabniki. Predstavitveni nivo je v našem primeru namenjen le vnosu podatkov in ustreznemu prikazu le teh. Celotni uporabniški vmesnik informacijskega sistema je implementiran na spletnem nivoju.

Life is like a box of chocolates, you never know what you're gonna get.

– Forrest Gump –

5 Zaključek

Pred začetkom razvoja aplikacij je potrebno sprejeti nekaj zelo pomembnih odločitev, ki lahko odločilno vplivajo na potek razvoja. Te odločitve so lahko odločilnega pomena za uspeh oz. neuspeh celotnega projekta. Izmed teh odločitev sta zelo pomembni tudi odločitvi o tehnologiji, v kateri bo aplikacija razvita in nenazadnje, katero orodje bomo pri implementaciji uporabili. Arhitektura J2EE postaja od svojega krsta leta 1998, vedno bolj pogosta izbira pri implementaciji večjih aplikacij. Arhitektura J2EE je komponentno zasnovana in kot takšna ponuja razvijalcem model, ki omogoča razvoj aplikacij sestavljenih

iz, v večini primerov, ponovno uborabnih komponent. Arhitektura je namenjena razvoju celotnih aplikacij s poudarkom na razvoju vmesnega (poslovnega) nivoja. Komponente J2EE aplikacije se izvajajo na vsebniku, ki komponentam, poleg upravljanja življenjskega cikla, zagotavlja tudi razne storitve, kot so: transakcijska podpora, upravljanje stanja komponent, beleženje dnevnikov itd. Te storitve so pri uporabi klasičnega pristopa pri razvoju aplikacije prepuščene razvijalcu in so ponavadi prepletene z rutinami poslovne logike, kar vsekakor poveča kompleksnost razvoja. Razvoj aplikacij za arhitekturo J2EE je kljub naštetim prednostim še vedno zelo zahteven, zato je izbira orodja za razvoj aplikacije zelo pomembna. Komercialnih produktov za razvoj J2EE aplikacij je na tržišču kar nekaj. Slabost le teh je zelo visoka cena in omejenost na J2EE vsebnik ta istega proizvajalca. Razlika med komercialnimi in prosto dostopnimi orodji, razvitimi po principu odprtega koda, je vedno manjša. Za razliko od komercialnih orodij so ta orodja dosti bolj cenovno ugodna (zastonj), prav tako pa ponavadi niso omejena le na enega ponudnika J2EE vsebnika.
Diplomsko delo podaja natančen opis arhitekture J2EE in pripadajočih tehnologij. Največ pozornosti je namenjeno strežniškim javanskim zrnom (EJB), ki so vsekakor osrednja tehnologija J2EE arhitekture. Glede na to, da je razvoj večjih J2EE aplikacij zelo kompleksno opravilo, smo preučili orodja za hiter razvoj le teh. Pri tem smo se omejili na prosto dostopna odprto kodna orodja. V diplomskem delu smo opisali tri taka orodja, ki zahtevajo povsem različen pristop pri razvoju aplikacij. Praktičen del diplomske naloge je implementiran z orodjem XDoclet in predstavlja praktičen primer razvoja celotne J2EE aplikacije.

A

Implementacija poslovnega nivoja

Poglavje prikazuje izvorni kod entitetnega strežniškega zrna `OsebaBean`*. Izvornamu kodu zrna so dodane XDoclet oznake za generiranje vmesnikov, pomožnih razredov, podatkovnih razredov itd.*

OsebaBean.java

```
package diploma.ejb;

import diploma.interfaces.*;
import diploma.util.*;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.ejb.*;
import java.util.*;

/**
* Podatki o osebi (uporabniku), ki uporablja spletno izposojevalnico.
* Razred je namenjen za nadaljno obdelavo s strani orodja XDoclet.
*
* XDoclet oznake.
*
* @ejb:bean name="OsebaBean" type="CMP" view-type="both" primkey-field="id" cmp-version="2.x"
* @ejb:finder signature="Collection findAll()" role-name="Diploma" transaction-type="NotSupported"
* @ejb:transaction type="Required"
* @ejb.util
*    generate="physical"
* @ejb:facade
* @ejb.remote-facade
* @ejb:data-object
* @ejb:pk generate="false" class="java.lang.Long"
* @ejb:finder
*      signature="Collection findByIme(java.lang.String pIme)" unchecked="true"
*      transaction-type="Required"
*      query="select object(o) from OsebaBean o where o.ime = ?1"
* @ejb:finder signature="Collection findByImePriimek(java.lang.String pIme, java.lang.String pPriimek)"
*      unchecked="true" transaction-type="Required"
*      query="select object(o) from OsebaBean o where o.ime = ?1 or o.priimek= ?2"
* @ejb:finder signature="Collection findByEmail(java.lang.String pEmail)"
*      unchecked="true" transaction-type="Required"
*      query="select object(o) from OsebaBean o where o.email = ?1"
* @struts:form name="" include-all="true"
* @jboss:table-name "diplOsebaBean"
* @jboss:create-table "true"
* @jboss:remove-table "true"
* @jboss:tuned-updates "true"
* @jboss:read-only "false"
*
*/
public  abstract  class OsebaBean  implements EntityBean {

    /**
    * @ejb:create-method
    */
    public Long ejbCreate(String pName, String pPriimek, String pTel, String pEmail, String pOpis)
                                                                    throws CreateException {
        // Primarni ključ se vedno generira!
       setId(CKeyGen.getNextKey("OsebaBean"));
       setIme(pName);
       setPriimek(pPriimek);
       setTel(pTel);
       setEmail(pEmail);
```

```
        setOpis(pOpis);
         return getId();
    }

     public  void ejbPostCreate(String pName, String pPriimek, String pTel, String pEmail, String pOpis)
    {}

     /**
     * Vsaki metodi dodamo XDoclet oznake.
     *
     * @ejb:interface-method
     * @ejb:persistence
     * @ejb.pk-field
     */
     public  abstract Long getId();

     /**
     * @ejb:interface-method
     * @ejb:persistence
     */
     public  abstract  void setId(Long pId);


     /**
     * @ejb:interface-method
     * @ejb:persistence
     * @ejb:ejb.facade-method
     */
     public  abstract String getIme();

     /**
     * @ejb:interface-method
     * @ejb:persistence
     * @ejb:ejb.facade-method
     */
     public  abstract  void setIme(String pIme);

     /**
     * @ejb:interface-method
     * @ejb:persistence
     * @ejb:ejb.facade-method
     */
     public  abstract String getPriimek();

     /**
     * @ejb:interface-method
     * @ejb:ejb.facade-method
     */
     public  abstract  void setPriimek(String pIme);

     /**
     * @ejb:interface-method
     * @ejb:persistence
     * @ejb:ejb.facade-method
     */
     public  abstract String getTel();

     /**
     * @ejb:interface-method
     * @ejb:ejb.facade-method
```

```
*/
public abstract void setTel(String pTel);

/**
* @ejb:interface-method
* @ejb:persistence
* @ejb:ejb.facade-method
*/
public abstract String getEmail();

/**
* @ejb:interface-method
* @ejb:ejb.facade-method
*/
public abstract void setEmail(String pEmail);

/**
* @ejb:interface-method
* @ejb:persistence
* @ejb:ejb.facade-method
*/
public abstract String getOpis();

/**
* @ejb:interface-method
* @ejb:ejb.facade-method
*/
public abstract void setOpis(String pOpis);

/**
* Dostop do podatkovnega objekta.
*
* @ejb:interface-method
*/
public abstract diploma.interfaces.OsebaBeanData getData();

/**
* @ejb:interface-method
*/
public abstract void setData(diploma.interfaces.OsebaBeanData pOsebaBeanData);

/**
* @return Oprema, ki je v lasti osebe.
*
* @ejb:interface-method view-type="local"
* @ejb:relation
*      name="Oseba_Oprema_1_to_n"
*      role-name="Oseba_Oprema"
* @jboss.relation-mapping style="foreign-key"
* @ejb:ejb.facade-method
*/
public abstract Set getOprema();

/**
* Dodajanje opreme.
*
* @ejb:interface-method  view-type="local"
* @ejb:ejb.facade-method
*/
public abstract void setOprema(Set pOprema);
```

```
    /**
    * @return Vse izposoje, ki jih je opravila oseba.
    *
    * @ejb:interface-method view-type="local"
    * @ejb:relation
    *      name="Izposoja_Oseba_1_to_n"
    *      role-name="Oseba_Izposoja"
    * @jboss.relation-mapping style="foreign-key"
    * @ejb:ejb.facade-method
    */
    public  abstract Set getIzposoja();

    /**
    *
    * @ejb:interface-method  view-type="local"
    * @ejb:ejb.facade-method
    */
    public  abstract  void setIzposoja(Set pIzposoja);
}
```

B

Implementacija spletnega uporabniškega vmesnika

Poglavje vsebuje izvorni kod spletnega uporabniškega vmesnika informacijskega sistema. Poglavje ne zajema izvornega koda celotnega spletnega uporabniškega vmesnika, ampak podaja le del, namenjen vpisu podatkov o novem uporabniku. Izpisi izvornega koda si sledijo v vrstnem redu, ki ga razvijalci običajno uporabljajo pri razvoju.

OsebaForm.java

Razred `OsebaForm` razširja razred `OsebaBeanForm`, ki je generiran z orodjem XDoclet. Razred `OsebaBeanForm` vsebuje metode get/set za vse podatke entitetnega zrna `OsebaBean`, na podlagi katerega se ta razred generira. Osnovna funkcionalnost razreda `OsebaBeanForm` je razširjena s preverjanjem pravilnosti podatkov, ki jih je vnesel uporabnik (metoda `validate`).

```
package diploma.web;

import java.text.*;
import javax.servlet.http.*;

import org.apache.struts.action.*;
import diploma.interfaces.*;

/**
* Razred predstavlja model vnosnega obrazca za vpis podatkov o uporabiku.
* Razširja razred "OsebaBeanForm", ki se generira v prvi fazi uporabe orodja XDoclet.
 *
* XDoclet oznaka za generiranje datoteke "struts-config.xml
* @struts.form name="osebaForm"
*/
public  class OsebaForm  extends OsebaBeanForm {

 /**
  * Implementacija metode za preverjanje pravilnosti podatkov razreda
  * (vnešenih podatkov v spletnem obrazcu)
  **/
  public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)
  {
   ActionErrors errors =  new ActionErrors();
   // Ali so posamezni atributi osebe vnešeni
   if(getIme() ==  null) {
      errors.add("ime",  new ActionError("Pozabili ste vpisati ime osebe!"));
  }
   if(getPriimek() ==  null) {
      errors.add("priimek",  new ActionError("Pozabili ste vpisati priimek osebe!"));
  }
   if(getTel() ==  null) {
      errors.add("tel",  new ActionError("Pozabili ste vpisati telefonsko številko osebe!"));
  }
   if(getEmail() ==  null) {
      errors.add("email",  new ActionError("Pozabili ste vpisati elektronski naslov osebe!"));
  }
   if(getOpis() ==  null) {
      errors.add("opis",  new ActionError("Pozabili ste vpisati opis osebe!"));
  }
   // Ostala preverjanja ...
  return errors;
  }
}
```

DodajOsebaAction.java

Razred `DodajOsebaAction` je implementacija poslovne logike, povezane s spletnim obrazcem, ki ga predstavlja razred `OsebaForm`.

```
 package diploma.web;

 import java.io.IOException;
 import java.util.HashMap;
 import javax.servlet.*;
 import javax.servlet.http.*;
 import org.apache.struts.action.*;
 import javax.ejb.FinderException;
 import javax.naming.NamingException;
 import diploma.interfaces.*;

/**
 * Razred je zadolžen za izvajanje poslovne logike uporabniškega vmesnika,
 * posredno tudi za izvajanje poslovne logike informacijskega sistema.
 *
 * XDoclet oznake za generiranje datoteke "struts-config.xml".
 * @struts.action
 *    path="/dodajOseba"
 *
 * @struts.action-forard
 *    name="form"
 *    path="/pages/dodajOseba.jsp"
 **/
 public  final  class DodajOsebaAction  extends Action {

    /**
     * Izvajanje poslovne logike.
     **/
     public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request,
             HttpServletResponse response)  throws IOException, ServletException
    {
    String target = "success";

    if (form !=  null) {
        try {
             // Posredovane podatke posredujemo entitetnemu zrnu.
            OsebaForm osebaForm = (OsebaForm)form;
            OsebaBeanFacadeRemote osebaFRemote =  new OsebaBeanFacadeRemote();
            osebaFRemote.create(osebaForm.getIme(),  osebaForm.getPriimek(),
            osebaForm.getTel(), osebaForm.getEmail(), osebaForm.getOpis());
         }
        catch (Exception e) {
         target = "error";
       }
   }
    return mapping.findForward(target);
  }
}
```

dodajOseba.jsp

Datoteka `dodajOseba.jsp` predstavlja vidni del spletnega grafičnega uporabniškega vmesnika.

```
<%@ page  import="org.apache.struts.action.*, diploma.web.*, diploma.interfaces.*"%>

<%@ taglib uri="/tags/struts-html" prefix="html" %>

<html:html locale="true">
<head>
<title>Spletna izposojevalnica - vpis podatkov o osebi</title>
</head>

<body>
<html:errors/> <!-- Izpis napak ob vnosu podatkov. -->
<h3>Vpis podatkov o osebi</h3>
<!-- Obrazec z vnosnimi polji za vnos podatkov o osebi. -->
<html:form action="dodajOseba" name="osebaForm" type="diploma.web.OsebaForm">
  <TABLE>
    <TR>
       <!-- Vrednost vnosnega polja se prenese iz razreda OsebaForm -->
      <TD>Ime: </TD><TD><html:text property="ime" /></TD>
    </TR>
    <TR>
      <TD>Priimek: </TD><TD><html:text property="priimek" /></TD>
    </TR>
    <TR>
      <TD nowrap>Tel. stevilka: </TD><TD><html:text property="tel" /></TD>
    </TR>
    <TR>
      <TD>E-mail: </TD><TD><html:text property="email" /></TD>
    </TR>
    <TR>
      <TD>Opis: </TD><TD><html:text property="opis" /></TD>
      <TD><html:submit value="Dodaj osebo"/></TD>
    </TR>
  </TABLE>
</html:form>

</body>
</html:html>
```

struts-config.xml

Datoteka `struts-config.xml` je jedro vsake aplikacije razvite z ogrodjem Struts. V datoteki opišemo vse uporabljene komponente, njihovo obnašanje ob dani situaciji in povezave med njimi.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
    <form-beans>
       <form-bean
            name="uploadForm"
            type="diploma.web.UploadForm"/>
       <form-bean
            name="osebaForm"
            type="diploma.web.OsebaForm"/>
    </form-beans>
    <action-mappings>
        <action
            path="/dodajOseba"
            type="diploma.web.DodajOsebaAction"
            name="osebaForm"
            scope="request"
            validate="true"
            input="/pages/dodajOseba.jsp">
            <forward name="success" path="/main.jsp"/>
            <forward name="error" path="/pages/errorOseba.jsp"/>
       </action>
    </action-mappings>
</struts-config>
```