UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Viktor Brajak

Avtomatsko generiranje grafičnega uporabniškega vmesnika z označevanjem strežniške javanske kode

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Viljan Mahnič

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01740/2011
Datum: 15.03.2011

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

Kandidat: **VIKTOR BRAJAK**

Naslov: **AVTOMATSKO GENERIRANJE GRAFIČNEGA UPORABNIŠKEGA VMESNIKA Z OZNAČEVANJEM STREŽNIŠKE JAVANSKE KODE**

AUTOMATIC GENERATION OF GRAPHICAL USER INTERFACE USING ANNOTATED SERVER-SIDE JAVA CODE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Proučite značilnosti spletnih, namiznih in mobilnih poslovnih aplikacij ter na podlagi tega izberite najprimernejša programska okolja za njihov razvoj. Za ta okolja definirajte strežniški del kode, ki bo za vse zgoraj naštete vrste aplikacij enak in bo omogočal avtomatsko generiranje grafičnega vmesnika s pomočjo ustreznih anotacij. Opišite potrebne anotacije in prikažite njihovo uporabo na konkretnem primeru.

Mentor:
prof. dr. Viljan Mahnič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Viktor Brajak,

z vpisno številko 63040014,

sem avtor diplomskega dela z naslovom: Avtomatsko generiranje grafičnega uporabniškega vmesnika z označevanjem strežniške javanske kode

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Viljana Mahniča
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 15.06.2011 Podpis avtorja:

Zahvala

Rad bi se zahvalil mentorju, prof. dr. Viljanu Mahniču, ki je s konstruktivnimi kritikami in vloženim trudom pripomogel h konsistentnejši in razumljivejši diplomski nalogi.
Sodelavcem v podjetju, ki so z razvojem ogrodja omogočili, da sem imel zanimivo temo diplomske naloge, še posebej pa Martinu, Mateju in Neži, ki so pomagali pri izdelavi praktičnega preizkusa.
Staršem in sestri, za podporo ob študiju, še posebej očetu, ki me je spodbudil za študij računalništva in informatike.

Očetu, da čim prej napišemo pravo knjigo

Kazalo

Seznam uporabljenih kratic in simbolov

Ajax (ang. Asynchronous JavaScript and XML) - Skupina klientnih razvojnih metod, za razvoj interaktivnih spletnih aplikacij

AOP (ang. Attribute Oriented Programming) - Atributno usmerjeno programiranje

API (ang. Application programming interface) - Programski vmesnik, ki zagotavlja, da ima računalniški program na razpolago funkcije drugega računalniškega programa

AWT (ang. Abstract Window Toolkit) - Programska knjižnica z moduli za izdelavo okenskih uporabniških programov v javi

CRUD (ang. Create, Read, Update, Delete) - Najvažnejše funkcije implementirane v aplikaciji z relacijsko podatkovno bazo (ustvari, beri, uredi, izbriši)

CSS (ang. Cascading Style Sheet) - Prekrivni slogi, v katerih je zapisana oblika spletne strani

EJB (ang. Enterprise Java Bean) - Poslovno javansko zrno je komponenta, ki vsebuje polja in metode za implementacijo modulov poslovne logike

EJB 3 (ang. Enterprise Java Bean) - Določena vrsta poslovnega javanskega zrna, ki vsebuje polja in metode za implementacijo modulov poslovne logike

GWT (ang. Google Web Toolkit) - Razvojno orodje, ki omogoča gradnjo in optimizacijo kompleksnih spletnih aplikacij

HTML (ang. Hypertext Markup Language) - Označevalni jezik za oblikovanje večpredstavnostnih dokumentov, ki omogoča povezave znotraj dokumenta ali med dokumenti

HTTP (ang. Hyper Text Transfer Protocol) - Protokol za izmenjavo nadbesedil ter grafičnih, zvočnih in drugih večpredstavnostnih vsebin na spletu

IDE (ang. Integrated development environment) - Integrirano razvojno okolje, namenjeno programiranju, ki navadno vsebuje urejevalnik besedila, prevajalnik, povezovalnik in iskalnik napak (npr. Eclipse)

Java EE (ang. Java Platform, Enterprise Edition) - Javansko razvojno okolje, ki vsebuje javanske knjižnice za razvoj poslovnih aplikacij

JDBC (ang. Java Database Connectivity) - Standard v programskem jeziku Java, ki pove kako odjemalce dostopa do podatkovne baze

JPA (ang. Java Persistence API) - Knjižnica, ki omogoča obstojno preslikovanje objektov v podatkovno bazo

JSON (ang. JavaScript Object Notation) - Format za izmenjavo podatkov oziroma implementacijo komunikacijskih protokolov

JSP (ang. JavaServer Pages) - Tehnologija, ki omogoča enostavno in hitro kreiranje dinamičnih spletnih strani, ki so neodvisne od izvajalnega okolja.

JVM (ang. Java Virtual Machine) - Navidezni javanski stroj, kjer tečejo virtualni ukazi, neodvisni od izvajalnega okolja

ORM (ang. Object Relational Mapping) - Objektno relacijsko preslikavanje je tehnika, ki omogoča objektno orientiran dostop do podatkovne baze

POJO (ang. Plain Old Java Object) - Navaden javanski objekt, ki ne vsebuje posebnih funkcionalnosti

REST (ang. Representational State Transfer) - Stil spletne arhitekture v porazdeljenem sistemu, kjer gre za komunikacijo med strežnikom in odjemalcem

RIA (ang. Rich Internet Application) - Obogatena spletna aplikacija

RMI (ang. Java Remote Method Invocation) - Koncept, ki omogoča porazdeljenim sistemom, da kličejo oddaljene objektne metode in s tem delijo skupne vire in procesorski čas

SPI (ang. Single Page Inteface) - Tip spletne aplikacije, kjer ob pridobivanju novih podatkov na odjemalcu ni potrebno osveževati spletne strani

SQL (ang. Structured Query Language) - Strukturiran povpraševalni jezik za delo s podatkovnimi bazami

Standard JSF (ang. JavaServer Faces) - Standard za razvoj strežniških uporabniških vmesnikov in tehnologija za poenostavitev razvoja spletnih aplikacij

UIDL (ang. User Interface Definition Language) - Jezik za serializacijo vsebine uporabniških vmesnikov s spletnega strežnika na spletni brskalnik

XML (ang. Extensible Markup Language) - Razširljivi označevalni jezik, ki označuje format podatkov za izmenjavo strukturiranih dokumentov v spletu

Povzetek

Zahtevnost in kompleksnost poslovnih aplikacij se iz dneva v dan povečuje. Posledično vsebujejo poslovne aplikacije veliko število mask za upravljanje s podatki. Poleg tega današnje zahteve narekujejo, da se iste poslovne aplikacije izvajajo v različnih okoljih. Ker gre pri gradnji številnih, ponavljajočih se mask za zamudno delo, je bilo razvito ogrodje za avtomatsko kreiranje grafičnih uporabniških vmesnikov, ki omogoča razvijalcu, da svoj dragoceni čas usmeri raje v razvoj poslovne logike aplikacij. V diplomski nalogi govorimo torej o ogrodju, ki na podlagi označenih strežniških javanskih komponent avtomatsko kreira grafične uporabniške vmesnike za spletne, namizne in mobilne aplikacije. Analiziramo različna izvajalna in razvojna okolja, kjer lahko ogrodje uporabimo, opisujemo temeljne koncepte, ki omogočajo delovanje ogrodja, opisujemo označbe, ki so bile izdelane za potrebe ogrodja, komunikacijo med aplikacijskim strežnikom in tremi različnimi tipi odjemalcev ter preizkusimo ogrodje na praktičnem primeru. Ogrodje deluje na principu atributno usmerjenega programiranja. Razvijalec s pomočjo javanskih označb opiše entitetne razrede in njihove metode. Na podlagi interpretacije označb, ogrodje na odjemalčevi strani avtomatsko nariše maske za obvladovanje poslovnih podatkov. Prednosti pri razvoju aplikacij z uporabo ogrodja so torej, da za različno izvajalno okolje razvite aplikacije uporabijo enotno strežniško arhitekturo, opisana entitetna zrna, poslovna pravila, validacijo, varnost, itd. Uporaba ogrodja posledično prinese hitrejši, enostavnejši in enotnejši razvoj, skupno strežniško arhitekturo in poslovna pravila za različna izvajalna okolja in enotnejši izgled celotne aplikacije.

Ključne besede:

poslovna aplikacija, okolje Java EE, atributno orientirano programiranje, označbe, entitetna zrna

Abstract

Due to increasing complexity and nature of data-driven business applications, graphical user interfaces nowadays contain great number of different components for managing business data. Furthermore, same applications run on different platforms. Consequently, developers tend to focus more on the user interface aspects and less on the business related code. In this thesis, we speak about the framework, that automatically generates GUI for web, desktop and mobile applications, based on the annotated server-side Java code. We analyze different platforms and development environments, where the framework can be applied, we describe key concepts for framework operation, describe annotations, that framework uses to render GUI components, communication between an application server and three different types of clients, and examine the framework on a small but generic test case. The framework operates on the principles of the Attribute Oriented Programming. By using framework's annotations, developer describes the behavior of entity beans and their methods. As a result of the description, framework's engine interprets and builds client-side GUI components. Main advantages of this approach are that applications developed for different platforms use common server-side architecture, annotated entity beans, business logic, validation, security, etc. The use of the framework consequently leads to a faster, easier and more uniform development.

Key words:
data-driven business application, Java EE Platform, Attribute Oriented Programming, annotations, entity beans

Poglavje 1

Uvod

Če govorimo o delu računalništva kjer imamo v mislih razvoj in programiranje najrazličnejše programske opreme in aplikacij, se zelo hitro srečamo s problemom pisanja konsistentne in enotne kode. Danes, ko na večjih projektih dela in programira tudi po nekaj sto in več ljudi, je lahko to zelo velik problem. Problem se pojavi tudi pri predstavitvi in uporabi aplikacij. Za njihov prikaz se uporabljajo najrazličnejši mediji, kot so namizni računalniki, mobilne naprave, tablični računalniki in vsi zahtevajo drugačno tehnologijo aplikacij, ki se na njih izvajajo.

Poslovne aplikacije kot temelj uporabljajo podatkovno bazo in vsebujejo grafične uporabniške vmesnike, ki nam omogočajo dodajanje, urejanje, brisanje poslovnih podatkov in številne druge funkcionalnosti. V diplomskem delu se bomo pobližje spoznali z ogrodjem za avtomatsko izdelavo grafičnih uporabniških vmesnikov za poslovne aplikacije. Grafični vmesniki so sestavljeni iz številnih elementov, kot so polja, gumbi, izbirni meniji, itd. V velikih poslovnih aplikacijah so lahko taki elementi uporabljeni velikokrat v različnih kontekstih, njihovo število pa eksponentno narašča s kompleksnostjo aplikacije. Posledično, je čas razvijalcev, ki se ukvarjajo z nepotrebnim risanjem elementov namesto s pisanjem poslovne logike, zelo dragocen. Prav tako v večini aplikacij izgled grafičnega vmesnika ne predstavlja tako pomembnega faktorja, kot ga predstavlja uporabniška izkušnja in funkcionalnosti, ki jih vmesnik ponuja. Zato smo za razvoj poslovnih aplikacij razvili ogrodje, ki omogoča avtomatsko kreiranje grafičnih uporabniških vmesnikov na osnovi označevanja strežniške javanske kode. Ključna prednost tega ogrodja je, da močno zmanjša čas, potreben za risanje komponent grafičnega vmesnika.

Tako kot pri klasičnem razvoju poslovih aplikacij, bo tudi razvoj z ogrodjem temeljil na podatkovni bazi iz katere se podatki preslikajo v entitetne razrede.

Osnovni princip in delovanje ogrodja bo v tem, da bomo pridobljene entitete nato opisali z javanskimi označbami (ang. annotation), ki jih bo znalo ogrodje interpretirati in narisati na posamezno izvajalno okolje. Na osnovi ustrezne interpretacije označb, se bo zgradil grafični vmesnik za poslovno aplikacijo, ki bo vsebovala vse potrebne funkcionalnosti. To pomeni, da ogrodje deluje tako, da omogoča označevanje entitet in opis razmerij med entitetami, ter s tem določi obnašanje aplikacije na odjemalčevi oziroma predstavitveni strani. Prednost uporabe ogrodja je, da ni potrebno prilagajati strežniškega dela kode na posamezno izvajalno okolje. Strežniški del kode, ki vsebuje poslovna pravila in skrbi za komunikacijo in transakcije s podatkovno bazo, varnost, sočasnost je vedno enak, ne glede na to kakšen medij uporabljamo za prikaz aplikacije. Posledica uporabe je enotnejša programska koda v različnih izvajalnih okoljih in s tem enostavnejše programiranje in prilagajanje aplikacij za različne odjemalce. Zaradi uporabe skupnih komponent, vgrajenih v ogrodje, pa omejimo in omilimo problem konsistentnosti programske kode, ki jo pišejo različni programerji. Programska koda je tako enotnejša, s tem pa je enotnejši tudi končni izgled aplikacije.

S podobnimi modeli in koncepti avtomatskega generiranja grafičnih vmesnikov se ukvarjajo številni raziskovalci in razvijalci [4, 2, 10]. Če primerjamo njihove raziskave lahko opazimo, da ne upoštevajo vseh faktorjev, ki jih srečamo pri dejanskem razvoju aplikacij. Ne predvidijo velikega števila različnih entitet na eni maski, kompleksnih izbirnih menijev, ki nastanejo zaradi razmerij med entitetami, za razvoj modela ne uporabljajo že uveljavljenih standardov, modeli niso prilagodljivi na vsa izvajalna okolja [4]. Zopet drugi modeli so prilagojeni specifično na posamezno domeno, na primer za avtomobilsko industrijo in temu prirejene enostavne maske [2], ali maske za administriranje telefonskih central [10]. Poleg raziskav obstajajo tudi komercialni produkti, kot so Oracle Forms [21], ki pa večinoma generirajo izključno maske za izvajanje operacij CRUD (ang. Create, Read, Update, Delete), ki jih je nato težko prilagajati, če pride do sprememb v modelu ali poslovni logiki. Dobra lastnost našega ogrodja v primerjavi s komercialnimi je, da ohranja neprekinjen razvojni cikel. To pomeni, da ogrodje ni klasičen programski generator, ampak se prilagaja spremembam in omogoča enostavno spreminjanje generiranih mask.

V nadaljevanju bomo skušali opisati različna izvajalna okolja ter izbrati najprimernejše razvojno okolje za uporabo ogrodja za avtomatsko generiranje grafičnih vmesnikov, skušali bomo prikazati osnovne koncepte izdelave in uporabe ogrodja in prikazati praktično uporabo ogrodja v različnih izvajalnih okoljih. Omejili se bomo le na odprtokodne aplikacije in uporabo programskega jezika Java.

Cilj diplomskega dela je prikazati prednosti opisanega ogrodja, za katere smatramo da so:

- enotnejši razvoj aplikacij za različna izvajalna okolja,
- hitrejši razvoj aplikacij,
- bogat nabor grafičnih komponent, ki se avtomatično zgradijo iz opisanih entitet,
- enoten strežniški podatkovni model in pripadajoča poslovna pravila v različnih izvajalnih okoljih,
- enotna validacijska logika,
- neodvisnost obnašanja in oblike grafičnih komponent ne glede na število razvijalcev,
- enostavno spletno oblikovanje,
- razvoj izključno v programskem jeziku Java, kar omogoča lažje vzdrževanje za razliko od aplikacij, ki vsebujejo različne tehnologije - npr. Html, JavaScript, CSS, Ajax, ogrodja JSP/JSF, itd.

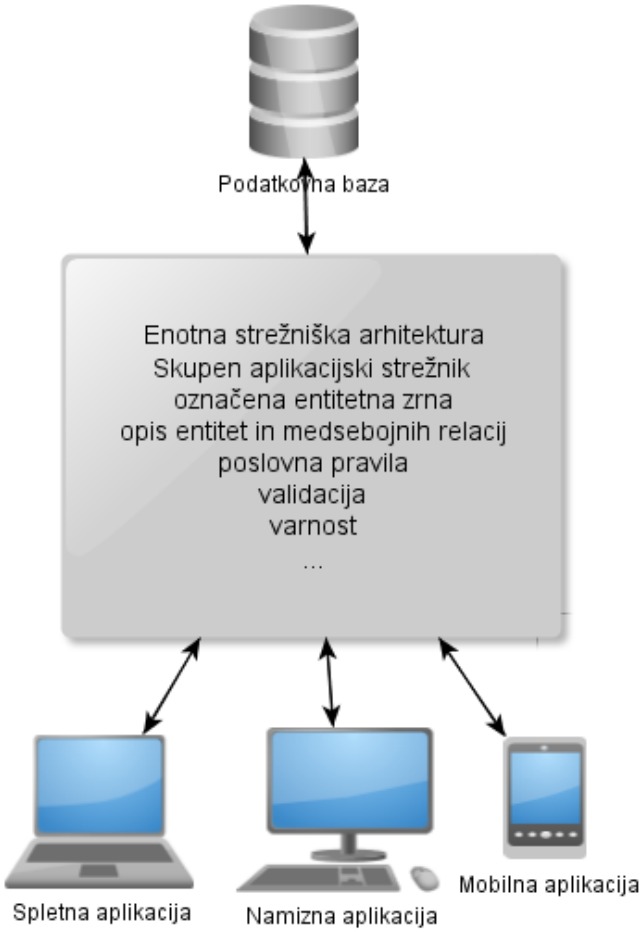
Poglavje 2

Izvajalno in razvojno okolje poslovnih aplikacij

V drugem poglavju si bomo podrobneje ogledali različna okolja za izvajanje aplikacij, ter znotraj posameznega izvajalnega okolja izbrali najprimernejše razvojno okolje za uporabo ogrodja za avtomatsko kreiranje grafičnih uporabniških vmesnikov. Izvajalna okolja smo ločili v tri veje: izvajalno okolje za spletne aplikacije, namizne aplikacije in mobilne aplikacije. Vsako izvajalno okolje zahteva svojevrsten pristop in tehniko razvijanja poslovnih aplikacij. Za posamezno okolje obstajajo najrazličnejši programski jeziki, v katerih so aplikacije napisane in najrazličnejša razvojna okolja ter ogrodja, ki nam pomagajo pri razvoju. Ker diplomsko delo govori o uporabi in konceptih izdelave ogrodja za avtomatsko izdelavo grafičnih vmesnikov v različnih izvajalnih okoljih, je nujno potrebno, da si podrobneje ogledamo posamezno izvajalno okolje in njihove razvojne značilnosti. Na podlagi tega, bomo enostavneje izbrali najprimernejše razvojno okolje za razvoj poslovnih aplikacij, z uporabo ogrodja za avtomatsko kreiranje grafičnih uporabniških vmesnikov v različnih izvajalnih okoljih.

Pri poslovnih aplikacijah v večini primerih operiramo s strogo zaupnimi podatki, ki morajo biti skrbno shranjeni in varovani. V osnovi so taki podatki največkrat shranjeni na strežniku v podatkovni bazi, do aplikacije pa hkrati dostopa veliko število uporabnikov. Zaradi tega je nujno potrebno, da poslovne aplikacije znajo upravljati s transakcijami, da zagotovijo atomarnost, konsistentnost, izolacijo, trajnost, da so varne in omogočajo dodajanje različnih pravic posameznim uporabnikom, da so interoperabilne in omogočajo enostavno povezovanje z ostalimi aplikacijami itd. Drugo vodilo, ki nam pogojuje izbiro razvojnega okolja, je osnovni cilj in ideja ogrodja za avtomatsko kreiranje grafičnih vmesnikov, kjer bi radi neodvisno od izbire izvajalnega okolja za

prikaz aplikacije uporabljali enotno strežniško arhitekturo, ki jo prikazuje slika 2.1. Poleg zgoraj naštetih kriterijev, bomo preverjali tudi številne druge lastnosti razvojnih okolij kot so odprtost kode, delež uporabe v produkcijskih okoljih, velikost in aktivnost skupnosti, potencial v prihodnosti, prijaznost pri programiranju, itd. Da bi zadostili vsem kriterijem in pogojem, ki jih postavljajo poslovne aplikacije, je potreben temeljit razmislek, kakšno razvojno okolje, ogrodje in orodje bomo izbrali za razvoj poslovne aplikacije.



Slika 2.1: Ideja za izdelavo ogrodja, ki uporablja enotno strežniško arhitekturo

2.1 Spletne aplikacije

Okolje, v katerem lahko komunicira več računalnikov in med seboj prenaša informacije in jih shranjuje v podatkovno bazo, imenujemo strežniška arhitektura. V tem primeru je klient oziroma odjemalec program, ki omogoča, da uporabnik uporablja aplikacijo, strežnik pa shranjuje potrebne podatke. Tako pridemo do ugotovitve, da je v spletni aplikaciji, kot odjemalec uporabljen spletni brskalnik. Prednost spletnih aplikacij je v tem, da niso omejene na specifičen tip računalnika ali operacijski sistem, zato je dostop do njih lažji, velikokrat odvisen le od možnosti dostopa do internetnega omrežja. Prav tako je administracija aplikacij centralizirana in bistveno lažja kot pri namiznih aplikacijah. V večini primerov uporabljajo spletne aplikacije določen programski jezik tako na strežniški strani (Java, ASP, Php, itd.), kjer gre za shranjevanje in priklic informacij, kot tudi na strani odjemalca (Html, JavaScript, itd.), ki skrbijo za ustrezen prikaz in predstavitev podatkov.

2.1.1 Kriteriji za izbor razvojnega okolja in ogrodja za izdelavo spletne aplikacije

Kot smo predhodno omenili, se bomo v diplomskem delu osredotočili na poslovne Java EE (ang. Java Platform, Enterprise Edition) aplikacije, razvite z odprtokodnimi orodji. V nadaljevanju bomo opisali osnovne koncepte razvoja aplikacij ter skušali izbrati najustreznejše razvojno okolje in ogrodje, v katerih bo moč implementirati ogrodje za avtomatsko izdelavo grafičnih vmesnikov in s tem poenostaviti načrtovanje, razvoj in vzdrževanje aplikacij, obenem pa zmogljivostno in funkcionalno zadovoljili zahtevam poslovnih uporabnikov. Glede na to, da danes poznamo nešteto število ogrodij za razvoj spletnih aplikacij, kar nam prikazuje slika 2.2, izbor ne bo lahek. V nadaljevanju bomo opisali lastnosti, ki si jih želimo, da bi jih imela naša spletna aplikacija.

2.1.1.1 Java in JVM

Java je objektno usmerjen programski jezik. Sintaksa je podobna C in C++, vendar za razliko od C in C++ ne vsebuje funkcij, ki bi bile kompleksne, nerazumljive in nevarne. Prvotno je bila Java razvita za izdelavo programske opreme v mrežnih napravah, kar omogoča arhitekturo z več gostitelji in varen prenos programskih komponent in prevedene kode. To omogoča, da Jave ni potrebno prilagajati na posamezne odjemalce ampak je koda za vse odjemalce enaka, kar v angleščini imenujemo *write once, run anywhere.*

Slika 2.2: Številna ogrodja za izdelavo spletnih aplikacij

S popularizacijo interneta so ti javanski koncepti še dodatno pridobili na vrednosti ter prikazali lažjo dostopnost do večpredstavitvenih vsebin. Tako lahko danes dostopamo do bogatih vsebin in aplikacij z najrazličnejših medijev, operacijskih sistemov, brskalnikov, kar postavlja Javo za enega izmed najbolj popularnih in uporabljenih programskih jezikov, ki se uporablja tako za razvoj spletnih aplikacij kot tudi pri izdelavi programske opreme.

JVM (ang. Java Virtual Machine) ali navidezni javanski stroj je temelj vseh javanskih okolij in je komponenta, ki je odgovorna za neodvisnost strojne opreme in neodvisnost med različnimi operacijskimi sistemi. JVM je navidezni računalniški stroj, ki vsebuje nabor ukazov in manipulira s spominom ob izvajanju aplikacij. JVM ne pozna programskega jezika Java, razume le prevedeno javansko kodo, ki jo imenujemo vmesna koda in vsebuje JVM navodila in ostale pomožne informacije. Kljub temu, da zaradi varnosti JVM določa striktne strukturne omejitve v vmesni kodi, se lahko vsak programski jezik, ki zadošča tem pogojem, izvaja na JVM. Tako je JVM uporabljen kot dostavno vozilo za ostale programske jezike, vseeno pa zagotavlja veliko robustnost, varnost in hitrost za razvoj [3].

V zadnjem času se kot alternativa programskemu jeziku Java pojavlja veliko

dinamičnih programskih jezikov, kot so Php, Groovy, Grails, Rails, Ruby in Pyhton. Na žalost ti programski jeziki ne dosegajo ključnih lastnosti za razvoj velikih poslovnih aplikacij. Zanje še niso razvita kvalitetna orodja IDE (ang. Integrated development environment), ki bi na primer omogočala hitro prestrukturiranje programske kode (ang. refactoring), ne omogočajo sočasnosti razvoja oziroma je razvoj težji v primeru večjega števila razvijalcev na isti kodi, ne zavedajo se prednosti prevajalnikov, ki generirajo robustno in hitro kodo, so počasna pri velikem številu uporabnikov in podatkov in zahtevna za kompleksno razhroščevanje, zlasti pri porazdeljenih aplikacijah.

Glede na opisane razloge dinamični programski jeziki zaenkrat še niso pripravljeni za razvoj kompleksnih poslovnih aplikacij, zato jeziki kot so Php, Groovy, Grails, Rails, Ruby in Pyhton ne pridejo v poštev pri razvoju spletnih aplikacij z ogrodjem za avtomatsko kreiranje grafičnih uporabniških vmesnikov.

2.1.1.2 Spletni odjemalci

Ker v diplomskem delu pišemo o razvoju poslovnih aplikacij, si želimo odprto, standardno in močno razvojno ogrodje. Poslovne aplikacije so transakcijske in večinoma shranjujejo podatke v podatkovno bazo, do katere dostopa večje število uporabnikov hkrati. Zato si želimo aplikacijo uporabljati s ,,pravim“ spletnim odjemalcem. Obenem si želimo razvijati aplikacijo, zasnovano na spletnih standardih, ki jih podpirajo vsi brskalniki (Html, CSS, JavaScript) in ne na osnovi ogrodij, ki so namenjena razvoju ,,baročnih“ grafičnih vmesnikov, kot je to značilno za JavaFX ali Flash, ki so tudi procesorsko zelo požrešni.

Na ta način so iz množice orodij izločena ogrodja Flash in JavaFX in interpretativna ogrodja kot so AjaxSwing in Eclipse RAP.

2.1.1.3 Strežniško usmerjeno ogrodje

Zahvaljujoči tehnologiji Ajax se zadnje čase uveljavlja razvoj v okolju programskega jezika JavaScript. Obstaja napačna predstava, da so JavaScript aplikacije RIA (ang. Rich Internet Application), obogatene spletne aplikacije. Aplikacije RIA vključujejo poleg kode JavaScript veliko oblikovalskih tehnologij (Html, CSS, slike, itd.), Ajax funkcionalnosti za doseganje koncepta SPI (ang. Single Page Inteface) ter veliko strežniških komponent za dostop do poslovnih pravil in podatkov.

Programiranje v tem okolju je zelo zahtevno zaradi naslednjih razlogov:

- JavaScript ni objektno usmerjen programski jezik,

- ni pomoči prevajalnikov in razvojnih ogrodij,
- ni dobrega razhroščevalnika,
- ni možno uvajati konceptov agilne metodologije razvoja (testi enot, urejanje kode, neprekinjena integracija, repozitorij skupne kode, itd.),
- nepreglednost kode,
- velika nevarnost vdora, če se v JavaScript jeziku direktno dostopa do podatkov (npr. direktna uporaba SQL ukazov),
- podvojenost validacijske logike na strežniku in JavaScript odjemalcu.

Zaradi zgoraj naštetih razlogov so iz množice JavaScript ogrodij izločena vsa ogrodja (ExtJS, Dojo, jQuery, YUI) razen ogrodja GWT (ang. Google Widget Toolkit), ki ima lastnost, da se programska koda, napisana v Javi, prevede v kodo JavaScript in izvaja v brskalniku [16]. Vendar ima tudi ogrodje GWT pomanjkljivost, saj je koda JavaScript avtomatsko kreirana in zahtevna za vzdrževanje. Razvojni cikel pa je zaradi nenehnega prevajanja kode počasen. Prav tako je validacijska logika podvojena na strežniku in odjemalcu.

2.1.1.4 SPI – Single Page Interface

Glede na to, da je ključen kriterij potreba po strežniško usmerjenem ogrodju, mora ogrodje podpirati izvajanje dogodkov in delnih sprememb v formah podobno, kot pri namiznih aplikacijah. To je kritičen pogoj za poslovne aplikacije, pri katerih so značilni masovni vnosi podatkov in izvajanje operacij CRUD. Zaradi tega ni zaželeno, da se nenehno izvaja osveževanje strani. Koncept, ki zagotavlja, da se vsebina strani na brskalniku spreminja brez popolne osvežitve strani imenujemo SPI (Single Page Interface). S tem omogočimo večjo zmogljivost in hitrost uporabniških aplikacij, čarovnikov, itd. Prav tako se izognemo problemu shranjevanja podatkov na trenutno sejo. Ob upoštevanju SPI pogoja se izločijo ogrodja, ki tega koncepta ne podpirajo (Spring, Struts, Tapestry, Wicket, itd).

2.1.1.5 Standard JSF

Java specifikacija predvideva implementacijo spletnih aplikacij na osnovi tehnologije JSF (ang. Java Server Faces). Na žalost je v nasprotju z idejo o uporabi enostavnega komponentnega modela postal razvoj v okolju JSF nočna mora razvijalcev in vzdrževalcev spletnih aplikacij, ki ne morejo enostavno

ločiti vizualnega dela projekta (Html, CSS) od poslovnih pravil in dostopa do podatkov. Koncept povezovanja komponent je programersko nenaraven, zato se pojavljajo ogrodja (WebBeans, Seam), ki skušajo omiliti problem razvoja in zagotavljanja koncepta SPI s povezovanjem JSF tehnologije z Ajax komponentami (Rich Faces, OpenFaces, ICEFaces, itd). Na žalost, so ta ogrodja večinoma črne škatle z vsiljeno vizualno obliko, zahtevnim načinom spremembe oblike in lastnosti komponent, ki povzročijo dolg razvojni cikel in zelo zahtevno vzdrževanje aplikacije.

2.1.2 Ogrodje Vaadin

Po analizi željenih kriterijev in številnih ogrodij ter želji, da je razvoj izključno v Javi pridemo do zaključka, da je ogrodje Vaadin edino, ki zadovoljuje vse kriterije, ki jih poslovna aplikacija potrebuje, obenem pa omogoča enostavno izdelavo in uporabo ogrodja za avtomatsko izdelavo grafičnih uporabniških vmesnikov na osnovi označevanja javanskih entitetnih zrn.

Lastnosti ogrodja Vaadin [1]:

- Razvojno okolje je enotno; uporablja se samo programski jezik Java, ostale tehnologije in ogrodja (JavaScript, Ajax, JSON, Html, CSS, itd.) so vgrajene v ogrodje in skrite programerju.
- Je strežniško okolje, ki omogoča prenos vseh sprememb form in dogodkov na odjemalca s konceptom SPI; preko http protokola se prenašajo samo dogodki in delne spremembe form.
- Obnašanje oziroma poslovna logika aplikacije je ločena od prikaza in na ta način omogoča enostavnejše vključevanje oblikovalcev pri razvoju aplikacije.
- Hkraten razvoj več programerjev na isti kodi je enostaven, ker razvoj poteka izključno v Javi.
- Ogrodje je enostavno vključiti v Java EE.

2.2 Namizne aplikacije

Namizne (ang. desktop) aplikacije so naravno okolje za kompleksne poslovne aplikacije, ki predvidevajo veliko število uporabnikov in masovni vnos podatkov z operacijami CRUD.

Razlogi za razvoj namiznih aplikacij za poslovne uporabniške vmesnike so naslednji:

- bogata uporabniška funkcionalnost grafičnih vmesnikov,
- hiter zagon in izvajanje, ker se izkorišča okolje lokalnega operacijskega sistema (ang. native look & feel),
- arhitektura omogoča enostavno vključevanje in izključevanje modulov oziroma vtičnikov (ang. plugins) in različnih uporabniških vlog v aplikaciji,
- možnost izvajanja aplikacije z ali brez povezave s spletom,
- razvojno okolje je enostavno in je zasnovano na grafičnem orodju (Eclipse RCP, Netbeans RCP) in ne na grafičnih komponentah (ang. widgets), zato je vzdrževanje cenejše in enostavnejše,
- velik nabor komponent za internacionalizacijo, pomoč, distribucijo, itd.,
- enostavno vključevanje spletnih storitev in boljša interakcija med aplikacijami z arhitekturo SOA,
- za centralizacijo poslovnih pravil in dostop do podatkov se uporablja strežniška Java EE arhitektura,
- namestitev in vzdrževanje je avtomatično in centralizirano, kot pri spletnih aplikacijah.

Bogata uporabniška funkcionalnost grafičnih vmesnikov je ključna za poslovne aplikacije. Uporabnik v sistemu dostopa do različnih entitet hkrati: pregleduje procesne dogodke, pripadajoče dokumente, išče po dokumentih, ureja in razvršča entitete, deli entitete z drugimi uporabniki, ustvarja nove entitete, oblikuje objave, itd. Da lahko uporabniku omogočimo enostavno in sistematično delo vsega naštetega, mora odjemalec vsebovati močan ter bogat grafični vmesnik. Razvoj aplikacij na osnovi spletnih tehnologij (JSP, JSF, Html, JavaScript - Ajax, CSS) je primeren za enostavnejše aplikacije, ki se občasno uporabljajo in ne zahtevajo kompleksnih uporabniških in poslovnih elementov. Spletne aplikacije so najprimernejše v okolju z večjim številom uporabnikov, ko ni možno izvajati kontrolo nad izvajalnim okoljem uporabnika (tip operacijskega sistema, brskalnika, itd.), ali v primeru, da sistem ne dovoljuje namestitve navideznega javanskega stroja na lokalnih računalnikih.

Za namizne poslovno kritične aplikacije je uporabniška izkušnja kritična. Zelo pomembna je uporaba funkcijskih tipk in tipk z bližnjicami, funkcionalnost primi in spusti, pogledi na različne vsebine hkrati, integracija z ostalimi namiznimi aplikacijami, kvalitetno izdelana poročila, itd. Namizne aplikacije so funkcionalno prilagojene masovnem vnosu in spreminjanju podatkov; uporabniki uporabljajo čarovnike ter bogat nabor visoko nivojskih grafičnih komponent kot so drevesa, zavihki, perspektive, pogledi, izbirne forme, dinamični meniji, orodne vrstice, mape, katalogi, seznami opravil, lastnosti, itd. Takšne namizne aplikacije so najpogosteje del poslovno kritičnih aplikacij, ki jih dnevno uporabljajo za uporabo izšolani uporabniki.

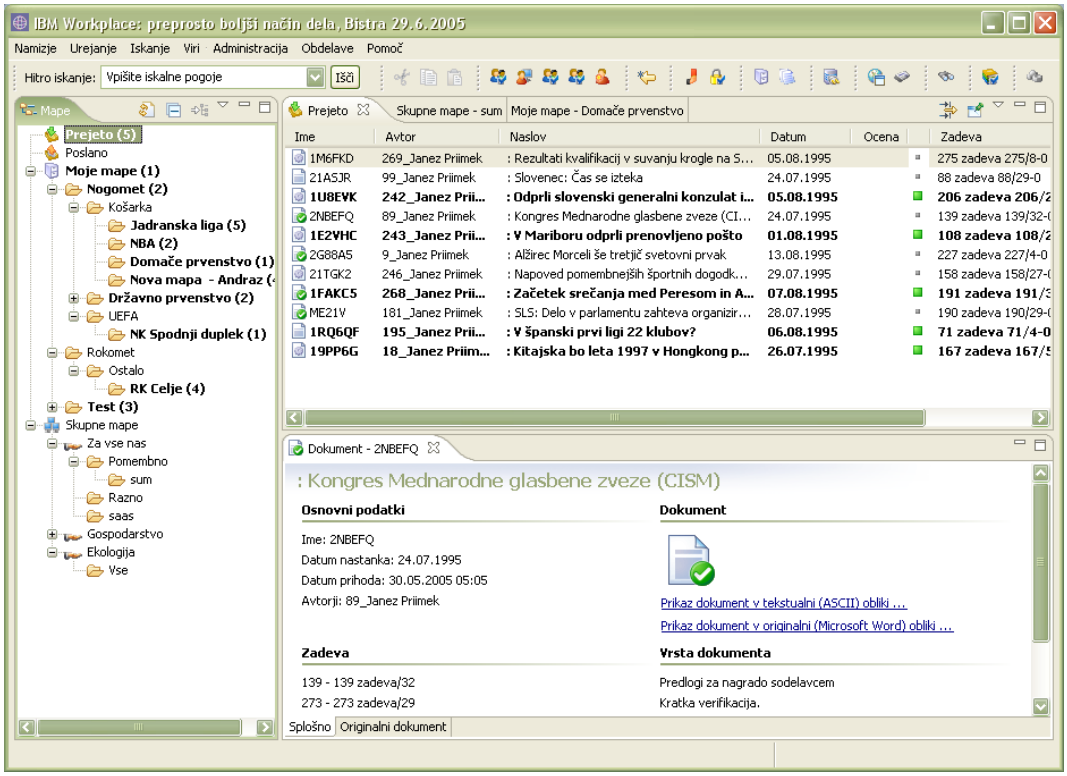
Zaradi omenjenih razlogov je pomembno, da ogrodje za razvoj namiznih aplikacij omogoča implementacijo in uporabo ogrodja za avtomatsko kreiranje form. Pogoji za izbiro razvojnega ogrodja so zanesljivo, standardno in uveljavljeno odprtokodno ogrodje, ki se izvaja na vseh okoljih (Windows, Linux, Mac, Unix). Ogrodji, ki sta obenem tudi orodji, ki popolnoma zadovoljita vse kriterije za razvoj bogatih namiznih aplikacij sta Eclipse RCP in NetBeans RCP. Eclipse RCP je že dlje prisoten, vendar pa je orodje NetBeans RCP izdelano na osnovi orodja Swing/AWT (ang. Abstract Window Toolkit), ki je bilo standardno razvojno orodje od samega začetka programskega jezika Java. Obe orodji omogočata uporabo ob velikem številu razvijalcev. Za razvoj Eclipse RCP je odgovorno podjetje IBM, medtem ko je za Netbeans odgovoren Oracle. Glede na to, da so razlike med obema orodjema majhne je izbira zelo težka. Ključni razlog za izbiro Eclipse RCP orodja je, da so komponente v tem razvojnem okolju zgrajene na osnovi gradnikov operacijskega sistema (Win32, GTK, Motif, Mac). S tem je izgled in uporaba aplikacije poenotena z ostalimi aplikacijami na sistemu, izris pa je bistveno hitrejši, ker je del operacijskega sistema in ne JVM.

2.2.1 Eclipse RCP

Projekt Eclipse je uveljavljen ne samo kot vodilno orodje IDE za razvoj javanskih aplikacij, ampak kot standarden API (ang. Application programming interface) za razvoj bogatih namiznih javanskih aplikacij. Eclipse RCP je edino orodje, ki se s hitrostjo, grafično podobo in enostavnostjo lahko primerja z razvojnim okoljem Microsoft .NET. Ima bogat nabor visoko nivojskih knjižnic API, izredno učinkovito in dodelano objektno metodologijo, veliko število razvijalcev in odprtokodnih projektov ter omogoča hiter razvoj namiznih aplikacij tudi v izvajalnem okolju Linux in MacOS [14].

Eclipse RCP je v bistvu splošno Eclipse IDE 3.0 orodje, ki ga razvijalci

lahko razširijo s svojimi dodatki oziroma vtičniki, da dobijo funkcionalnosti za potrebe lastnih aplikacij. Aplikacija je tako sestavljena iz vsaj enega uporabniškega vtičnika, uporablja pa ostale vtičnike orodja Eclipse IDE 3.0, ki so potrebni, da aplikacija dobi potrebne funkcionalnosti in bogat videz grafičnega vmesnika. Na ta način je orodje Eclipse RCP postalo ključen API za razvoj bogatih namiznih javanskih aplikacij. Slika 2.3 prikazuje tipičen primer namizne aplikacije, razvite v razvojnem okolju Eclipse RCP, ki vključuje visoko nivojske komponente, značilne za namizne odjemalce, kot so perspektive, pogledi, čarovniki, zavihki, forme, povečave, izbirne menije, dialoge, vgrajena orodja (npr. Open Office), itd.



Slika 2.3: Primer namizne aplikacije, razvite v razvojnem okolju Eclipse RCP

Uporaba orodja Eclipse RPC omogoča razvijalcu, da se osredotoči na funkcionalnost lastnih grafičnih komponent, razvojnem okolju, v katerem deluje orodje, pa prepusti ostale funkcionalnosti, kot so oblika, videz, postavitev in grafično obnašanje namizne aplikacije.

Z nastankom Eclipse RCP je nastalo razvojno orodje (ang. toolkit) za izdelavo uporabniških vmesnikov, ki ga lahko uporabljamo tudi samostojno,

in se imenuje SWT (ang. Standard Widget Toolkit). Gre za platformno neodvisen API, ki je tesno povezan z okoljem operacijskega sistema. Ta pristop omogoča javanskim razvijalcem, da lahko implementirajo rešitve, ki so z uporabniškega stališča enake hitrosti in oblike kot domorodne aplikacije operacijskega sistema. Ta API odpravlja mnoge implementacijske omejitve s katerimi se razvijalci srečujejo pri uporabi orodja Java AWT (ang. Abstract Window Toolkit).

2.3 Mobilne aplikacije

Hiter razvoj mobilnih naprav in tabličnih računalnikov je strmo povečal povpraševanje za uporabo poslovnih aplikacij na terenu. Kljub svoji majhnosti in priročnosti tablični računalniki omogočajo skoraj povsem enako uporabo in izvajanje kompleksnih aplikacij kot običajni namizni ali prenosni računalniki in v prihodnje se bodo zmogljivosti le še izboljševale.

2.3.1 Google Android SDK

Danes poznamo več operacijskih sistemov, ki tečejo na mobilnih in tabličnih računalnikih in omogočajo izvajanje lastno razvitih aplikacij. Ogrodje za avtomatsko kreiranje grafičnih vmesnikov je prilagojeno operacijskemu sistemu Google Android, ki v tem trenutku dominira na trgu pametnih mobilnikov in tabličnih računalnikov. Operacijski sistem Google Android ima danes 33 odstotni delež in najvišjo rast med vsemi ponudniki mobilnih operacijskih sistemov, sledijo mu, RIM (BlackBerry) z 28,9 odstotnim deležem in Apple s 25,2 odstotnim deležem [11]. Kot odprtokodni projekt spada Google Android v konzorcij osemdesetih podjetij, ki se ukvarjajo s telekomunikacijami in z izdelavo bodisi strojne ali programske opreme. Še večji pomen pa ima skupnost razvijalcev, ki je do danes razvila že več kot 150.000 uradnih aplikacij, ki močno povečajo funkcionalnost mobilnih naprav z operacijskim sistemom Android.

Android je odprto kodni paket programske opreme za mobilne naprave, ki vsebuje operacijski sistem, vmesno programsko opremo (ang. middleware) in ključne aplikacije. Narejen je na osnovi operacijskega sistema Linux. Sestavljen je iz javanskih aplikacij, ki tečejo na objektno usmerjenem aplikacijskem okolju na osnovi javanskih knjižnic in navideznem stroju Dalvik.

Zaradi odprtokodnega razvojnega okolja omogoča razvijalcem izdelavo zelo bogatih in inovativnih aplikacij. Razvijalci lahko svobodno uporabljajo vse strojne zmogljivosti naprave, dostopajo do informacij, izvajajo osnovne storitve,

nastavljajo alarme, itd. Omogočen je popoln dostop do programskega vmesnika Android API, ki ga uporabljajo temeljne aplikacije, zato je aplikacijska arhitektura načrtovana tako, da omogoča čim lažjo uporabo obstoječih komponent. Zaradi vseh zgoraj naštetih lastnosti se zdi izbira operacijskega sistema Google Android zelo smiselna, saj je v prednosti pred ostalimi ponudniki, kot so Apple, BlackBerry in Microsoft, predvsem v odprtosti kode in prijaznosti do razvijalcev programske opreme.

Android aplikacije so napisane v programskem jeziku Java, ogrodje Android SDK pa prevede kodo v paket Android. Paket Android je datoteka s končnico *.apk* in šteje kot ena aplikacija, ki jo Android namesti za uporabo na napravi. Ko je aplikacija nameščena, ji operacijski sistem dodeli unikatno identifikacijsko številko, kjer vsaka številka predstavlja enega uporabnika. Vsak proces ima prav tako svoj navidezni stroj, tako aplikacijska koda teče ločeno od ostalih aplikacij. S tem je zagotovljeno varno okolje, kjer imajo posamezne aplikacije določene privilegije dostopa do sistemskih storitev in drugih aplikacij [12].

Poglavje 3

Temelj ogrodja - okolje Java EE

Temelj ogrodja za avtomatsko kreiranje grafičnih vmesnikov v različnih izvajalnih okoljih je okolje Java EE. Zato se bomo v tretjem poglavju bolj podrobno posvetili prav okolju Java EE ter njegovim osnovnim komponentam, ki omogočajo izdelavo in uporabo ogrodja za avtomatsko kreiranje grafičnih vmesnikov na osnovi označevanja strežniških komponent Java EE. Preden pa začnemo s podrobno razlago okolja Java EE in njegovih značilnosti, je dobro, da spoznamo osnovno idejo ogrodja za avtomatsko kreiranje grafičnih uporabniških vmesnikov.

Koncept ogrodja

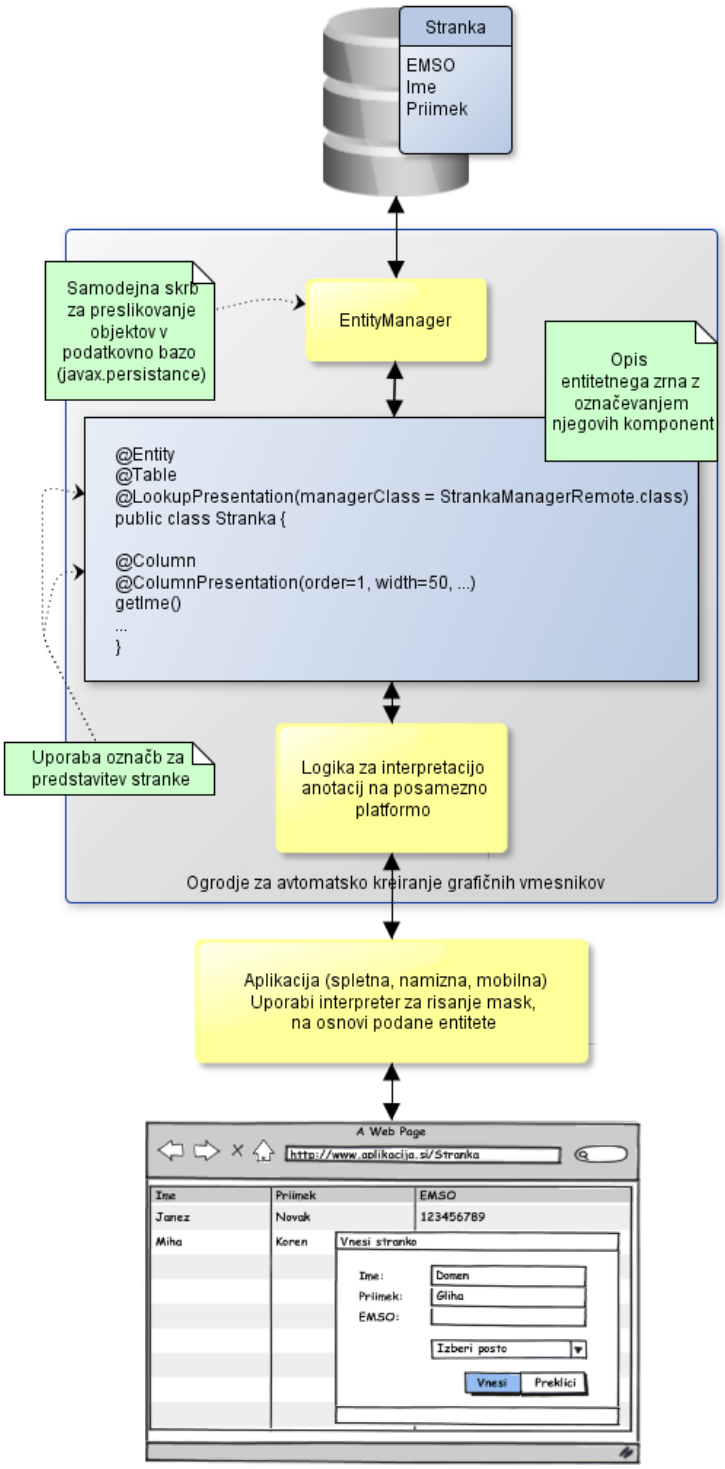
Motivacija za izdelavo ogrodja izhaja predvsem iz tega, da bi poenostavili in poenotili razvoj grafičnih vmesnikov v različnih izvajalnih okoljih. Če med seboj primerjamo aplikacije v različnih izvajalnih okoljih vidimo, da lahko vse uporabljajo skupno podatkovno bazo in skupen strežniški del kode. To pomeni, da bomo za razvoj spletne, namizne ali mobilne aplikacije povsod uporabili skupno podatkovno bazo in strežniški del kode s pripadajočimi poslovnimi pravili, prav tako pa bodo vse uporabljale skupno ogrodje za avtomatsko kreiranje grafičnih uporabniških vmesnikov.

Najenostavneje bo koncepte ogrodja za avtomatsko kreiranje grafičnih vmesnikov razložiti s primerjavo klasičnega razvoja aplikacij Java EE z razvojem s pomočjo našega ogrodja. Za poslovne aplikacije Java EE je značilna trinivojska arhitektura, kjer je prvi nivo podatkovna baza, drugi nivo aplikacijski strežnik in tretji nivo odjemalec. Zato razvoj v večini primerov začnemo z načrtovanjem in realizacijo podatkovne baze. Ko imamo podatkovno bazo, je iz nje enostavno pridobiti entitetna zrna. Entitetna zrna so javanski razredi, nameščeni na aplikacijskem strežniku, ki s pomočjo koncepta JPA (ang. Java

Persistence API) preslikujejo podatke med podatkovno bazo in aplikacijskim strežnikom. Entitetno zrno predstavlja tabelo v podatkovni bazi, instanca entitetnega zrna pa predstavlja en zapis v tabeli. Da bi javanski razred postal entitetno zrno, ga je potrebno opisati z označbami JPA (npr. @Entitiy, @Table). Na podlagi teh anotacij zna entitetni upravljalec (ang. EntityManager) operirati z entitetnimi zrni. Sedaj, ko je poskrbljeno za komunikacijo med podatkovno bazo in aplikacijskim strežnikom, je potrebno realizirati tisti del aplikacije, ki poskrbi za izris grafičnega vmesnika na odjemalčevi strani. To pomeni, da če želimo preko odjemalca, bodisi je to spletni brskalnik, namizna aplikacija ali mobilna naprava, upravljati s poslovnimi podatki, je potrebno narisati grafični vmesnik z ustreznimi maskami. Iz tega sledi, da mora razvijalec za posamezno entitetno zrno narisati vrstice, stolpce, gumbe, vnosna polja, izbirne menije, itd. V poslovnih aplikacijah je lahko takih entitetnih zrn tudi več sto, obenem pa jih je potrebno narisati za odjemalca vsake platforme. Da bi se izognili tako zamudnemu delu in poenotili razvoj, je nastalo ogrodje za avtomatsko kreiranje grafičnih vmesnikov. Ker si vse platforme delijo skupno strežniško kodo, je potrebno logiko ogrodja za avtomatsko kreiranje grafičnih vmesnikov aplicirati prav na ta del kode. Zato so, poleg že obstoječih anotacij JPA, nastale dodatne anotacije za risanje mask grafičnih vmesnikov, s katerimi opišemo entitetna zrna in njihove metode, ter opišemo povezave med entitetnimi zrni. To pomeni, da ob uporabi našega ogrodja aplikacije v različnih izvajalnih okoljih uporabljajo enako strežniško kodo, ki pa je še dodatno opisana s temi označbami. Na podlagi z označbami opisanih entitetnih zrn in interpreterja ogrodja, ki zna za posamezno izvajalno okolje narisati ustrezne maske na grafičnem vmesniku, se izognemo nepotrebnemu programiranju, ki bi ga morali opraviti v primeru klasičnega razvoja aplikacije.

Osnovni koncept ogrodja prikazuje slika 3.1, kjer lahko opazimo, da sta temelj ogrodja podatkovna baza in koncept JPA. Razred *Stranka* na sliki prikazuje primer z označbami opisnega entitetnega zrna. Kot vidimo, je entitetno zrno opisano z označbami JPA (@Entity, @Table, @Column) in dodatnimi označbami ogrodja (@LookupPresentation, @ColumnPresentation) za prikaz na grafičnem vmesniku. Za komunikacijo in preslikovanje podatkov s podatkovno bazo skrbi entitetni upravljalec. Sedaj je potrebno kreirati aplikacijo, bodisi je to spletna, namizna ali mobilna aplikacija, ki kliče strežniško logiko v obliki sejnega zrna. Sejna zrna prenesejo entitena zrna na zahtevo v aplikacijo. V aplikaciji je vključeno ogrodje, ki to entiteto uporabi za risanje maske na odjemalcu. V aplikaciji ogrodju povemo, katera komponenta naj se nariše (npr. tabela, vnosna maska, itd.), in kot parameter pošljemo entiteto. Ogrodje nato na podlagi podanih parametrov in označb, s katerimi je opisana

entiteta, nariše komponente na odjemalcu. Na ta način nastanejo privzeti grafični vmesniki. V primeru, da privzeti grafični vmesniki niso zadovoljivi, jih lahko preprosto spreminjamo, brez da bi spreminjali strežniško javansko kodo.



Slika 3.1: Osnovni koncept ogrodja z označenimi entitetnimi zrni

Sedaj, ko smo spoznali osnovno idejo ogrodja in se zasuli z njegovimi številnimi koncepti, je čas, da začnemo z njihovo podrobno razlago.

Predhodno smo velik pomen posvetili potrebi po porazdeljenih, transakcijskih in prenosnih aplikacijah, ki morajo biti hitre, varne in zanesljive. V svetu informacijskih tehnologij morajo biti poslovne aplikacije načrtovane in izdelane hitro, poceni in s čim manjšo porabo razpoložljivih virov. Z uporabo okolja Java EE je razvoj bistveno hitrejši in lažji, njen cilj pa je zagotoviti razvijalcem močno množico programskih vmesnikov API, ki jih lahko uporabijo za uresničitev zahtev današnjih aplikacij. Okolje Java EE uporablja enostaven model za programiranje. Namestitev deskriptorjev XML (ang. Extensible Markup Language) je opcijska, namesto tega lahko razvijalec vnese informacijo kot označbo (anotacijo) neposredno v javansko izvorno kodo, strežnik Java EE pa definira komponento ob namestitvi oziroma izvajanju aplikacije. S tem je opis podatka zapisan neposredno v kodi, zraven programa, na katerega se podatek nanaša, in ne v namestitvenem deskriptorju.

Aplikacijski model Java EE

Temelj aplikacijskega modela Java EE je programski jezik Java in JVM, osnova pa je prenosljivost, varnost in razvojna produktivnost. Java EE je načrtovana tako, da omogoča podporo aplikacijam, ki vsebujejo storitve za stranke, zaposlene, dobavitelje, partnerje in ostale, ki dostopajo do storitev nekega podjetja. Take aplikacije so zato kompleksne, saj dostopajo do podatkov na različnih virih in distribuiranih sistemih in do različnih odjemalcev.

Aplikacijski model Java EE določa arhitekturo za implementacijo storitev. Deli jo v dva dela in sicer, poslovno in predstavitveno logiko, ki jo implementira razvijalec, in standardne sistemske storitve, zagotovljene z okoljem Java EE. S tem se lahko razvijalec zanese, da bo platforma skrbela za težke sistemske probleme, ki se pojavijo pri razvoju večnivojske aplikacije.

Porazdeljene večnivojske aplikacije

Okolje Java EE za gradnjo poslovnih aplikacij potrebuje porazdeljeni večnivojski model. Aplikacijska logika je deljena na komponente glede na funkcijo, komponente pa so lahko nameščene na različnih strojih, odvisno od nivoja kateremu posamezna komponenta pripada. Porazdeljene aplikacije delimo na tri nivoje [9]:

- Nivo odjemalca - komponente tečejo na odjemalčevem stroju (prenosni računalnik, namizni računalnik, mobilna naprava)

- Spletni in poslovni nivo - komponente tečejo na strežniku Java EE (aplikacijski strežnik)
- Podatkovna baza

Govorimo lahko tudi o treh skupinah komponent:

- Spletni odjemalci in apleti - tečejo na odjemalcu (lahki odjemalci)
- Tehnološke ali spletne komponente (ang. Java Servlet, JavaServer Faces, JSP - JavaServer Pages) - tečejo na spletnem delu aplikacijskega strežnika
- Poslovne komponente (EJB - ang. Enterprise Java Beans) - tečejo na poslovnem ali EJB delu aplikacijskega strežnika

Java EE povečuje težnjo po razvoju lahkih aplikacij in poenostavitvi razvojnega cikla. V nadaljevanju bomo spoznali nekatere funkcionalnosti Java EE, ki so ključne za razvoj ogrodja za avtomatsko kreiranje grafičnih vmesnikov na osnovi označevanja strežniških komponent Java EE.

3.1 Atributno usmerjeno programiranje

Atributno usmerjeno programiranje ali AOP (ang. Attribute Oriented Programming) je tehnika, ki omogoča razvijalcem, da opišejo programske elemente, kot so razredi, metode, lastnosti, v izvorni kodi. Z njim skrijemo implementacijske podrobnosti in povišamo nivo abstrahiranosti ter s tem zmanjšamo kompleksnost programiranja, kar pripelje do bolj berljive programske kode. Atributi z dodajanjem deklarativnih informacij entitetam povečajo vrednost metapodatkov, ki jih vsebuje aplikacija. Atributi omogočajo spreminjanje aplikativne logike s pomočjo programov, ki interpretirajo atribute in na ta način v realnem času spreminjajo izvajanje aplikacije, ne da bi se pri tem spremenila izvorna koda [6]. Z razvojem atributov v okolju .NET in anotacij v okolju Java je atributno usmerjeno programiranje razširejena tehnika v različnih domenah, kot so spletne storitve, preslikovanje, varnost, itd. AOP je osnovni koncept uporabe ogrodja za avtomatsko kreiranje grafičnih vmesnikov.

3.1.1 Označbe

Označbe ali anotacije so bile v programskem jeziku Java uvedene v verziji JDK 1.5 in omogočajo da povežemo metapodatke in konfiguracijske datoteke direktno z javanskimi razredi, metodami in polji v izvorni kodi. Na primer,

z anotacijo lahko direktno povemo, ali je nek javanski razred entitetno zrno. Entitetno zrno je tip javanskega zrna, ki predstavlja podatek, zapisan v podatkovni bazi. Z anotacijami lahko nadomestimo veliko konfiguracijskih datotek in v veliki večini primerov povsem odstranimo potrebo po nepreglednih namestitvenih deskriptorjih XML [8]. Uporabljamo jih kot meta oznake v naši kodi, kjer lahko označimo deklaracije paketov, tipov, konstruktorje, metode, parametre in spremenljivke. Tako lažje označimo, ali je kakšna metoda odvisna od katere druge metode, ali imajo razredi reference na kakšen drug razred, itd. Anotacije interpretira aplikacijski strežnik in jih upošteva pri svojem delovanju. V primeru, da okolje anotacije ne prepozna, le ta nima nobenega vpliva na delovanje. Oznako za anotacijo predstavlja znak „@“, ki ji sledi niz znakov oziroma tekst s katerim želimo dodatno označiti našo kodo.

Naredimo lahko tudi svoje anotacije ter s tem v programsko kodo dodamo podporo svojim metapodatkom. Lastno razvite anotacije bo za prikazovanje podatkov na maskah grafičnih vmesnikov uporabljalo tudi naše ogrodje. Ko razvijamo svoje anotacije, lahko določimo, ali bodo dostopne samo v izvorni kodi, ali tudi v prevedenem razredu, torej dostopne JVM ob izvajalnem času. Primer lastno razvite anotacije @TablePresentation, ki jo bomo kasneje uporabili za gradnjo grafičnih vmesnikov, prikazuje izsek programske kode 3.1. Gre za anotacijo, kjer se zrno, ki je označeno s @TablePresentation, v grafičnem vmesniku predstavi z lastnostmi tabele.

Izvorna koda 3.1 Primer izdelane anotacije @TablePresentation

```
@Retention(RetentionPolicy.RUNTIME) //anotacija bo
//dostopna VM, da jo lahko bere ob izvajalnem času.
@Target(ElementType.METHOD) //povemo da se anotacijo
//lahko uporabi na metodi
public @interface TablePresentation {
    final String preSet="default";
    String[] prezentacije() default preSet;
    int orderWeight() default 0;
    boolean editable() default false;
}
```

3.2 POJO

POJO je angleški akronim za *Plain Old Java Object*, torej navaden javanski objekt, ki ne vsebuje posebnih funkcionalnosti. Razvojni model POJO zagovarja stališče, da morajo razvijalci vedno oceniti, ali prednosti ogrodja, ki ga uporabljajo pri razvoju opravičijo njegovo kompleksnost. Dejstvo je, da kompleksna oziroma težka ogrodja razvijalcem vzamejo veliko časa. Zato se povečuje uporaba razvojnega modela POJO, hkrati pa je tudi vedno več lahkih ogrodij, kot so Hibernate in Spring, ter nazadnje EJB3. Prednost lahkih ogrodij je, da razvoj ne poteka v neposredni bližini aplikacijskega strežnika, ampak aplikacijo samo namestimo nanj.

3.3 Objektno relacijske preslikave

Običajno morajo razvijalci v javanskih aplikacijah upravljati s podatki v dveh različnih strukturah. V aplikaciji dostopamo in manipuliramo s podatki, ki so predstavljeni kot javanski objekti. Ko pa želimo podatke shraniti v relacijsko podatkovno bazo jih moramo najprej pretvoriti v ustrezno obliko, primerno za shranjevanje v relacijske tabele. Upravljanje javanskih objektov in relacijskih tabel posledično zahteva poznavanje in uporabo različnih sintaks in programskih vmesnikov API. Razvijalec mora tako dvakrat modelirati iste podatke, če želi upravljati z obema sistemoma. Takšen pristop ni učinkovit, poleg tega pa močno poveča možnost nastanka napak. Zaradi naštetih težav se je pojavila tehnika objektnih relacijskih preslikav, ki ustvari učinek virtualne objektne podatkovne baze in omogoči razvijalcu, da upravlja zgolj z objekti in se ne obremenjuje z relacijsko predstavitvijo podatkov.

Ena izmed tehnik objektno relacijskih preslikav ali ORM (ang. Object Relational Mapping) je vsebovana v specifikaciji EJB 3 [15], ki jo bomo v praktičnem delu diplomske naloge uporabili pri preslikovanju objektov iz aplikacije v podatkovno bazo in obratno. Entitetna zrna v EJB 3.0 so namenjena ravno odpravljanju težav, ki nastanejo zaradi razlik med objektno ter relacijsko predstavitvijo podatkov. Entitetna zrna so navadni objekti POJO, ki so dodatno opisani z anotacijami in določajo lastnosti objekta in vrsto preslikave v podatkovno bazo. Vsa skrb glede ustrezne pretvorbe in preslikave pa je prepuščena vsebniku EJB 3 (ang. EJB 3 Container). Vsebnik EJB 3 je program, ki vsebuje storitve za celovito skrb večine sistemskih težav, ki se pojavijo pri razvoju poslovnih aplikacij. Razvijalcem tako ni več potrebno skrbeti za podrobnostih podatkovne baze, kot so tabelarične sheme, upravljanje povezav ter poznavanje specifičnih programskih vmesnikov API za dostop do

podatkovne baze, itd. ORM ponuja tudi možnost kreiranja podatkovne baze neposredno iz entitetnih zrn. Več o EJB 3 in entitetnih zrnih, ki so osnovni gradniki ogrodja za avtomatsko kreiranje grafičnih vmesnikov bomo izvedeli v nadaljevanju.

3.4 EJB 3

Preden napišemo eno samo vrstico kode, je potreben temeljit razmislek, kaj in kako bomo programirali. Predstavljajmo si, da razvijamo odjemalca za elektronsko pošto in mu želimo dodati funkcijo registriranja novega uporabnika. S tradicionalno objektno usmerjeno metodologijo bi bil seznam nalog precej dolg. Najprej bi preverili ali imamo pravice za registriranje novega uporabnika, začeli transakcijo in pazili na sočasno izvajanje oziroma izolacijo, dostopali do podatkovne baze, shranili uporabnika, dostopali do strežnika elektronske pošte, poslali potrditveno elektronsko sporočilo ter zaprli transakcijo. Ko se navidez tako preprosta naloga izkaže kot precej težka za implementacijo pridemo do spoznanja, da je bolje, da sistemu prepustimo čim več nalog in se tako osredotočimo le na ključno nalogo implementacije poslovne logike. V tem trenutku nastopi koncept EJB programiranja, kjer pri enaki nalogi pridemo do preprostejše rešitve:

Izvorna koda 3.2 EJB način programiranja

```
pseudofunkcija registracijaUporabnika(uporabnik)
{
        podatkovnabaza.shraniUporabnika(uporabnik);
        mail.posljiUporabniku(uporabnik);
}
```

Arhitektura EJB je primerna za razvoj in namestitev kompetentnih poslovnih aplikacij, ki so skalabilne oziroma prilagodljive, transakcijske ter varne. Tako narejene aplikacije lahko namestimo na katerikoli strežnik ali platformo, ki podpira specifikacijo EJB. Posplošeno, je EJB strežniški model za razvoj porazdeljenih poslovnih aplikacij [7]. To pomeni, da EJB definira model, ki združuje celoten sistem z integracijo posameznih modulov, kjer vsaka posamezna komponenta predstavlja zbirko poslovnih procesov, ki centralizirano tečejo na strežniku. EJB poleg naštetega združuje tudi ostale plati Java EE, kot so pošiljanje sporočil, transakcije, upravljanje z viri, obstojnost in spletne storitve.

3.4.1 Tipi komponent

Z uvedbo verzije EJB 3.0 pri razvoju aplikacij POJO ni potrebna razširitev, implementacija ali kakršna koli referenca na EJB. To pomeni, da lahko sedaj kreiramo razredno hierarhijo bolj prosto ter uporabimo kreirane objekte tudi v okolju, ki ne vsebuje EJB. Objekti POJO postanejo EJB šele, ko je sestavljen, nameščen in do njih dostopa vsebnik. Vsebnik ponavadi zagotavlja storitve za zunanje aplikacije. V okolju EJB pa je odgovoren za oskrbo objektov POJO s storitvami EJB in predstavitvijo njihovega vedenja. Poznamo tri vedenjske tipe objektov POJO, ki jih imenujemo tipi komponent. To so sejna, sporočilna in entitetna zrna. Za razvijalca poznavanje njihove semantične implementacije ni potrebno, pomembno pa je dobro poznavanje vedenjskih karakteristik.

3.4.1.1 Strežniški tipi komponent (Sejna in sporočilna zrna)

Strežniški tipi komponent se nahajajo izključno na strežniku, odjemalec pa z njimi komunicira indirektno. Poznamo dve osnovni vrsti strežniških komponent: Sejna zrna, na katere se sklicujejo odjemalci, in sporočilna zrna, ki delujejo kot poslušalci dogodkov.

Sejna zrna

Če je EJB slovnica, so sejna zrna glagoli. Oni sprejemajo ukrepe in vsebujejo poslovne metode. Odjemalec ne dostopa do EJB direktno, kar omogoča vsebniku, da izvaja najrazličnejše naloge, preden zahteva pride do želene metode. Ravno ta tehnika omogoča, da odjemalcu niso potrebne informacije o lokaciji strežnika, nadzorni politiki ali upravljanju z viri. Odjemalec deluje nad EJB in se sklicuje na nadomestno (ang. proxy) referenco, ki delegira zahteve vsebniku in vrne ustrezen odgovor. Tukaj se srečamo še z dvema tipoma sejnih zrn. Zrna, ki ohranjajo stanje in jih uporabimo, ko si želimo zapomniti sejo, ki nam omogoča pogovor z odjemalcem, in zrna, ki stanja ne ohranjajo in jih uporabimo za enkraten dostop ter s tem zmanjšamo porabo računalniškega spomina.

Sporočilna zrna

Sporočilna zrna so poslušalci, ki prejemajo sporočila in jih bodisi obravnavajo neposredno ali posredujejo v nadaljnje procesiranje drugim komponentam EJB. Pogovornega stanja si ne zapomnijo. Delovanje je asinhrono, kar pomeni, da pošiljatelju sporočila ni potrebno čakati na odgovor.

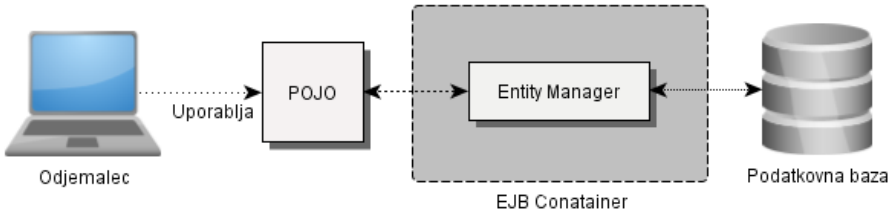
3.4.1.2 Entitetna zrna

Preden začnem z razlago konceptov entitetnih zrn je nujno potrebno omeniti, da so entitetna zrna ključni gradniki ogrodja za avtomatsko kreiranje grafičnih vmesnikov. So tiste komponente, ki jih bomo označevali z anotacijami. To pomeni, da bomo opisovali poslovno logiko in vlogo posameznega entitetnega zrna v naši aplikaciji, kaj predstavlja, katere lastnosti ima in kako je povezano z drugimi entitetnimi zrni. Na podlagi teh označb se bodo avtomatično zgradili grafični vmesniki, ki bodo omogočali obvladovanje podatkov.

Ena izmed ključnih prednosti EJB je možnost kreiranja entitetnih zrn. Če smo rekli, da so sejna zrna glagoli, potem so entitetna zrna samostalniki. Gre za komponente, ki predstavljajo obstojne objekte v sistemu za shranjevanje podatkov, ki bo v našem primeru podatkovna baza. Cilj entitetnih zrn je, da izrazijo objektni pogled na vire, shranjene v sistemu za upravljanje relacijske podatkovne baze. To pomeni, da je osnova za delovanje EJB objektno relacijsko preslikovanje, ki sem ga omenil predhodno. Entitetna zrna predstavljajo podatke, ki bi jih radi shranili v podatkovno bazo, kot so na primer informacije o bančnih računih (številka računa, stanje), podatki o osebah (ime, oddelek, plača), itd. Sedaj vidimo, da entitetna zrna vsebujejo ključne poslovne podatke (poslovne informacije), ki jih želimo uporabljati v poslovni aplikaciji in ne vsebujejo metod za modeliranje procesov ali delovnega toka aplikacije, kot to vsebujejo sejna zrna. Za razliko od sejnih zrn imajo entitetna zrna tudi identiteto in stanje, vidno odjemalcu. Njihova življenjska doba je lahko povsem neodvisna od življenjske dobe odjemalčeve aplikacije. S poznavanjem njihove identitete pa lahko dostopamo do posameznih instanc entitetnega zrna, jih med seboj primerjamo, delimo z drugimi odjemalci, itd. Entitetna zrna se znajo obstojno preslikati v podatkovno bazo in so fizični deli poslovne aplikacije. To pomeni, da lahko preživijo kritične odpovedi, kot so sesutje strežnika ali podatkovne baze. To je možno zato, ker so entitetna zrna le predstavitev podatkov shranjenih v obstojnih shranjevalnih medijih (npr. podatkovna baza), iz katerih lahko rekonstruiramo podatke v primeru izgube [5].

Tako kot sejna zrna so tudi entitetna zrna enaka objektom POJO, ki postanejo obvladljivi objekti šele, ko so povezani s storitvijo *javax.persistence.-EntityManager,* ki spremlja spremembe stanja in te spremembe, če je potrebno, sinhronizira s podatkovno bazo, kar prikazuje slika 3.2. Da se vse uspešno prenese v podatkovno bazo skrbi knjižnica JPA, na katero gledamo kot višji nivo JDBC (ang. Java Database Connectivity). Če označimo podatke v podatkovni bazi kot objekte, lahko razvijalec vnaša, briše, spreminja podatke brez uporabe

jezika SQL (ang. Structured Query Language).



Slika 3.2: Uporaba entitetnega upravljalca pri povezovanju POJO objektnega stanja in obstojne relacijske podatkovne baze

Tako označevanje objektov oziroma entitetnih zrn v specifikaciji EJB imenujemo preslikovanje z označevanjem JPA.

3.4.2 Preslikovanje z označevanjem JPA

Preslikovanje se izvaja glede na vrste označb, ki so definirane v JDK 5.0 (ang. Java Development Kit) [19] in jih delimo v dve kategoriji. Anotacije za logično preslikovanje, ki omogočajo opisovanje objektnega modela in opisovanje povezav med razredi ter anotacije za fizično preslikovanje, s katerimi opisujemo fizično shemo, tabele, vrstice, indekse, itd. [17] Nekaj anotacij bomo predstavili v nadaljevanju, več o preslikovanju z uporabo anotacij pa lahko najdemo v dokumentaciji Java EE v paketu *javax.persistence.**.[1]

@Entity

Vsak POJO razred je lahko entitetno zrno, če ga deklariramo z anotacijo *@Entity*. Ko je objekt deklariran kot entitetno zrno, mu lahko dodamo tudi druge označbe in ga z njimi opisujemo. Na primer z anotacijo *@Id* deklariramo identifikatorja entitetnega zrna, kar prikazuje izvorna koda 3.3.

@Table

Razred *Oseba* je kot entitetno zrno sedaj preslikano v tabelo *Oseba* v podatkovni bazi, njen primarni ključ pa je EMŠO številka. Če bi želeli dodatno opisati oziramo definirati tabelo, katalog ali shemo entitetnega zrna, lahko to storimo z anotacijo *@Table*. Od posamezne anotacije je odvisno ali omogoča dodajanje različnih atributov. V primeru *@Table* lahko na primeru izvorne kode 3.4 vidimo, kako so vneseni dodatni atributi.

[1] http://download.oracle.com/javaee/5/api/javax/persistence/package-summary.html

Izvorna koda 3.3 Primer uporabe anotacije @Entity

```
@Entity
public class Oseba implements Serializable {
    Integer emso;

    @Id
    public Long getEmso() { return emso; }
    public void setEmso(Integer emso)
        { this.emso = emso; }
}
```

Izvorna koda 3.4 Primer uporabe anotacije @Table

```
@Entity
@Table(name="tbl_oseba",
uniqueConstraints =
{@UniqueConstraint(columnNames={"ime", "priimek"})})
public class Oseba implements Serializable {
...
```

Preslikovanje enostavnih komponent

@Transient

Vsaka nestatična metoda ali polje entitetnega zrna je obstojna, razen če je ne označimo z anotacijo @Transient. To pomeni, da bo ignorirana s strani entitetnega upravljalca. V primeru, da anotacija *@Transient* ni nastavljena, ima entitetno zrno privzeto nastavitev *@Basic*, kar pomeni, da bo objekt obstojen in shranjen v podatkovni bazi. Takrat lahko z dodatnimi atributi označimo tudi strategijo pridobivanja objektov iz podatkovne baze, ki je lahko takojšnja (ang. Eager) ali z zakasnitvijo (ang. Lazy). Pri takojšnji gre za takojšnje pridobivanje vseh vrednosti, pri strategiji z zakasnitvijo pa za pridobivanje vrednosti šele, ko prvič dostopamo do metode ali polja. Primer anotacije: *@Basic(fetch = FetchType.EAGER* ali *FetchType.LAZY)*

@Column

Z anotacijo @Column lahko opišemo vrstice, ki jih uporabljamo pri preslikovanju. Zopet lahko definiramo različne atribute kot so ime, unikatnost,

privzeto vrednost, velikost, število decimalnih mest, decimalno lestvico, itd., kar nam prikazuje izvorna koda 3.5.

Izvorna koda 3.5 Primer uporabe anotacije @Column

```
@Entity
public class Oseba implements Serializable {
...
@Column(updatable = false, name = "oseba_ime",
nullable = false, length=50)
public String getIme() { ... }
```

S sledečimi anotacijami opišemo povezave med entitetnimi zrni:

@OneToOne

Pri povezavi ena proti ena lahko entitetna zrna povežemo z anotacijo *@OneToOne.* Poznamo tri možnosti povezave ena proti ena. Lahko si entiteti delita primarni ključ, lahko ena izmed entitet hrani tuj ključ, ali pa imamo povezavo z vmesno tabelo. Vsako izmed možnosti lahko opišemo z ustrezno anotacijo. Izsek izvorne kode 3.6 prikazuje povezavo s tujim ključem.

Izvorna koda 3.6 Primer povezave 1:1

```
@Entity
public class Oseba implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="potniList_fk")
    public PotniList getPotniList() {
        ...
    }
@Entity
public class PotniList implements Serializable {
    @OneToOne(mappedBy = "potniList")
    public Oseba getOsebaPotniList() {
    ...
}
```

Z anotacijami lahko poleg omenjene povezave 1:1 preslikamo tudi povezave kot so ena proti mnogo, kar naredimo z anotacijo *@OneToMany*, mnogo proti

ena, z anotacijo *@ManyToOne*, in mnogo proti mnogo, z anotacijo *@ManyToMany*. V specifikaciji EJB 3 je omogočeno tudi preslikovanje zbirk, seznamov in množic, ki jih lahko nato poljubno urejamo z anotacijo *@OrderBy*. Tako lahko na primer seznam oseb uredimo po imenu ali priimku. Kot lahko vidimo EJB 3 podpira številne funkcije, ki jih lahko označimo z anotacijami, kot so preslikovanje sestavljenih primarnih in tujih ključev, preslikovanje poizvedb, itd.

3.4.3 Metode entitetnega upravljalca

Če želimo dejansko izvajati funkcije ustvarjanja, branja, urejanja in brisanja podatkov v EJB 3 je najbolj enostaven način uporaba entitetnega upravljalca, ki vsebuje metode, s katerimi lahko izvajamo operacije CRUD [7]. Tako kot pri preslikovanju z označevanjem gre zopet za operacije definirane v specifikaciji JPA. Ko smo ustrezno označili objekte z anotacijami JPA, ki nam bodo omogočile preslikavo objekta v podatkovno bazo, je sedaj potreben klic na ustrezno metodo, ki bo preslikavo izvedla.

persist()

S klicem metode *persist()* preslikamo entiteto, ki do sedaj še ni bila ustvarjena, v podatkovno bazo. Pred klicem je nujno potrebno inicializiati entitetno zrno nato pa s klicem metode *EntityManager.persist()* zapišemo entiteto v podatkovno bazo. Primer klica metode *persist()* nam prikazuje izvorna koda 3.7.

Izvorna koda 3.7 Primer klica metode persist()

```
@PersistenceContext
protected EntityManager manager;

public Stranka save(Stranka entity) {
        manager.persist(entity);
}
```

Ko je metoda klicana, entitetni upravljalec zgradi čakalno vrsto instanc objekta *Stranka*, ki čakajo na vnos v podatkovno bazo. Kdaj se dejansko izvede vnos v podatkovno bazo, je odvisno od večih faktorjev. Ponavadi je klic metode izveden znotraj transakcije, kjer se vnos zgodi ali nemudoma ali ob koncu transakcije. To pomeni, da z entiteto še vedno lahko upravljamo

in jo spreminjamo, spremembe pa se bodo zapisale ob koncu transakcije. V primeru, da je entiteta povezana z drugimi entitetami in so razmerja ustrezno opisana, se bo klic metode nanašal tudi na referenčne entitete. To pomeni, da če imamo definirana ustrezna pravila povezav med entitetami se lahko z ustvarjanjem ene entitete ustvari ali uredi tudi entiteta, ki je na njo povezana. Klic metode *persist()* omogoča tudi nekatere druge funkcije, kot je avtomatsko generiranje primarnega ključa.

find()

Entitetni upravljalec omogoča dva načina iskanja objektov v podatkovni bazi. Prvi klasični način je s pisanjem in izvajanjem poizvedbe, drugi pa s preprosto metodo *EntityManager*, ki poišče entiteto glede na primarni ključ. Ta metoda se imenuje *find()*, ki kot vhodna parametra potrebuje razred entitete in instanco primarnega ključa entitete. Izgled metode prikazuje izvorna koda 3.8.

Izvorna koda 3.8 Metoda find()

```
find(Class<T> Stranka, Object primarniKljuc);
```

merge()

JPA specifikacija omogoča združevanje sprememb stanja na entiteti in preslikovanje v podatkovno bazo tudi z uporabo metode *merge()*. Za razliko od metode *persist()*, kjer lahko entiteto urejamo po klicu metode, to pri metodi *merge()* ni možno. Če spreminjamo stanje entitete po klicu metode *merge()*, te spremembe ne bodo zapisane v podatkovno bazo razen, če zopet ne naredimo klic nanjo.

Za upravljanje z entitetami obstajajo še nekatere druge metode definirane v entitetnem upravljalcu, kot so *remove()*, *refresh()*, *clear()*.

remove() - Izbriše entiteto iz podatkovne baze

refresh() - Osveži entiteto (vrne stanje iz podatkovne baze in prekopira nastale spremembe)

clear() - Sprosti instanco entitete tako, da je ni več možno urejati

3.4.4 Storitve vsebnika

Storitve vsebnika v okolju EJB skrbijo za injeciranje odvisnosti, sočasnost dostopa različnih odjemalcev do enakih virov, pridobivanje in shranjevanje in-

stanc (instance niso dosegljive za procesiranje dokler se določene zahteve ne končajo), upravljanje s transakcijami (atomarnost, konsistentnost, izolacija, trajnost), varnost in dostop do posameznih storitev glede na vlogo uporabnika, interoperabilnost oziroma komunikacijo z zunanjimi programi, integracijo okolij (združevanje ogrodij ostalih izvajalnih okolij in programskih vmesnikov API, kot so Java Transaction Service, Java Persistance API, JNDI, Security Service, Web Service), itd. Ker se v večini primerov, razvijalcu ni potrebno ukvarjati z naštetimi nalogami se v podrobnosti delovanja vsebnika ne bomo spuščali.

Poglavje 4

Ogrodje

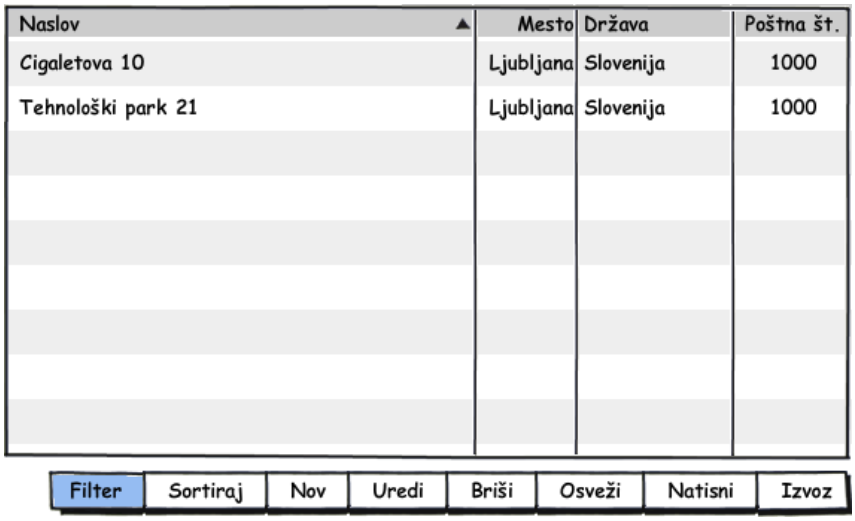
V predhodnih poglavjih smo se spoznali z vsemi ključnimi gradniki ogrodja za avtomatsko izdelavo grafičnih vmesnikov. Spoznali smo kaj so to objekti POJO in EJB in kako lahko take objekte opišemo z označbami in tako pridobimo številne funkcionalnosti, ki jih omogoča okolje Java EE. Z dobrim poznavanjem vseh teh tehnologij je prišla ideja za izdelavo ogrodja, kjer bo možno z opisovanjem strežniških komponent poleg vseh funkcionalnosti, kot so komunikacija in preslikovanje objektov v podatkovno bazo, tudi avtomatsko kreirati grafične vmesnike v različnih izvajalnih okoljih.

To pomeni, da je nastalo ogrodje, ki ga uporabljamo tako za razvoj spletnih, namiznih, kot tudi mobilnih aplikacij in deluje na osnovi opisovanja, vsem trem aplikacijam skupne, strežniške kode. Za opisovanje objektov so bile za potrebe ogrodja za avtomatsko kreiranje grafičnih vmesnikov razvite dodatne anotacije. V primeru, da je entitetno zrno opisano z eno ali več dodatno razvitimi anotacijami, ogrodje ustrezno interpretira to anotacijo glede na izvajalno okolje, ter kreira grafični vmesnik za upravljanje s tem označenim entitetnim zrnom. Dodatne anotacije za opisovanje entitetnih zrn bomo opisali v nadaljevanju, najprej pa si poglejmo kakšne maske si želimo zgraditi z uporabo našega ogrodja.

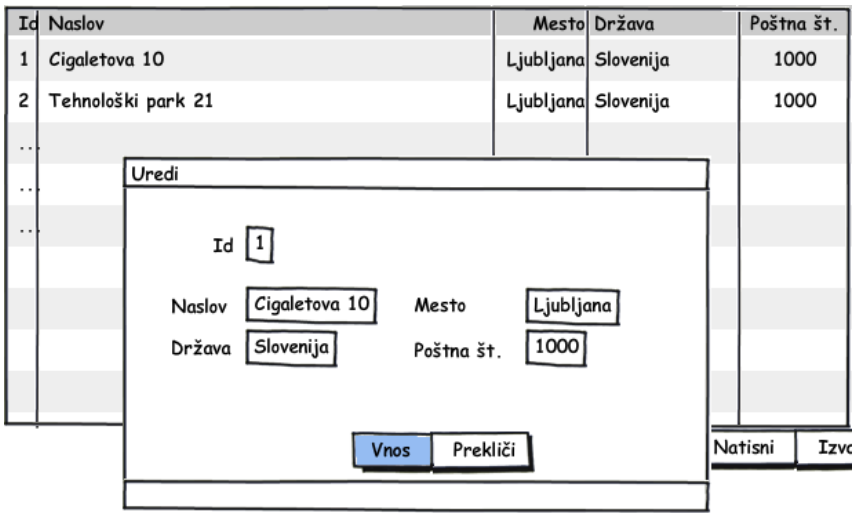
4.1 Primeri mask

Ker gre pri uporabi ogrodja za kreiranje grafičnih vmesnikov za izključno poslovne aplikacije, je vredno razmisliti kakšne grafične vmesnike si želimo in potrebujemo za učinkovito upravljanje s poslovnimi podatki. Najbolj osnovna maska, ki jo potrebujemo je maska s šifranti. Pri njej gre za prikazovanje podatkov posamezne tabele v podatkovni bazi, ter urejanje, razvrščanje, do-

dajanje, brisanje, izvažanje podatkov, filtriranje, itd. Osnovno masko s šifranti prikazujeta sliki 4.1 in 4.2.



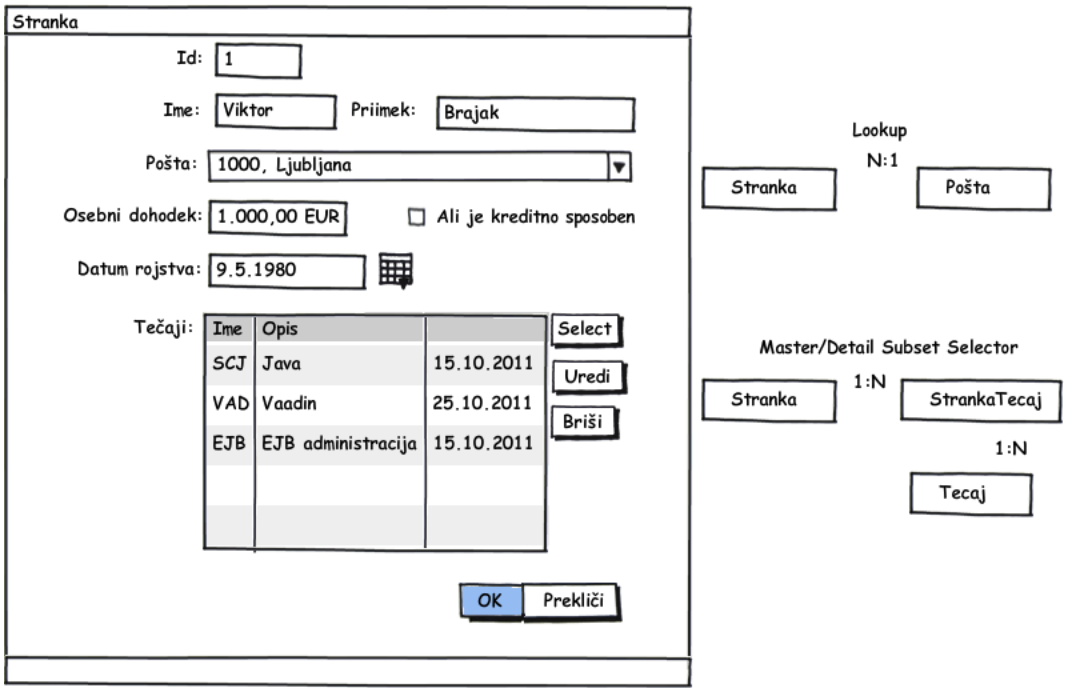
Slika 4.1: Prikaz maske s šifranti



Slika 4.2: Prikaz maske za urejanje s šifranti

Ker se tabele s podatki v podatkovni bazi referencirajo ena na drugo in so med seboj povezane, potrebujemo tudi kompleksnejše maske od mask s šifranti. Potrebujemo maske, ki bodo omogočale urejanje podatkov med tabelami povezanimi z najrazličnejšimi povezavami kot so ena proti ena, ena

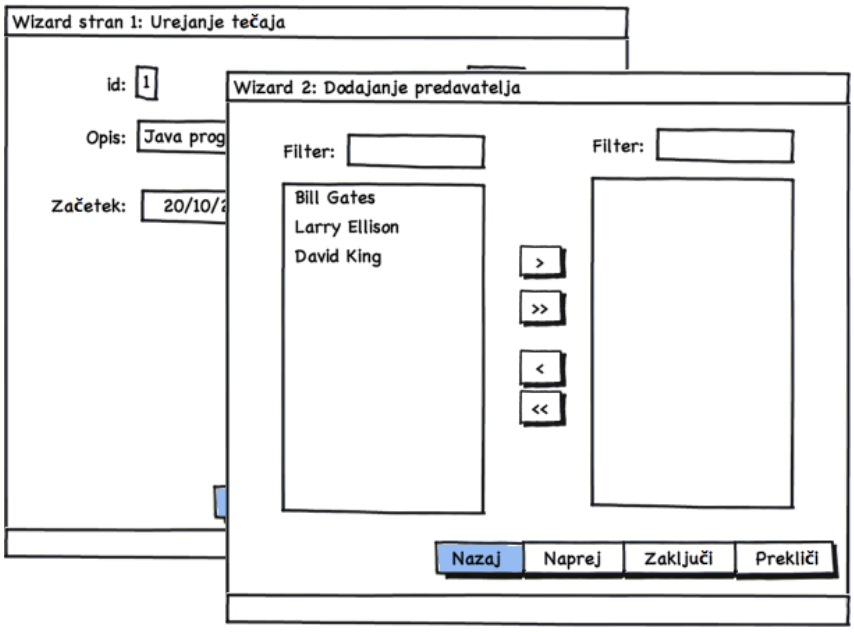
proti mnogo, mnogo proti ena, mnogo proti mnogo. Kompleksnejše maske nam omogočajo, da urejamo podatke na večih tabelah hkrati, aplikacija pa skrbi za omejitve med povezavami. V večini primerov vsebujejo tudi gradnike, ki omogočajo enostavnejšo in pravilnejšo uporabo aplikacije. Taki gradniki so lahko najrazličnejši izbirni meniji, validatorji pri vnosu podatkov, potrditvena polja, itd. Primer kompleksnih mask vidimo na sliki 4.3, kjer vidimo primer gradnika *Lookup* in *Master/Detail*. Pri gradniku *Lookup* gre za izbirni meni, pri katerem želimo stranki dodati natanko eno pošto. To pomeni, da gre za relacijo mnogo proti ena, kjer ima lahko stranka natanko eno pošto in pošta večje število strank. Gradnik *Master/Detail* pa prikazuje razmerje mnogo proti mnogo, kjer lahko ena stranka obiskuje več tečajev in en tečaj lahko obiskujejo različne stranke.



Slika 4.3: Prikaz kompleksne maske za urejanje podatkov o strankah

Da bi omejili in poenostavili uporabo poslovnih aplikacij, se pri uporabi poslovnih aplikacij velikokrat uporabljajo čarovniki. Čarovniki omejijo uporabo aplikacije s tem, da nam onemogočijo nadaljevanje uporabe, če predhodne zahteve niso bile izpolnjene, in tako zagotovijo pravilni vrstni red uporabe. Izgled preprostega čarovnika v poslovni aplikaciji nam prikazuje slika 4.4. Na drugem oknu čarovnika vidimo še eno vrsto izbirnega menija, ki ga imenujemo *Subset-*

Selector in ga uporabimo v razmerjih mnogo proti mnogo.



Slika 4.4: Prikaz urejanja tečaja in dodajanja predavatelja s čarovnikom

4.2 Predstavitev anotacij in njihova uporaba

Če pri razvoju poslovne aplikacije uporabljamo ogrodje za avtomatsko kreiranje grafičnih vmesnikov dosežemo, da se predhodno omenjene maske avtomatsko generirajo v različnih izvajalnih okoljih. Da zna ogrodje pravilno interpretirati in narisati maske, pa je potrebno opisati strežniško kodo. Strežniško kodo opišemo z anotacijami ogrodja za avtomatsko kreiranje grafičnih vmesnikov. Z njimi opišemo entitetne razrede in njihove metode, ki so skupni vsem aplikacijam, ne glede na to v katerem okolju se izvajajo.

Anotacije ogrodja delimo v tri skupine:

1. Anotacije *Editor*
    - *@FieldPresentation*
    - *@EditorLayout*
    - *@EditorLayouts*

- *@EditorPresentation*
- *@EditorPresentations*

2. Anotacije *Table*

- *@ColumnPresentation*
- *@ExportPresentation*
- *@TablePresentation*
- *@TablePresentations*

3. Anotacije *Lookup*

- *@LookupPresentation*
- *@LookupPresentations*
- *@LookupMethod*

V nadaljevanju bom opisal vsako skupino, posamezne anotacije v skupini, njihove parametre, ter kako se jih uporabi za opis enitetnih zrn. S slikami pa bo prikazano kakšne maske nastanejo na podlagi opisov entitetnih zrn.

4.2.1 Anotacije *Editor*

Anotacije *Editor* so namenjene urejanju objektov. To pomeni, da lahko objekte in vrednosti posameznih atributov dodajamo, spreminjamo in brišemo. Objekte lahko z anotacijami opišemo na tri načine. Prvi način je samo z uporabo *@FieldPresentation* anotacije, v drugih dveh primerih pa poleg anotacije *@FieldPresentation* uporabimo še *@EditorLayout* ali *@EditorPresentation*.

@FieldPresentation

Anotacija *@FieldPresentation* se uporablja za prikaz polj na maski. Z njo povemo kako naj se obnašajo atributi objekta na maski, nad katerimi to anotacijo uporabljamo. Z njo anotiramo metode *get()*, ki kličejo atribute objektov.

Parametri:

- *order* (int)

  Določa vrstni red prikazovanja na formi. Prikazujejo se v zaporedju od elementa z najnižjo do tistega z najvišjo vrednostjo *order*. Privzeta vrednost je najvišja možna vrednost, ki jo lahko zavzame tip *Integer*.

- *level* (LevelType)

  Določi, na katerem nivoju naj se lastnost objekta prikazuje v editorju. Možnosti so SIMPLE (vidna je, ko je editor v kateremkoli nivoju), NORMAL (vidna je, ko je editor v nivojih NORMAL in DETAILED) in DETAILED (vidna je samo takrat, ko je editor v nivoju DETAILED). Privzeto je NORMAL.

- *type* (FieldType)

  Določi tip uporabljenega polja, kjer imamo naslednje možnosti:

  - NORMAL Uporabi se navadno tekstovno polje. To je tudi privzeta vrednost.
  - MULTILINE Uporabi se tekstovno polje z več vrsticami.
  - LOOKUP Iskalnik oziroma meni *Lookup* za mnogo proti ena relacije.
  - LOOKUP_SIMPLE Enako kot pri LOOKUP, le da namesto iskalnika uporabimo element ,,combobox“.
  - LOOKUP_MANY Iskalnik za ena proti mnogo relacije.
  - LOOKUP_MANYTOMANY Iskalnik za mnogo proti mnogo relacije.

- *groupName* (String)

  Določi v kateri skupini na formi je to polje. Skupine na formi so urejene po imenu skupine. Privzeta skupina je *default*.

- *required* (boolean)

  Določi ali je polje obvezno. Torej ali mora uporabnik obvezno izpolniti polje za nadaljevanje. Privzeta vrednost je *false*.

- *id* (boolean)

  Določi ali je lastnost objekta tudi primarni ključ. Tipično se tak atribut obnaša drugače od ostalih atributov objekta. Če je objekt že anotiran z anotacijo @Id (javax.persistance.Id) se parameter ignorira. Privzeta vrednost je *false*.

- *maxLength* (int)

  Določa, koliko znakov lahko zavzema vrednost polja. Privzeta vrednost je negativna, kar pomeni, da ni omejitve.

- *lookupName* (String)

  Kreirani izbirni meni lahko poimenujemo.

- *lookupClass* (Class)

  Določa, kateri razred se naj uporabi v izbirnem meniju. Uporablja se pri kreiranju tipa LOOKUP.

- *manyToManyClass* (Class)

  Uporablja se pri LOOKUP_MANYTOMANY. S tem parametrom določimo vmesno entiteto ki jo iskalnik uporablja.

- *manyToManyProperty* (String)

  Uporablja se pri LOOKUP_MANYTOMANY. Z njim določimo atribut vmesne entitete, ki jo iskalnik po izbiri napolni.

- *manyToManyBackProperty* (String)

  S tem parametrom določimo kateri atribut vmesne entitete se uporabi kot prvi del razmerja. Drugi se določi z *manyToManyProperty*.

- *mapperClass* (Class)

  Določa, kateri razred se uporabi za preslikavo med lastnostjo objekta in nizom, ki ga prikazujemo v polju. Tipično ga uporabljamo za prikaz spremenljivk tipa *Date* in *Timestamp* (obstajajo že razredi *UtilDateMapper*, *SQLDateMapper* in *TimestampMapper*) ter števil (*NumberMapper*).

- *mapperParams* (String[])

  Parametri, ki se uporabijo kot konstruktor razreda *Mapper*. Uporablja naj se takrat, ko je določen *mapperClass*. Privzeto je seznam prazen.

- *validatorClass* (Class)

  Določa, kateri razred naj se uporabi za validiranje polja. Uporablja se lahko katerikoli validator, ki v konstruktor sprejema tip *String*. Obstoječi validatorji so *NumberInRangeValidator*, *DateInRangeValidator* in *RegexValidator*.

- *validatorParams* (String[])

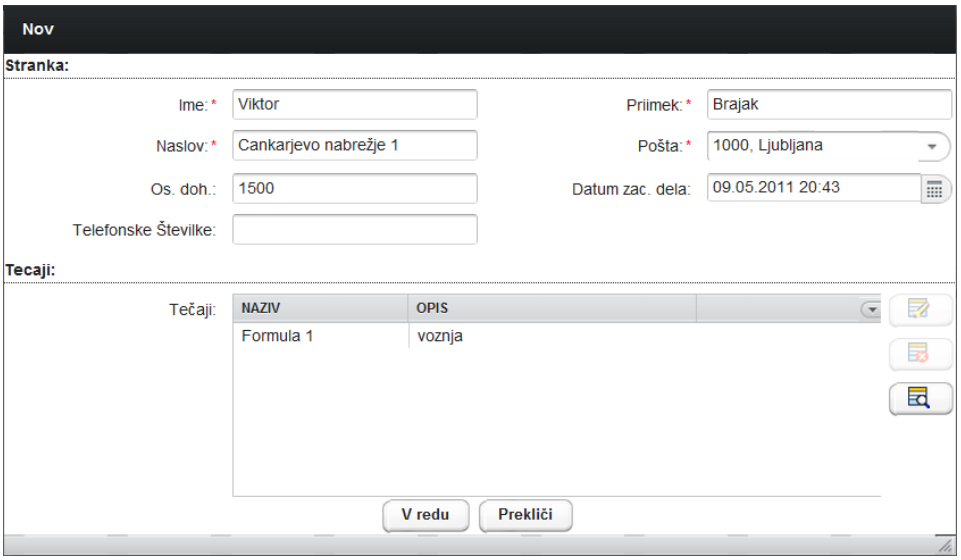
  String parametri, ki se uporabijo kot konstruktor razreda *Validator*.

- *manyLookupActions* (ManyLookupActions[])

  Določa, kateri ukazi naj se izvajajo nad lastnostjo objekta. Uporabimo anotacijo @ManyLookupActions, ki vsebuje naslednje lastnosti:

  – *add* (boolean) Omogočeno dodajanje. Privzeto *false*.
  – *edit* (boolean) Omogočeno urejanje. Privzeto je *false*.
  – *select* (boolean) Omogočeno poizvedovanje. Privzeto je *true*.
  – *remove* (boolean) Omogočeno brisanje. Privzeto je *true*.

Kot primer si poglejmo urejanje objekta *Stranka*. Masko prikazuje slika 4.5.



Slika 4.5: Prikaz maske za urejanje objekta Stranka

Kot lahko vidimo imamo na maski nekaj preprostih polj (Ime, Priimek, Naslov, Osebni dohodek). Poglejmo si primer uporabe anotacije @FieldPresentation za priimek, ki ga prikazuje izvorna koda 4.1.

Ker nimamo podanega tipa, se uporabi privzeta vrednost, v tem primeru je to tekstovno polje. Parameter *order* ima vrednost 2, kar pomeni, da se bo polje na maski pojavilo kot drugo po vrsti. Zaradi parametra *maxLength*=50 lahko v polje vpišemo samo 50 znakov. Parameter *required* je nastavljen na

Izvorna koda 4.1 Primer uporabe anotacije @FieldPresentation za polje Priimek

```
@FieldPresentation(order = 2, groupName = "stranka_id",
maxLength = 50,required=true)
public String getPriimek(){return this.priimek;}
```

vrednost *true*, kar pomeni, da aplikacija avtomatsko preveri, ali polje vsebuje vrednost ali ne. S parametrom *groupName* združujemo zgornja polja na maski v isto skupino.

Poleg enostavnih polj, s katerimi določimo vrednosti enostavnih lastnosti objekta (Integer, String, itd.), imamo na maski tudi iskalnike, ki so namenjeni določanju vrednosti lastnosti objekta, ki so lahko tudi objekti. Tak primer lahko vidimo pri izbiri pošte, kjer imamo razmerje mnogo proti ena, ali pri izbiri tečaja, kjer imamo razmerje mnogo proti mnogo. Izvorna koda 4.2 prikazuje uporabo anotacije za prikaz razmerja mnogo proti ena.

Izvorna koda 4.2 Primer uporabe anotacije @FieldPresentation pri izbiri pošte

```
@FieldPresentation(order = 4, groupName = "stranka_id",
type=FieldType.LOOKUP_SIMPLE, lookupClass=Posta.class,
required=true, level=LevelType.SIMPLE)
public Posta getPosta(){return this.posta;}
```

Kot vidimo, je vrednost parametra *type* LOOKUP_SIMPLE, kar pomeni, da dobimo enostavni izbirni meni. Če želimo meniju določiti vrednosti pa moramo določiti tudi parameter *lookupClass*, ki se sklicuje na objekt *Posta.class*. Ker se sklicujemo na objekt *Posta,* je potrebno anotirati razred *Posta*. Označimo ga z anotacijo @LookupPresentation, ki pove, katere podatke naj prikaže meni. O tej anotaciji bomo več povedali v nadaljevanju.

Pri izbiri tečaja gre za razmerje mnogo proti mnogo, kjer imamo vmesno tabelo *StrankaTecaj*. V razredu *Stranka* zato označimo metodo, ki kliče instanco objekta *StrankaTecaj*. Izvorna koda 4.3 prikazuje primer izbire mnogo proti mnogo.

Uporabljen tip je LOOKUP_MANYTOMANY. Vmesni objekt *StrankaTecaj* povezuje objekta *Stranka* in *Tecaj*. Zato je potrebno določiti *manyToManyBackProperty* in *manyToManyProperty*. Določiti moramo še *manyToManyClass* in *lookupClass*. Prvega zato, da iskalnik ve kateri je vmesni objekt, ki

Izvorna koda 4.3 Primer uporabe anotacije @FieldPresentation pri izbiri tečaja

```
@FieldPresentation(order = 9,groupName = "stranka_tecaj",
lookupName="tecajiZaStranke",
manyToManyBackProperty="stranka",
manyToManyProperty="tecaj",
type=FieldType.LOOKUP_MANYTOMANY,
lookupActions=Actions.add_edit_remove,
manyToManyClass=StrankaTecaj.class,
lookupClass=Tecaj.class)
public Set<StrankaTecaj> getStrankaTecajs()
{return this.strankaTecajs;}
```

ga urejamo, drugega pa zato, da ve kje iskati informacije o tečajih.

Vsi našteti primeri so vidni na sliki 4.5.

@EditorLayout

@EditorLayout je anotacija, s katero določimo posebno razporeditev urejevalne maske. To pomeni, da lahko atribute objekta poljubno prestavljamo po formi. Z njo anotiramo razred.

Parametri:

- *name* (String)

  Določa, katera predstavitev naj se uporablja za prikaz v editorju. Ta možnost se uporabi, če ne uporabimo anotacije @EditorPresentation nad razredom, ampak samo anotacije na metodah kot je @FieldPresentation. Imamo naslednje možnosti:

  - *default_simple* (DefaultPresentation.DEFAULT_SIMPLE) Prikazujejo se vsa polja z nivojem SIMPLE.
  - *default* (DefaultPresentation.DEFAULT_NORMAL) Prikazujejo se vsa polja z nivojem NORMAL ali SIMPLE.
  - *default_detail* (DefaultPresentation.DEFAULT_DETAILED) Prikazujejo se vsa polja.

- *customLayout* (String)

  Določa ime razporeditve, ki naj se uporabi.

- *labelPosition* (LabelPosition)

  Določa, kako so labele postavljene zraven polj. Možnosti so:

  - VERTICAL Labela je nad poljem.
  - HORIZONTAL Labela je na levi strani polja.
  - NONE Labela ni prikazana.

  Prizveta vrednost je HORIZONTAL.

@EditorLayouts

Anotacija @EditorLayouts predstavlja seznam @EditorLayout anotacij. Za en objekt imamo lahko več različnih razporeditev urejevalnih mask.

Parametri:

- *value* (EditorLayout[])

@EditorPresentation

Anotacija @EditorPresentation določa način predstavitve urejevalne maske. Uporablja se v posebnih primerih, ko želimo določiti, katere atribute objekta bomo urejali v njej. Z njo anotiramo razred.

Parametri:

- *name* (String)

  Določi ime predstavitve editorja. Na strani odjemalca pa se določi na kakšen način bo izvedel predstavitev s tem imenom.

- *properties* (String [])

  Lastnosti, ki se uporabljajo pri tej predstavitvi editorja. Če niso določene, bo uporabljena predstavitev editorja na nivoju NORMAL. To pomeni, da se bodo prikazale lastnosti objekta, ki imajo ustrezne anotacije @Field-Presentation.

- *excludedProperties* (String[])

  Lastnosti, ki naj bodo izključene iz predstavitve editorja.

@EditorPresentations

Anotacija @EditorPresentations predstavlja seznam anotacij @EditorPresentation. Uporabimo jo v primerih, ko ima en objekt več različnih editorjev.

4.2.2 Anotacije *Table*

Anotacije *Table* so namenjene predstavitvi podatkov v tabelah. Anotacija @ColumnPresentation je namenjena posameznim stolpcem, ostale anotacije pa celi tabeli.

@ColumnPresentation

Anotacija @ColumnPresentation atributu objekta določi, kako naj bo prikazan v stolpcu tabele. Z njo anotiramo metode.

Parametri:

- *order* (double)

  Določi pozicijo stolpca v tabeli. Tabela prikazuje stolpce od leve proti desni, v zaporedju od elementa z najnižjo do tistega z najvišjo vrednostjo parametra *order*. Privzeta vrednost je najvišja možna vrednost, ki jo lahko zavzame *Integer*.

- *level* (LevelType)

  Določi na katerem nivoju naj se stolpec prikazuje v tabeli. Možnosti so SIMPLE (stolpec je viden, ko je tabela v kateremkoli nivoju), NORMAL (stolpec je viden ko je tabela v nivojih NORMAL in DETAILED) in DETAILED (stolpec je viden samo takrat ko je tabela v nivoju DETAILED). Privzeto je stolpec v stanju NORMAL.

- *folowOnLevel* (LevelType)

  Določi na katerem nivoju se prikazujejo ugnezdeni stolpci. Uporabljajo se isti nivoji kot pri parametru *level*. Tu je privzeta vrednost SIMPLE (prikažejo se vsi stolpci).

- *width* (int)

  Določa širino stolpca v tabeli. Privzeta vrednost je -1. Ta vrednost pomeni, da se širina stolpca določa avtomatsko glede na število stolpcev.

- *limitLength* (int)

  Določa, koliko znakov naj se prikazuje v stolpcu. Privzeta vrednost je -1. Ta vrednost pomeni, da naj se prikažejo vsi znaki.

- *align* (ColumnAlignType)

Določa poravnavo besedila znotraj stolpca. Možne vrednosti so tipa *ColumnAlignType*. In sicer LEFT (leva poravnava), RIGHT (desna poravnava), CENTER (sredinska poravnava) in DEFAULT (privzeta poravnanost). Slednja je tudi privzeta vrednost parametra.

- *filterable* (boolean)

  Določa, ali naj bo nad stolpcem omogočeno filtriranje. Privzeta vrednost je *true*.

- *printable* (boolean)

  Določa, ali naj bo stolpec vključen pri tiskanju ali ne. Privzeta vrednost je *true*.

- *id* (boolean)

  Določa, ali je lastnost objekta tudi primarni ključ. Tipično se tak atribut obnaša drugače od ostalih atributov objekta. Če je objekt že anotiran z anotacijo @Id (javax.persistance.Id) se parameter ignorira. Privzeta vrednost je *false*.

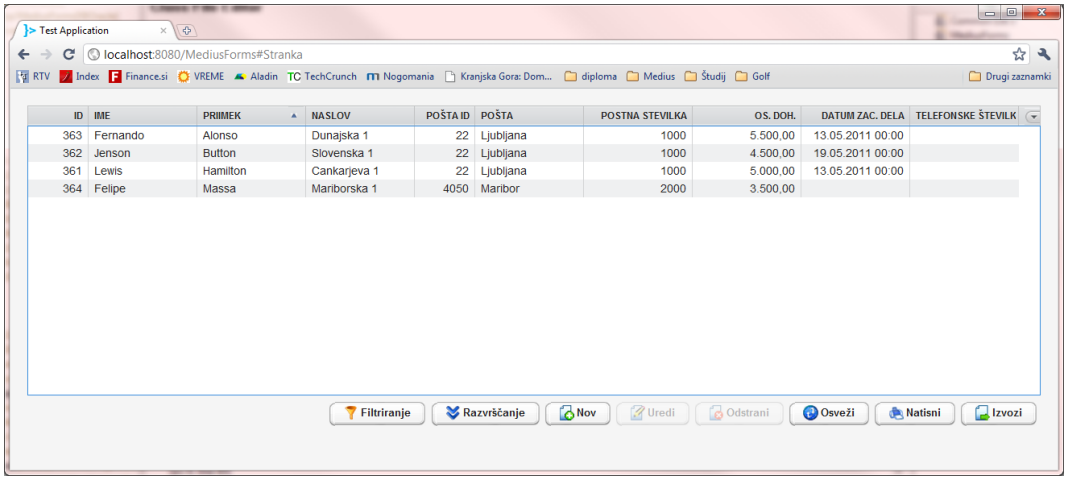
- *mapperClass* (Class)

  Določa kateri razred se uporabi za preslikavo med lastnostjo objekta in nizom, ki ga prikazujemo v stolpcu. Tipično ga uporabljamo za prikaz številk ali spremenljivk tipa *Date, Timestamp.*

- *mapperParams* (String[])

  Parametri, ki se uporabijo za konstruktor razreda *Mapper* (mapperClass). Uporablja naj se takrat, ko je določen *mapperClass*. Privzeto je seznam prazen.

Kot primer si poglejmo sliko 4.6, ki prikazuje tabelo šifrantov objekta *Stranka*.



Slika 4.6: Prikaz tabele s strankami

Primer kode za izpis osebnega dohodka. 4.4

Izvorna koda 4.4 Primer uporabe anotacije @ColumnPresentation za prikaz stolpca Osebni dohodek

```
@ColumnPresentation(order = 8, filterable = true,
width = 100, mapperParams="#,##0.00")
public BigDecimal getOsebniDohodek()
{return this.osebniDohodek;}
```

Parameter *order* ima vrednost 8, zato je osebni dohodek prikazan kot osmi v tabeli. Parameter *mapperParams* nam določi, kako naj se prikaže vrednost, *filterable* pa določi, da bomo po tej vrednosti lahko filtrirali seznam tabele.

V zgornji tabeli so prikazani tudi stolpci, ki so del entitete *Posta*. Če jih želimo prikazati, mora biti nastavljen *followOnLevel* na NORMAL, anotacije pa morajo biti prisotne tudi na atributih razreda *Posta* (v tem primeru gre za atribute idPosta, postaIme, postaStevilka). Primer vidimo v izvorni kodi 4.5.

@ExportPresentation

Anotacija, ki določi način predstavitve izvoza podatkov iz aplikacije. Uporablja se, ko želimo izvažati vsebino tabel v formate kot so PDF, XLS, itd. Z njo anotiramo razred.

Parametri:

Izvorna koda 4.5 Primer uporabe anotacije @ColumnPresentation za prikaz stolpcev pošte

```
@ColumnPresentation(order = 5, filterable = true,
folowOnLevel = LevelType.NORMAL)
public Posta getPosta(){return this.posta;}
```

- *orientation* (PageOrientation) Določimo orientacijo strani. Imamo tri možnosti:
  - VERTICAL Vertikalna ali portretna orientacija.
  - HORIZONTAL Horizontalna ali ležeča orientacija.
  - AUTOMATIC Uporabljena bo vertikalna orientacija, razen če bo število stolpcev večje kot je bilo določeno.
- *widths* (ColumnWidthSource) Določimo širino stolpcev v tabelah za izvoz. Imamo dve možnosti:
  - ANNOTATION Širino stolpcev se bo prebralo iz anotacij @ColumnPresentation.
  - CALCULATED Širina stolpcev bo izračunana na podlagi dejanske vsebine.

@TablePresentation

@TablePresentation je anotacija, ki določi način predstavitve tabele. Uporablja se v posebnih primerih, ko želimo določiti, katere lastnosti naj se uporabijo za prikaz in katere ne. Z njo anotiramo razred.

Parametri:

- *name* (String)

  Določi ime predstavitve tabele. Odjemalec pa določi na kakšen način bo izvedel predstavitev s tem imenom.

- *properties* (String [])

  Lastnosti, ki so uporabljajo pri tej predstavitvi tabele. Če niso določene, bo uporabljena predstavitev tabele na nivoju NORMAL.

- *excludedProperties* (String[])

  Lastnosti, ki naj bodo izključene iz predstavitve tabele.

Primer uporabe anotacije @TablePresentation nam predstavlja izvorna koda 4.6.

Izvorna koda 4.6 Primer uporabe anotacije @TablePresentation za izris tabele z želenimi atributi

```
@TablePresentation(name = "preprosta",
properties={"idStranka","ime", "priimek"})
public class Stranka
```

@TablePresentations

Anotacija @TablePresentations predstavlja seznam anotacij @TablePresentation. Za en objekt imamo lahko namreč več različnih tabel.

Parametri:

- *tables* (TablePresentation[])

4.2.3 Anotacije *Lookup*

Anotacije *Lookup* predstavljajo skupino anotacij, ki jih potrebujemo pri kreiranju izbirnih mask.

@LookupPresentation

Z anotacijo @LookupPresentation definiramo obnašanje izbirne maske za razred. Uporabljamo jo s kombinacijo anotacije @FieldPresentation in njeno vrednostjo *lookupClass*. V anotaciji @FieldPresentation povemo kateri razred naj se uporabi, v anotaciji @LookupPresentation pa kako naj se ta razred uporabi. Z njo torej anotiramo razred.

Parametri:

- *name* (String)

  Določa ime iskalnika. Odjemalec določi katera predstavitev se bo uporabljala s tem imenom.

- *managerClass*(Class)

  Upravljalski razred, ki ga podamo za dostop do podatkov.

- *tablePresentation*(String)

Določa katera predstavitev naj se uporablja za prikaz podatkov v tabeli (ki vsebuje dodane ali izbrane vrednosti). Ta možnost se uporabi, ko ne uporabimo anotacije @TablePresentation nad razredom ampak samo anotacije na metodah, kot je anotacija @ColumnPresentation. Imamo naslednje možnosti:

- *default_simple* (DefaultPresentation.DEFAULT_SIMPLE) Prikazujejo se vsa polja z nivojem SIMPLE.
- *default* (DefaultPresentation.DEFAULT_NORMAL) Prikazujejo se vsa polja z nivojem NORMAL ali SIMPLE.
- d*efault_detail* (DefaultPresentation.DEFAULT_DETAILED) Prikazujejo se vsa polja.

  Privzeta vrednost je *default_simple*.
- *format* (String) Določa kako naj bodo izbrani podatki predstavljeni v iskalnem polju.

Primer enega izmed izbirnih menijev *Lookup* prikazuje slika 4.7, kodo za kreacijo tega menija pa izvorna koda 4.7.

Slika 4.7: Izgled menija *Lookup* za izbor pošte

Izvorna koda 4.7 Primer uporabe anotacije @LookupPresentation za izris menija za izbiro pošte

```
@LookupPresentation(managerClass=PostaManagerRemote.class,
format = "#{postaStevilka}, #{postaIme}")
public class Posta
```

Poleg parametra *managerClass* je uporabljen še format, s katerim opišemo način prikaza objektov.

@LookupPresentations

Anotacija @LookupPresentations predstavlja seznam anotacij @LookupPresentation. Za en objekt imamo lahko namreč več različnih iskalnikov.

Parametri:

- *lookups* (LookupPresentation[])

@LookupMethod

Z anotacijo @LookupMethod opišemo metode razreda, ki se bo uporabljal za dostop do podatkov pri anotaciji @LookupPresentation.

Parametri:

- *lookupNames*(String[])

  Seznam imen @LookupPresentation (name), pri katerih se bo ta metoda uporabljala za dostop do podatkov.

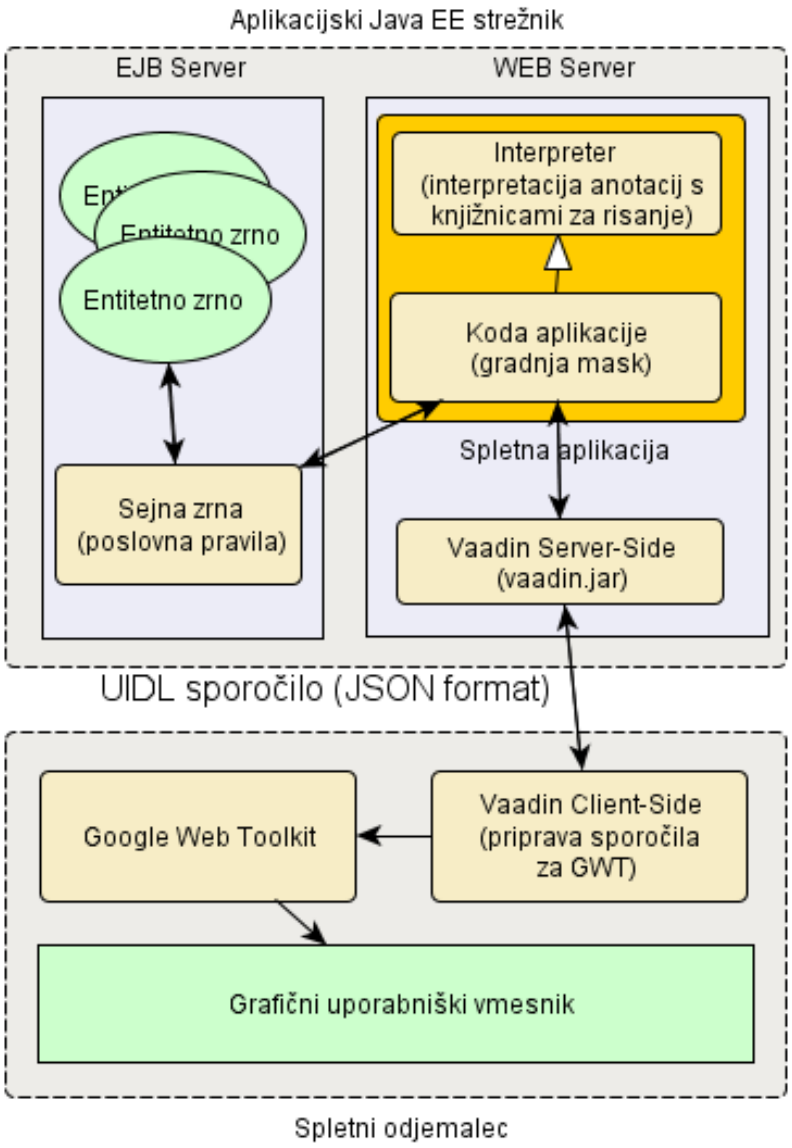
4.3 Interpretacija anotacij in komunikacija

V prejšnjem poglavju smo spoznali anotacije ogrodja za opisovanje entitetnih zrn, na podlagi katerih se na odjemalčevi strani zgradijo ustrezni grafični vmesniki. Z enostavnimi primeri javanske kode je bila prikazana tudi njihova uporaba ter slike mask, ki nastanejo na podlagi opisov. Da bi se maske dejansko izrisale na različnih platformah, pa je potrebno anotacije interpretirati na posamezno platformo. V nadaljevanju bomo spoznali, kdaj in kje se anotacije interpretirajo in kako posamezna aplikacija komunicira z aplikacijskim strežnikom. V delovanje interpreterja se ne bomo spuščali, saj to z vidika razvijalca poslovne aplikacije ni pomembno.

4.3.1 Spletne aplikacije

Spletne aplikacije se v osnovi razlikujejo od namiznih ali mobilnih aplikacij, pri katerih aplikacijo namestimo na napravo, kjer se bo aplikacija izvajala. Kot smo predhodno omenili, je za spletno Java EE aplikacijo potrebna tri nivojska arhitektura. Prvi nivo je podatkovna baza, drugi je aplikacijski strežnik in tretji spletni brskalnik oziroma odjemalec. V veliko primerih, še posebej kadar želimo večjo varnost aplikacije, lahko opazimo štiri nivojsko arhitekturo, kjer je aplikacijski strežnik ločen na dva dela: na strežniški in spletni del, ki tečeta na ločenih strežnikih.

Slika 4.8 prikazuje arhitekturo spletne Java EE aplikacije, ki uporablja ogrodje za avtomatsko kreiranje grafičnih vmesnikov.



Slika 4.8: Prikaz arhitekture spletne aplikacije in komunikacije

Kot lahko vidimo, je aplikacijski strežnik ločen na strežniški ali EJB (poslovni) del in spletni del. Strežniški del vsebuje entitetne razrede, kjer so z anotacijami opisana entitetna zrna, ter upravljalce EJB oziroma sejna zrna, v katerih so opisana poslovna pravila in skrbijo za prenos objektov iz strežniškega dela v spletni del aplikacije in obratno. Na spletnem delu strežnika teče naša

spletna aplikacija. Spletna aplikacija za risanje mask uporablja komponente razvojnega ogrodja Vaadin ter dodatno razvit interpreter za potrebe ogrodja, ki zna interpretirati z dodatnimi anotacijami opisana entitetna zrna. Opisano entitetno zrno vsebuje logiko oziroma navodila, kako narisati formo za določen objekt. Da se opisi entitet ustrezno izrišejo na grafičnem vmesniku, skrbi interpreter, ki iz anotacije in njenih parametrov razbere, za kakšno formo gre (navaden prikaz tabele, tip izbirnega menija, kakšna bo velikost vrstic, vnosna polja, števila ali znake, ali je polje obvezno, ali bo omogočeno razvrščanje, itd). To pomeni, da če želimo na spletnem odjemalcu dobiti masko za branje ali urejanje objekta in njegovih instanc, je potrebno v kodi aplikacije samo uporabiti ustrezen razred interpreterja, ki ponavadi kot parameter potrebuje opisani entitetni objekt (kateri objekt bomo risali in kako) in sejno zrno (upravljalec za pridobitev instanc objekta). Kako aplikacija uporablja razrede interpreterja bomo videli v praktičnem preizkusu v petem poglavju. Od tu naprej strežniški del ogrodja Vaadin poskrbi za ustrezno komunikacijo z odjemalcem in pretvori paket v sporočilo UIDL (ang. User Interface Definition Language), kjer ga sprejme odjemalski del ogrodja Vaadin in ga v formatu JSON (ang. JavaScript Object Notation) pripravi za risanje orodju GWT.

Spletne aplikacije torej za komunikacijo med aplikacijskim strežnikom in spletnim brskalnikom uporabljajo sporočila UIDL, ki so napisana v formatu JSON.

UIDL

UIDL je jezik za serializacijo vsebine uporabniških vmesnikov s spletnega strežnika na spletni brskalnik. Ideja je, da se strežniške komponente z UIDL narišejo na spletni brskalnik. To pomeni, da so sporočila UIDL poslana na spletni brskalnik, kjer so prevedena v GWT, ki nariše ustrezne komponente. Komunikacija UIDL je v ogrodju Vaadin realizirana v formatu JSON [1].

JSON

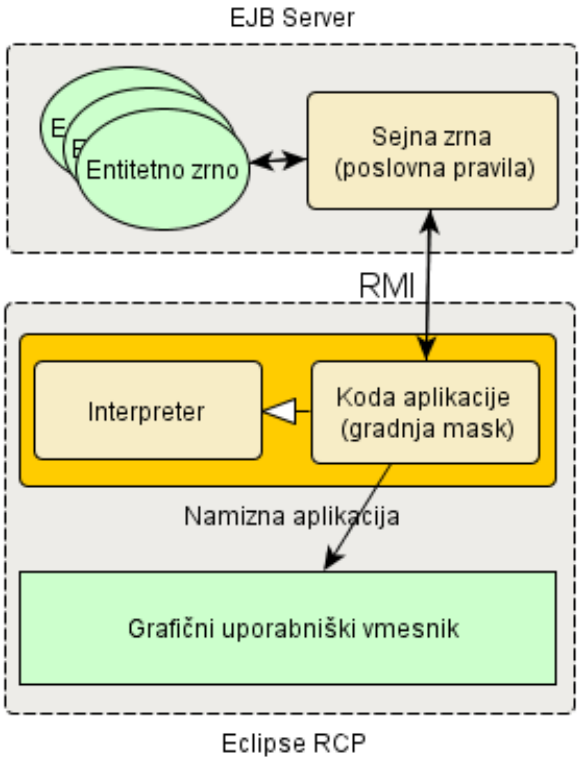
JSON se uporablja kot format za izmenjavo podatkov oziroma za implementacijo komunikacijskih protokolov. Zaradi preprostosti formata je razvijalcu enostaven za branje in pisanje, obenem pa ga računalnik lahko preprosto generira in razčlenjuje. Je tekstovni format, ki je povsem neodvisen od programskega jezika, nastal pa je na osnovi podmnožice programskega jezika JavaScript [20].

Zaradi hitrosti in preprostosti se ga uporablja namesto jezika XML. Za razliko od XML kjer so podatki zapisani samo kot niz znakov, lahko formatu JSON podamo tip podatkov (niz, število, tabelo, Boolovo vrednost). Prav tako je tudi pridobivanje podatkov iz kode JavaScript pri formatu JSON enostavneje kot pri jeziku XML.

Poenostavljeno lahko rečemo, da s sporočilom UIDL povemo katere lastnosti mora vsebovati sporočilo za pravilen in željen izris določene komponente (npr. gumba), JSON pa nam določa format, v katerem bomo sporočilo poslali spletnemu brskalniku.

4.3.2 Namizne aplikacije

Arhitektura namiznih Java EE aplikacij se malce razlikuje od arhitekture spletne aplikacije. Največja razlika je, da je namizna aplikacija nameščena na odjemalčevem računalniku. To pomeni, da sta koda aplikacije in interpreter ogrodja na odjemalcu.



Slika 4.9: Prikaz arhitekture namizne aplikacije in komunikacije

Slika 4.9 prikazuje primer namizne aplikacije, ki uporablja za risanje mask interpreter ogrodja za avtomatsko kreiranje grafičnih vmesnikov. Tako kot

pri spletnih aplikacijah, uporablja namizna aplikacija skupno strežniško kodo, nameščeno na EJB (strežniškem) delu aplikacijskega strežnika. Entitetna zrna so seveda opisana z našimi dodatnimi anotacijami za prikazovanje mask, ki jih bo na odjemalčevi strani interpretiral in s pomočjo orodja Eclipse RCP narisal interpreter. Komunikacija med strežnikom in aplikacijo poteka preko koncepta Java RMI (ang. Remote Method Invocation). Če še enkrat primerjamo arhitekturo spletne in namizne aplikacije, opazimo, da namizna aplikacija za gradnjo mask uporablja okolje, v katerem je aplikacija razvita (Eclipse RCP), medtem ko spletne aplikacije za risanje uporabljajo komponente (ogrodje Vaadin), ki so nameščene na aplikacijskem strežniku.

Java RMI

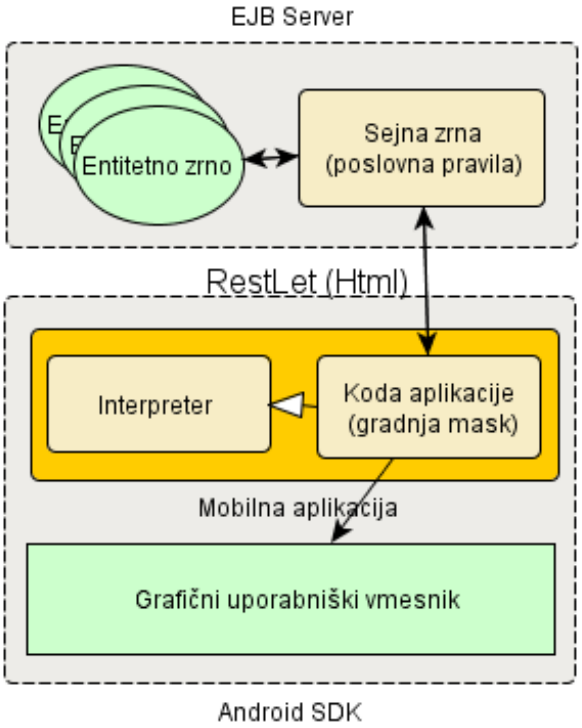
Kot smo omenili, je komunikacija med aplikacijskim strežnikom in namizno aplikacijo realizirana s konceptom Java RMI, ki omogoča razvijalcem, da kreirajo porazdeljene sisteme, ki kličejo oddaljene objektne metode in s tem delijo skupne vire in procesorski čas. Za razliko od drugih sistemov, ki omogočajo zgolj pošiljanje enostavnih tipov, RMI omogoča uporabo javanskih objektnih tipov, kjer se klici na objektne metode izvajajo med navideznimi javanskimi stroji. Kljub temu, da so lahko stroji locirani na različnih računalnikih lahko, en stroj kliče metode objekta, ki so shranjene v drugem [24].

4.3.3 Mobilne aplikacije

Arhitektura mobilnih Java EE aplikacij je zelo podobna arhitekturi namiznih aplikacij. Mobilna aplikacija, ki je nameščena na mobilni napravo z operacijskim sistemom Google Andorid, uporablja skupno strežniško kodo z anotiranimi entitetnimi zrni in pripadajočimi poslovnimi pravili. Interpreter ogrodja za avtomatsko kreiranje grafičnih vmesnikov interpretira anotacije v format, ki je razumljiv okolju Android SDK, ki nato nariše ustrezne maske na zaslon mobilne naprave. Komunikacija med aplikacijskim strežnikom in mobilno aplikacijo, nameščeno pod operacijskim sistemom Android, poteka s pomočjo ogrodja RestLet, ki za pošiljanje in sprejemanje podatkov uporablja format Html. Slika 4.10 prikazuje arhitekturo mobilne aplikacije.

RestLet

RestLet je celovito, a vseeno enostavno ogrodje Java REST, ki omogoča vključitev spletne arhitekture v aplikacijo ter izkorišča njeno prilagodljivost in enostavnost. Akronim REST (ang. Representational State Transfer) predstavlja tip spletne arhitekture za porazdeljene spletne sisteme. V bistvu gre za arhitekturo tipa odjemalec strežnik ter medsebojno komunikacijo, kjer gre za pošiljanje zahtev in sprejemanje odgovorov. Z uporabo ogrodja RestLet je



Slika 4.10: Prikaz arhitekture mobilne aplikacije in komunikacije

enostavneje združevati spletne storitve, spletne strani in odjemalce v enotno spletno aplikacijo. Zahvaljujoč konceptu dodajanja komponent, podpira ogrodje vse koncepte REST (Resource, Representation, Connector, Component, itd.) in je primerno tako za strežniški kot tudi klientni del aplikacije. Prav tako podpira vse spletne standarde, kot so Http, Xml, JSON, itd. [23]. Za komunikacijo med operacijskim sistemom Android in aplikacijskim strežnikom obstaja posebna izdaja ogrodja RestLet, ki smo jo uporabili v ogrodju za avtomatsko kreiranje grafičnih vmesnikov.

Poglavje 5

Praktična uporaba

Sedaj, ko smo spoznali, kaj je ogrodje za avtomatsko kreiranje grafičnih uporabniških vmesnikov in kako deluje, ter izbrali ustrezna razvojna orodja za posamezno izvajalno okolje, se bomo lotili njegove praktične uporabe pri razvoju poslovnih aplikacij. Ogrodje bomo preizkusili pri izdelavi spletnih aplikacij, namiznih aplikacij in mobilnih aplikacij.

S praktičnim preizkusom bomo skušali imitirati močno poenostavljeno in preprosto poslovno aplikacijo, ki vsebuje grafični vmesnik, s katerim lahko beremo, urejamo, vnašamo in brišemo poslovne podatke ter izvajamo ostale poslovno pomembne funkcije. V tretjem in četrtem poglavju smo govorili o osnovnih konceptih ogrodja in uporabi njegovih anotacij, sedaj pa bomo skušali te koncepte prenesti v praktično uporabo.

Najprej bomo definirali podatkovni model, iz katerega bomo kreirali podatkovno bazo. Iz podatkovne baze bomo kreirali entitetna zrna, ki jih bomo opisali z anotacijami JPA za preslikovanje iz in v podatkovno bazo. Pridobljena entitetna zrna bomo nato opisali še z dodatno izdelanimi anotacijami za gradnjo grafičnih vmesnikov, ki jih zna ogrodje interpretirati glede na ustrezno izvajalno okolje. Da bi se maske na grafičnem uporabniškem vmesniku dejansko narisale, pa je potrebno za vsako izvajalno okolje narediti aplikacijo, ki bo vsebovala kodo, v kateri definiramo, kje naj se maske zgradijo. Vse informacije o tem, kakšne maske in kateri podatki naj se prikažejo, bodo vsebovala entitetna zrna, risanje pa bo izvedel interpreter ogrodja, ki bo glede na izvajalno okolje uporabil ustrezne komponente.

V praktični uporabi si želimo torej s pomočjo razvitega ogrodja zgraditi tri aplikacije, ki se bodo izvajale v treh različnih okoljih, in bodo vse uporabljale isto podatkovno bazo, aplikacijski strežnik in strežniški del kode, na podlagi katere se bodo na odjemalčevi strani avtomatsko kreirale maske za up-

ravljanje s podatki. S praktičnim preizkusom želimo pokazati preprostost uporabe ogrodja pri razvoju aplikacij in s tem dokazati, da je gradnja grafičnih uporabniških vmesnikov z uporabo ogrodja hitrejša, učinkovitejša, enotnejša in enostavnejša.

5.1 Opis primera

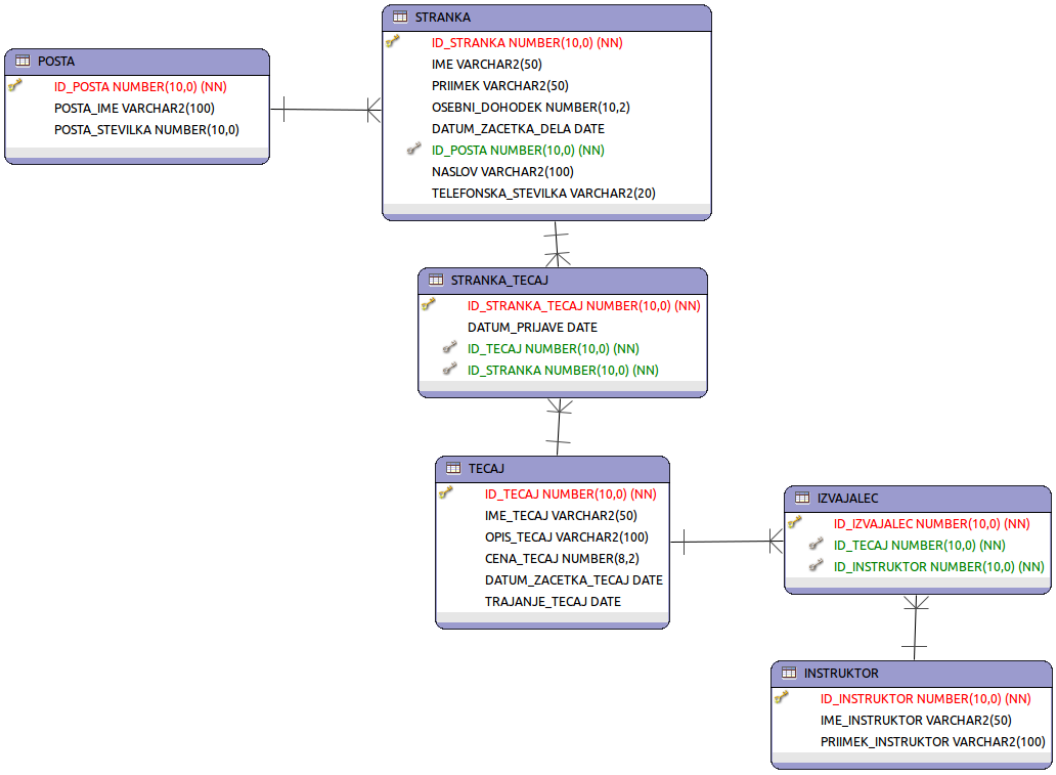
Za praktični preizkus smo izbrali preprost primer implementacije izobraževalnega sistema, ki pozna dva različna tipa oseb: stranke in inštruktorje. Stranke se prijavljajo na tečaje, inštruktorji pa tečaje izvajajo. Kljub preprostosti nam primer omogoča, da uporabimo večino anotacij ogrodja in tako prikažemo funkcionalnosti, ki jih danes vsebujejo poslovne aplikacije.

Podatkovna baza

Osnova vsake poslovne aplikacije za upravljanje s poslovnimi podatki je podatkovna baza, zato bo tudi naš preizkus ogrodja izhajal iz le-te. Podatkovno bazo bomo zgradili s pomočjo orodja Oracle Designer [22], ki bo iz načrtovanega podatkovnega modela za kreiranje podatkovne baze ustvarilo skripto SQL. Preprost entitetni model, ki ga prikazuje slika 5.1, smo implementirali v podatkovni bazi Oracle.

Na sliki vidimo, da entitetni model vsebuje šest tabel (štiri osnovne in dve vmesni tabeli) s pripadajočimi atributi:

- POSTA (Tabela POSTA vsebuje podatke o poštah.)
- STRANKA (V njej shranjujemo podatke o strankah. Je v ena proti mnogo razmerju s tabelo POSTA. To pomeni, da ima ena stranka lahko samo eno pošto, pošta pa ima lahko več različnih strank.)
- STRANKA_TECAJ (Predstavlja vmesno tabelo med entitetama STRANKA in TECAJ. Potrebna je zaradi razmerja mnogo proti mnogo. Omogoča prijavo strank na več tečajev, obenem pa ima lahko tečaj več strank.)
- TECAJ (V tabeli TECAJ shranjujemo podatke o tečajih.)
- IZVAJALEC (Je vmesna tabela med entitetama TECAJ in INSTRUKTOR, ki je potrebna zaradi razmerja mnogo proti mnogo. Tečaj ima lahko več različnih inštruktorjev, prav tako pa lahko inštruktor izvaja več različnih tečajev.)



Slika 5.1: Prikaz entitetnega modela za praktični preizkus ogrodja

- INSTRUKTOR (V tabeli INSTRUKTOR shranjujemo podatke o inštruktorjih.)

Grafični vmesniki

Vse tri aplikacije bodo vsebovale osnovne maske grafičnega vmesnika, ki bodo predstavljene kot maske s tabelami podatkov za vsako entiteto. Vsak stolpec bo predstavljal določen atribut entitete. Nad podatki v tabelah pa bo omogočeno izvajanje številnih akcij. Vsakič, ko bomo s pritiskom na gumb želeli izvesti neko akcijo, se bo odprla nova maska, ki bo vsebovala potrebne funkcije za uspešno izvedbo te akcije. To bodo bodisi polja za vnos podatkov, kompleksni izbirni meniji, ali pa samo potrditveni dialogi. Ker gre za poslovno aplikacijo, pri katerih je vsebina podatkov zelo pomembna, bo nujno potrebno poskrbeti tudi za ustrezno validacijo pri vnosu podatkov.

V naši aplikaciji bomo lahko upravljali z vsemi podatki naših entitet. Lahko bomo vnašali, spreminjali, brisali, izvažali, tiskali, razvrščali in filtrirali podatke. Ko bomo na primer urejali ali kreirali stranko, bomo lahko preprosto

preko izbirnega menija izbrali pošto in tečaje, na katere bomo stranko prijavili. Ko bomo urejali tečaje, jim bomo lahko dodajali nove stranke in inštruktorje. Kot primer v preizkusu, bomo kreirali tudi preprostega čarovnika, s katerim bomo lahko vnašali nove tečaje.

Aplikacijski strežnik

Aplikacijski strežnik je programsko ogrodje, ki nudi okolje v katerem se lahko izvajajo aplikacije, ne glede na to, kakšne so ali kaj delajo. Je nekakšen vmesnik, ki skrbi za povezavo s podatkovno bazo na eni strani in povezavo z odjemalcem na drugi. Tak koncept imenujemo tri-nivojska strežniška arhitektura, ki je predvsem značilna za Java EE aplikacije. Aplikacijski strežnik skrbi za številne storitve, zato se lahko razvijalci posvetijo implementaciji poslovne logike.

V praktičnem preizkusu ogrodja bomo uporabili aplikacijski strežnik JBoss [18], ki je najpogosteje uporabljen javanski aplikacijski strežnik na tržišču. Uporabljena bo različica strežnika JBoss 5.0, na katerega bomo namestili naš strežniški del kode. Koda bo predhodno opisana z anotacijami JPA za preslikovanje s podatkovno bazo, in pa dodatnimi anotacijami ogrodja, ki bo opise entitetnih zrna interpretiralo kot grafične vmesnike na odjemalcih. V našem preizkusu bo aplikacijski strežnik na eni strani komuniciral s podatkovno bazo, na drugi pa s kar tremi aplikacijami oziroma odjemalci; spletnim brskalnikom, namizno aplikacijo ter mobilno aplikacijo.

Strežniška javanska koda

Kot smo že večkrat povedali, je poleg skupne podatkovne baze, koda na strežniškem delu aplikacijskega strežnika edina skupna točka vsem trem aplikacijam. Zato je ravno strežniška koda ključnega pomena pri uporabi ogrodja za avtomatsko kreiranje grafičnih vmesnikov, saj jo uporabljajo tako spletne, namizne kot tudi mobilne aplikacije. Vsebuje entitetna zrna, to so javanski razredi, ki predstavljajo tabele v podatkovni bazi, in poslovna pravila oziroma upravljalce z entitetnimi zrni.

V našem primeru smo iz podatkovne baze naredili entitetna zrna, označena z anotacijami JPA. Pomagali smo si z ogrodjem Seam [25], ki zna iz podatkovne baze sam generirati entitetna zrna. S tem smo zagotovili komunikacijo med aplikacijskim strežnikom in podatkovno bazo, ki se izvaja s pomočjo metod entitetnega upravljalca. Metode za upravljanje, ki jih imenujemo tudi poslovna pravila, smo definirali v generičnem razredu *SifrantManagerBean.java.* Ta razred vsebuje tri osnovne metode: *getAll(), save(), delete()*, ki uporabljajo metode entitetnega upravljalca, ki smo jih omenili predhodno (*merge(), per-*

sist(), remove(), createQuery()). Uporabljali ga bomo, če bomo želeli izvajati osnovne operacije branja, shranjevanja in brisanja podatkov. Primer metode *save()* v razredu *SifrantManagerBean.java* nam prikazuje izvorna koda 5.1.

Izvorna koda 5.1 Primer metode *save()* razreda *SifrantManagerBean.java*

```
public abstract class SifrantManagerBean
<T extends SifrantEntity> implements SifrantManager<T> {
    protected EntityManager manager;
        public T save(T entity) {
        if (entity.getId() == null) {
            manager.persist(entity);
        } else {
            manager.merge(entity);
        } return entity;
    }
...
}
```

Za kompleksnejše operacije, ki se nanašajo predvsem na branje, shranjevanje in brisanje entitet, ki so med seboj povezane, pa bomo posameznemu entitetnemu razredu kreirali še njegovega upravljalca. Ker je entiteta *Stranka* povezana še z entitetama *Tecaj* in *Posta*, lahko kot primer kompleksnejšega upravljalca podatkov navedemo razred *StrankaManager.java*. Ta razred vsebuje metode, ki so opisane z anotacijo *@Override*, ter s tem povozijo metode, ki jih vsebuje razred *SifrantManagerBean.java*. Primer metode *save(),* v razredu *StrankaManager.java,* za shranjevanje stranke in njenih razmerij prikazuje izvorna koda 5.2. Iz kode je razvidno, da je v primeru shranjevanja entitete *Stranka* poskrbljeno tudi za shranjevanje entitet, ki so v razemerju s to entiteto. Za upravljanje z ostalimi entitetnimi zrni, vsebuje strežniški del aplikacije še razrede *InstruktorManager.java*, *PostaManager.java* in *TecajManager.java.*

Z ogrodjem Seam, smo pridobili entitetna zrna, ki znajo s pomočjo entitetnih upravljalcev komunicirati in se preslikovati v podatkovno bazo. Da pa bi lahko ogrodje narisalo želene maske, je potrebno sedaj entitetne razrede anotirati z anotacijami ogrodja. S tem jih bo lahko ogrodje interpretiralo in narisalo za posamezno izvajalno okolje. V našem primeru, kjer imamo stranke, ki se prijavljajo na tečaje, in inštruktorje, ki tečaje izvajajo, potrebujemo ustrezen grafični vmesnik za upravljanje z vsemi podatki. Če si torej želimo masko, ki bo prikazovala stranke, ki jih je moč dodajati, brisati in urejati, je potrebno z ano-

Izvorna koda 5.2 Primer metode *save()* razreda *StrankaManager.java*

```
...
@Override
public Stranka save(Stranka entity) {
 if (entity.getId() == null) {  // nov
  manager.persist(entity);
  for (StrankaTecaj st : entity.getStrankaTecajs()) {
   st.setDatumPrijave(new Date());
   st.setStranka(entity);
   manager.merge(st);
  }
 } else {
  manager.merge(entity)
  manager.createQuery(DELETE StrankaTecaj s
  WHERE s.stranka.idStranka = :id_stranka)
  .setParameter("id_stranka",entity.getId())
  .executeUpdate();

  for (StrankaTecaj st : entity.getStrankaTecajs()) {
   if (st.getId() == null)// novi stranka tečaj zapis {
    st.setDatumPrijave(new Date());
    st.setStranka(entity);
    manager.persist(st);
   } else {  // stari
    st.setStranka(entity);
    manager.merge(st);
   }
  }
}
...
```

tacijami ogrodja ustrezno opisati entiteni razred *Stranka.java.* Kako je opisan razred *Stranka.java* in njegove metode lahko vidimo, če si pogledamo primere uporabe anotacij, predstavljene v poglavju 4.2 v primerih izvorne kode 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 in 4.7. Opisan je z anotacijami @LookupPresentation, @ExportPresentation, @TablePresentation, njegove *get()* metode pa z anotacijami @ColumnPresentation in @FieldPresentation. Zaradi anotacije @ColumnPre-

sentation, ogrodje dobi navodila za risanje stolpcev v tabeli šifrantov in zgradi gumbe za akcije, kot so natisni, osveži, nov, uredi, odstrani, izvozi. Zaradi anotacije @FieldPresentation, pa ogrodje dobi navodila za risanje mask z vnosnimi polji, risanje izbirnih menijev in vključi validacijska pravila. Podobno, kot je opisan razred *Stranka.java*, so opisani vsi enitetni razredi naše aplikacije. Poleg razreda *Stranka.java*, so to še razredi *Posta.java*, *Instruktor.java*, *Tecaj.java* in razreda *StrankaTecaj.java* ter *Izvajalec.java*, ki predstavljata vmesni tabeli razmerij mnogo proti mnogo.

V našem preizkusu smo do sedaj kreirali podatkovno bazo in entitetne razrede, ki znajo podatke preslikati iz objektne oblike v relacijsko in nazaj, kar nam omogoča komunikacijo s podatkovno bazo. Prav tako smo skupno strežniško kodo opisali z anotacijami ogrodja. To skupno kodo je potrebno sedaj namestiti kot knjižnico na strežniški oziroma poslovni del aplikacijskega strežnika, ki ga bodo uporabljale vse aplikacije, ne glede na izvajalno okolje.

Kako uporabimo ogrodje za avtomatsko kreiranje grafičnih vmesnikov v določenem izvajalnem okolju, da se na podlagi opisa entitetnih zrn narišejo ustrezne maske na odjemalcu, pa si bomo pogledali v nadaljevanju.

5.2 Spletna aplikacija

Če želimo, da zna spletna aplikacija interpretirati anotacije, je ključnega pomena to, da vsebuje knjižnico ogrodja za avtomatsko kreiranje grafičnih vmesnikov, prirejeno spletnim aplikacijam. Ta knjižnica se imenuje *VaadinCommon.jar.* V njej so poleg osnovne knjižnice *Vaadin.jar* in še nekaterih drugih, javanski razredi ogrodja, ki jih v spletni aplikaciji pokličemo, če želimo da se narišejo maske. Ti javanski razredi interpretirajo anotacije in jih s pomočjo komponent ogrodja Vaadin narišejo na spletnem brskalniku. Izpostavili bomo najpomembnejša dva razreda, ki jih bomo uporabili tudi v našem preizkusu ogrodja. To sta razreda *TableView* in *GenericForm.*

Razred *TableView*

Razred *TableView* je del ogrodja za avtomatsko kreiranje grafičnih vmesnikov in se nahaja v knjižnici *VaadinCommon.jar.* Narejen je bil za potrebe risanja tabele šifrantov v spletni aplikaciji. Informacijo o tem, kako naj razred nariše tabelo šifrantov in pripadajoče gumbe dobi iz opisanih entitetnih zrn. Da bi razred prebral anotacije, pa mu moramo podati ustrezne parametre. Kot parametre mu podamo entitetni razred, iz katerega prebere anotacije, in

pripadajočega entitenega upravljalca, s katerim dobi vrednosti entitete. To pomeni, da se razred *TableView* v zanki sprehodi čez celoten entiteni razred in prebere njegove anotacije. Na podlagi anotacij, s komponentami ogrodja Vaadin, nariše masko na spletnem odjemalcu in jo napolni s podatki. Razred naredi tudi poslušalce dogodkov za gumbe. To pomeni, da zazna željeno akcijo in zato naredi klic na druge razrede ali metode, ki nato narišejo ustrezne komponente. Za lažje razumevanje si lahko predstavljamo vnos novega podatka. Ob pritisku na gumb ,,nov" v tabeli šifrantov, razred *TableView* naredi klic na drug razred ogrodja, ki zna narisati masko z vnosnimi polji. Razred, ki zna narisati vnosna polja se imenuje *GenericForm*.

Razred *GenericForm*

Tudi razred *GenericForm* je del ogrodja za avtomatsko kreiranje grafičnih vmesnikov. Kot smo povedali, ga uporabljamo za risanje maske z vnosnimi polji, za vnos novih ali urejanje starih podatkov. Vhodni parameter je entiteni razred, za katerega želimo narisati masko z vnosnimi polji. Razred *GenericForm* se v zanki sprehodi čez entitetni razred in prebere anotacije. Tako dobi informacijo kako, kje in kakšna polja je potrebno narisati. V primeru, da obstajajo še povezave na druge entitetne razrede, razred *GenericForm* nariše tudi ustrezne izbirne menije.

Sedaj, ko spletna aplikacija vsebuje vse potrebne knjižnice, je strežniška koda nameščena na aplikacijskem strežniku in poznamo razreda, s katerima povemu ogrodju, da nariše anotirana entietna zrna, lahko začnemo s pisanjem naše spletne aplikacije. V glavnem razredu, v našem primeru je to *TestApplication.java*, kjer se začne izvajanje aplikacije, bomo dodali pogled za vsak entiteni razred, ki ga želimo prikazati na grafičnem vmesniku. Če želimo na aplikacijo dodati pogled o strankah, to naredimo s klicem metode *addView(,,Stranka", new StrankaView())*, kot je prikazano v izvorni kodi 5.3.

V klasični spletni aplikaciji bi morali sedaj ta pogled ročno kreirati. To pomeni, da bi v razredu *StrankaView.java* ročno sprogramirali tabelo, urejevalne maske, labele, gumbe, itd. Tukaj nastopi ogrodje za avtomatsko kreiranje grafičnih vmesnikov. Sedaj namesto ročnega programiranja naredimo klic na razred ogrodja, ki interpretira opisan entitetni razred *Stranka.java.* Za kreiranje pogleda *StrankaView.java,* uporabimo razred *TableView*, ki nariše tabelo šifrantov.

Izvorna koda 5.4 nam prikazuje primer uporabe razreda *TableView*. Če želimo, da ogrodje nariše tabelo šifrantov za stranke, mu moramo podati strežniški del kode, in sicer entiteni razred *Stranka*, kjer prebere anotacije, in

Izvorna koda 5.3 Primer dodajanja pogleda v spletno aplikacijo

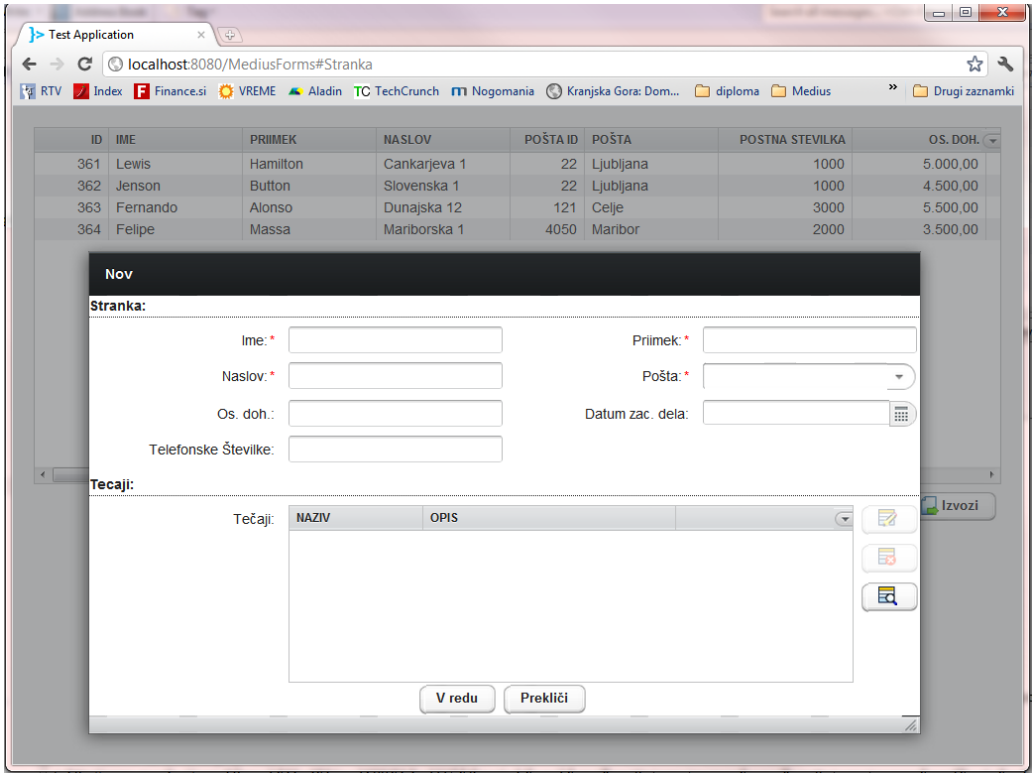
```
public class TestApplication extends CommonApplication {
 private TPTMultiView controller;
 public void init() {
  super.init();
  Window mainWindow = new Window("Test Application");
  setMainWindow(mainWindow);
  controller = new TPTMultiView(true);
  controller.addView("Stranka", new StrankaView());
  // ... dodamo ostale poglede
  mainWindow.addComponent(controller);
 }
}
```

Izvorna koda 5.4 Primer kreiranja pogleda v spletni aplikaciji z uporabo razreda *TableView*

```
public class StrankaView extends BaseView {
 public StrankaView(){
  TableView<Stranka> table = new TableView<Stranka>
  (Stranka.class, StrankaManagerLocal.class);
  addComponent(table);
 }
}
```

upravljalca entitete, *StrankaManager*, kjer dobi informacijo o tem, kako pridobiti objekt oziroma instanco objekta. S tem preprostim klicem se na spletnem odjemalcu nariše začetna maska s tabelo šifrantov in pripadajočimi gumbi. Slika 5.2 prikazuje primer maske, ki se zgradi na podlagi izvorne kode 5.4. Kot vidimo, se zaradi klica razreda *TableView* najprej nariše tabela šifrantov, ob pritisku na gumb „nov“, pa razred *TableView* naredi klic še na razred *GenericForm*, ki nariše vnosna polja.

V opisu primera smo si zadali tudi nalogo kreiranja preprostega čarovnika. Kreirali bomo maske za vnos tečaja preko čarovnika. Zopet je realizacija zelo preprosta. Namesto tabele šifrantov, si želimo, da bo začetna maska vsebovala vnosna polja za vnos podatkov o tečajih. Zato za izris ne uporabimo razreda



Slika 5.2: Prikaz avtomatsko generiranih mask v spletni aplikaciji

TableView, temveč razred *GenericForm*. Primer metode za izris maske za vnos podatkov prikazuje izvorna koda 5.5.

Izvorna koda 5.5 Primer kreiranja maske za vnos podatkov v spletni aplikaciji z uporabo razreda *GenericForm*

```
public class TecajPage extends FormWizardPage {
...
 @Override
 protected Form createForm() {
         GenericForm<Tecaj> form =
         new GenericForm<Tecaj>(Tecaj.class);
     return form;
 }
...
}
```

Kakšen je izgled spletne aplikacije na podlagi interpretacije anotacij lahko vidimo na sliki 5.2 ali slikah v poglavju 4.2. Slika 4.6 prikazuje tabelo šifrantov. Obenem vidimo, da so se narisali tudi gumbi za dodajanje, urejanje, osveževanje, razvrščanje, filtriranje, brisanje in tiskanje. Polja, ki vsebujejo informacijo o osebnem dohodku, so formatirana na izpis, ki je bil definiran s parametri anotacije. Slika 4.5 prikazuje masko za urejanje oziroma vnos stranke. Poleg osnovnih tekstovnih polj sta se narisala tudi menija za izbor pošte in tečaja. Prilagojenega formata je tudi polje za vnos datuma.

5.3 Namizna aplikacija

Sedaj si poglejmo še primer uporabe ogrodja pri izdelavi namizne aplikacije. Če želimo uporabljati ogrodje za avtomatsko kreiranje grafičnih vmesnikov, moramo zopet uporabiti knjižnice ogrodja, ki vsebujejo razrede, ki interpretirajo in rišejo maske na odjemalcu. Ker gre za namizno aplikacijo, je temu primerno prilagojena tudi knjižnica.

Izris tabele šifrantov in mask za dodajanje in urejanje podatkov se ne razlikuje bistveno od spletne aplikacije. Namesto risanja vseh komponent, ogrodje zopet prebere anotacije strežniške kode in avtomatsko kreira maske. Primer uporabe razreda pri kreaciji pogleda v namizni aplikaciji, nam prikazuje izvorna koda 5.6.

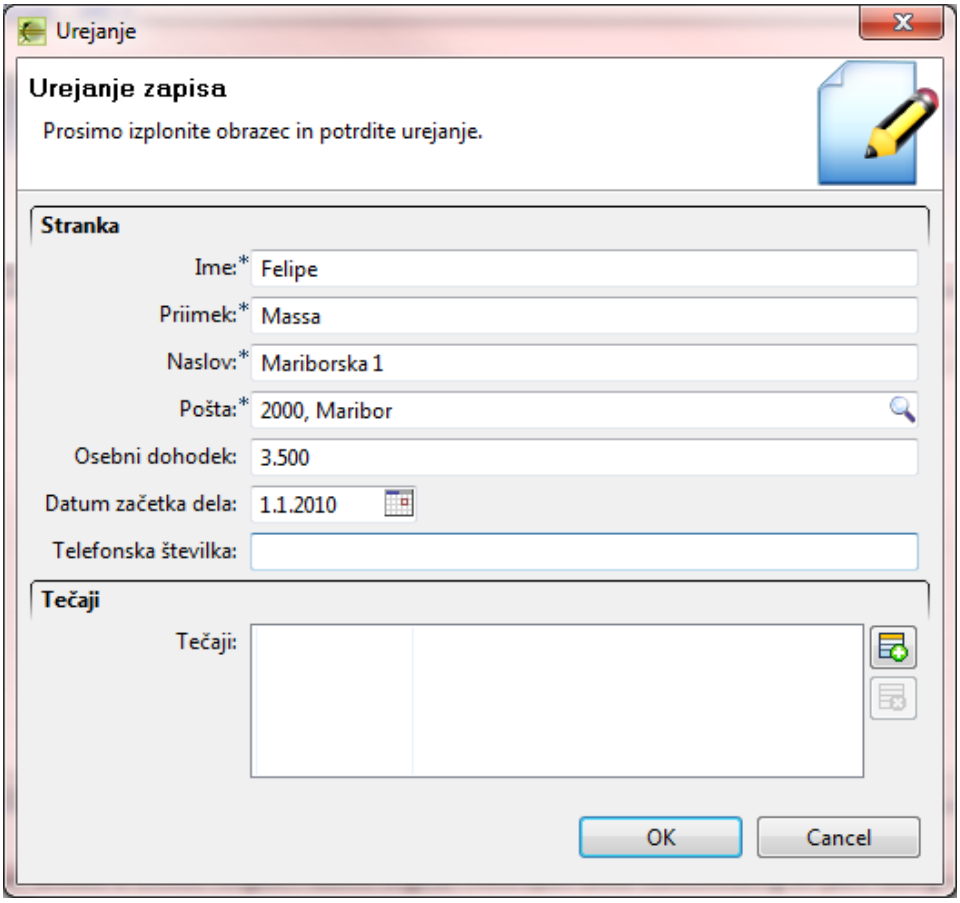
Izvorna koda 5.6 Primer kreiranja pogleda v namizni aplikaciji z uporabo razreda *ATableView*

```
public class StrankaView extends ATableView<Stranka> {
    public StrankaView() {
     super(Stranka.class, StrankaManagerRemote.class);
    }
...
}
```

Za interpretacijo in risanje zopet kličemo razred ogrodja. V primeru namiznih aplikacij pogled razširja razred *ATableView*, ki kot vhodni parameter zopet zahteva entiteto in njenega upravljalca. Razred *ATableView*, ki ga uporabljamo pri risanju namiznih aplikacij, se obnaša podobno kot razred *TableView* v spletnih aplikacijah. Podobno je tudi pri kreiranju čarovnika. Tudi tukaj za kreiranje vnosnega polja uporabimo razred ogrodja. Gre za razred *EditorBuilder*, ki deluje podobno kot razred *GenericForm* pri spletnih

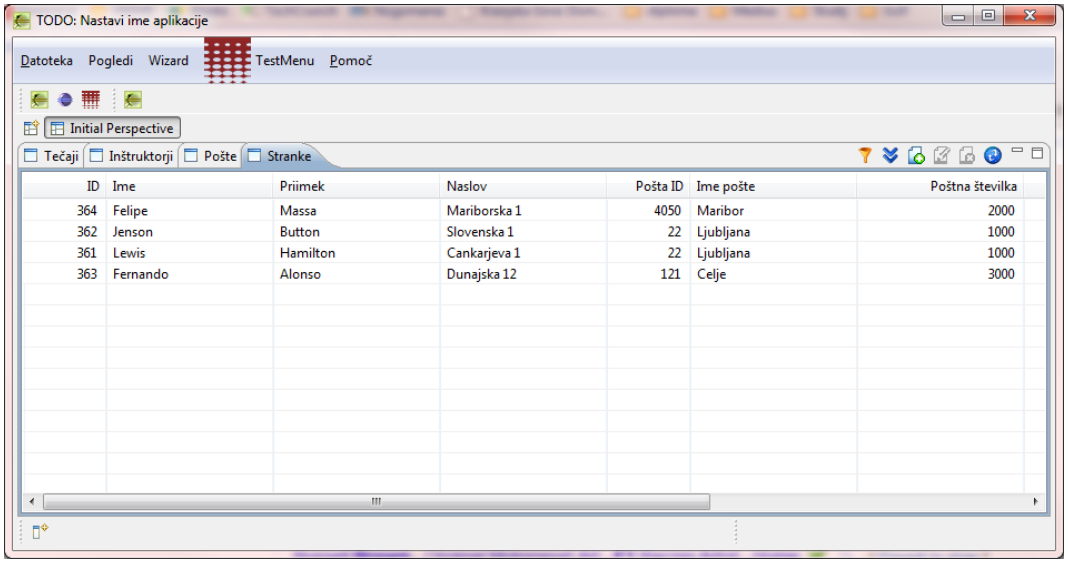
aplikacijah. Na podlagi opisa entitetnega razreda *Tecaj.java* bo narisal polja za vnos imena tečaja, opisa, cene, začetka tečaja, trajanja tečaja in izbirne menije za izbor strank in inštruktorjev.

Izgled namizne aplikacije, nam prikazujejo slike 5.3, 5.4 in 5.5.

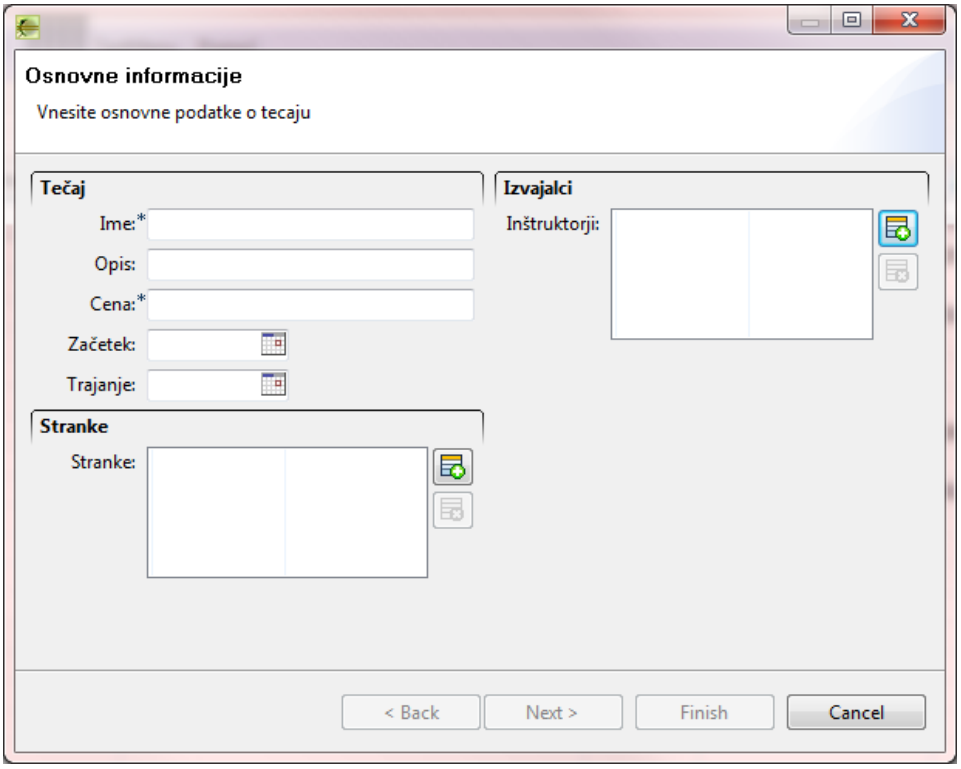


Slika 5.3: Prikaz maske za urejanje ali dodajanje entitete v namizni aplikaciji

Na sliki 5.3 lahko vidimo masko za vnos ali urejanje stranke. Tako kot v spletni aplikacij imamo osnovna vnosna polja in menije za izbor datuma, pošte in tečajev. Slika 5.4 prikazuje tabelo šifrantov in pripadajoče gumbe. Na sliki 5.5 je prikazan preprost čarovnik za vnos novega tečaja. Vidimo lahko, da so narisana vnosna polja za osnovne podatke in meniji za izbiro udeležencev tečajev in inštruktorjev, ki te tečaje izvajajo.



Slika 5.4: Prikaz tabele šifrantov v namizni aplikaciji



Slika 5.5: Prikaz čarovnika za vnos tečaja v namizni aplikaciji

5.4 Mobilna aplikacija

Izdelava mobilne aplikacije z ogrodjem za avtomatsko kreiranje grafičnih vmesnikov je zopet zelo enostavna. Mobilna aplikacija uporablja knjižnico z razredi za interpretiranje in risanje mask. Ta knjižnica je prilagojena izvajanju aplikacij v izvajalnem okolju Android. Poleg te knjižnice mora mobilna aplikacija uporabiti tudi knjižnico s strežniško kodo, ki je nameščena na aplikacijskem strežniku.

Ko v razvojnem okolju poskrbimo za našteta pogoja, je potrebno le kreirati pogled za posamezno entiteto. Logika kreiranja je povsem podobna spletnim in namiznim aplikacijam. Če želimo prikazati seznam strank, kreiramo pogled *StrankaView.java*, ki razširja *TableView*. *TableView* je razred ogrodja, ki nariše seznam instanc entitete na zaslon mobilne naprave. Uporablja komponente okolja Andorid SDK. Poleg metod za risanje seznama, kliče tudi metode za risanje vnosnih polj, gumbov, izbirnih menijev, komponent za vnos datuma, itd. Kot vhodne parametre potrebuje entiteto in entitetnega upravljalca. Kreiranje pogleda *StrankaView.java* prikazuje izvorna koda 5.7.

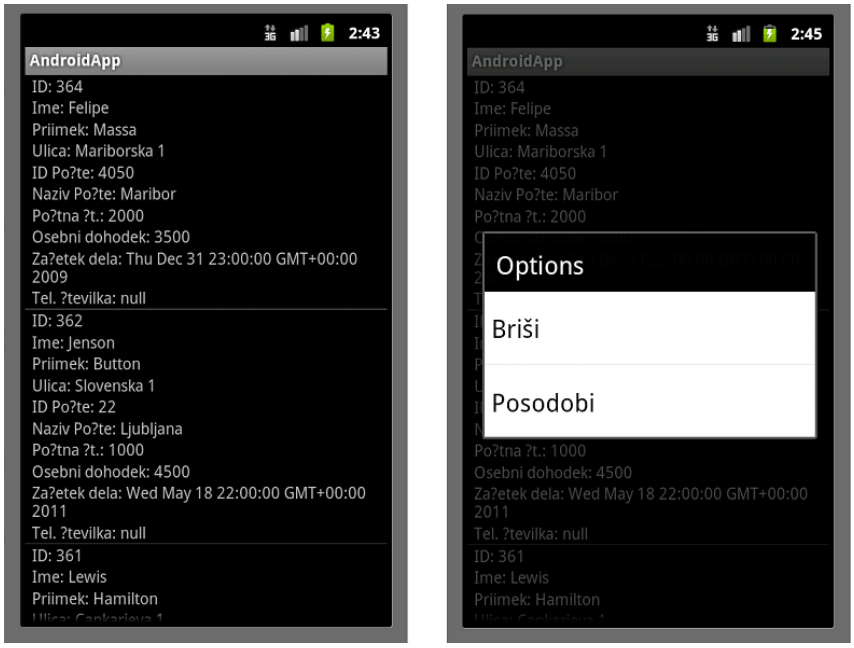
Izvorna koda 5.7 Primer kreiranja pogleda v mobilni aplikaciji z uporabo razreda *TableView*

```
public class StrankaView extends TableView<Stranka> {
    @Override
    protected Class<Stranka> initEntityClass() {
        return Stranka.class;
    }
    @Override
    protected Class<? extends SifrantManager<Stranka>>
                                    initManagerClass() {
        return StrankaManagerLocal.class;
    }
}
```

Pri razvoju aplikacij v okolju Android, je potrebno upoštevati še dodatno pravilo, ki določa, da moramo v datoteki *AndroidManifest.xml* vedno definirati vse aktivnosti, ki jih izvajamo v mobilni aplikaciji. *AndoridManifest.xml* je datoteka, ki jo mora vsebovati vsaka aplikacija Android. Vsebuje ključne informacije za uspešno izvajanje aplikacije. V njej poimenujemo javanski paket aplikacije, definiramo aktivnosti aplikacije, kakšna dovoljenja so potrebna za izvajanje aplikacije, itd. [13]. V našem primeru moramo definirati aktivnost

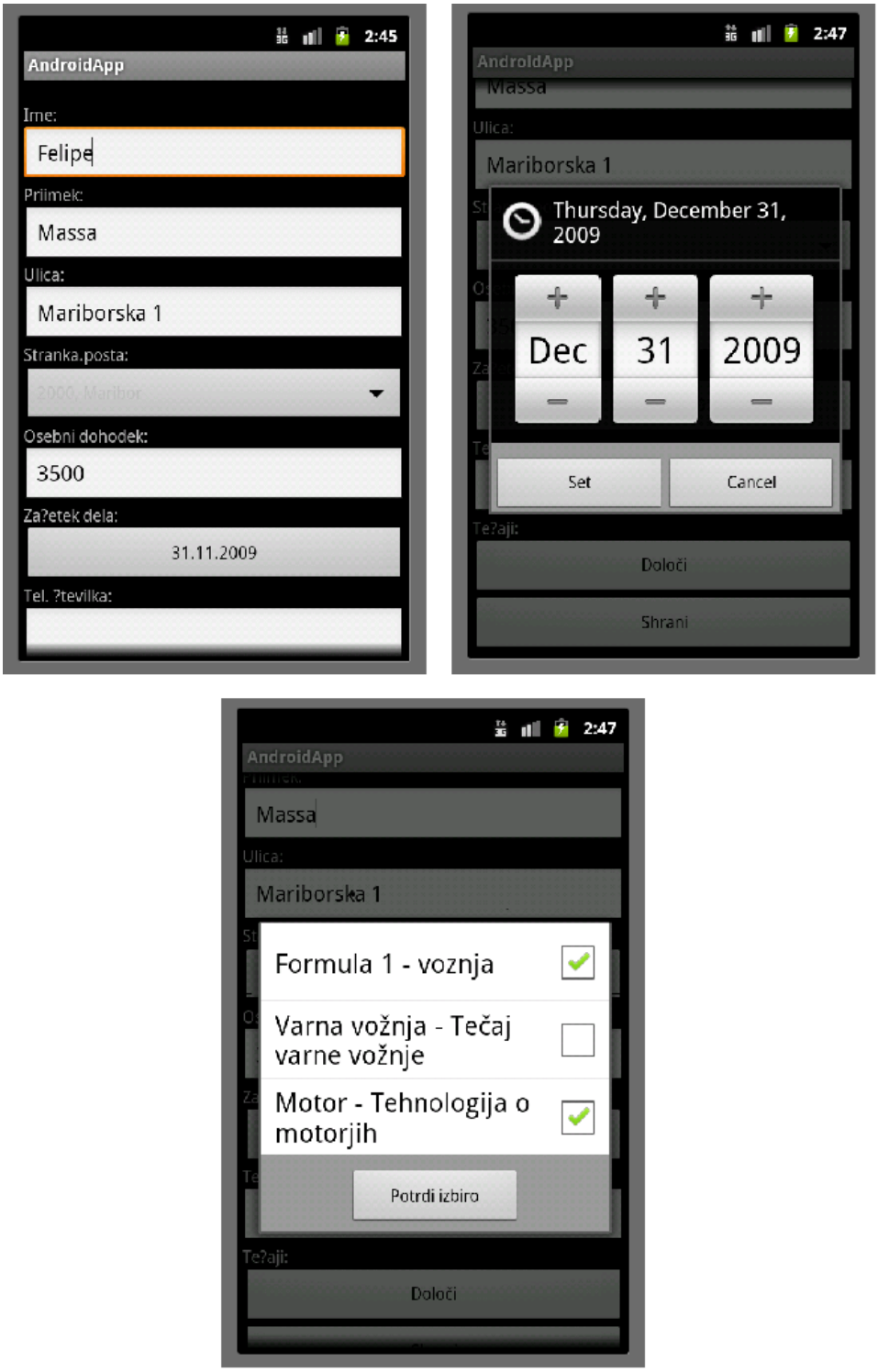
TableView, ki nariše seznam entitet in aktivnost *EditorView*, ki jo razred *TableView* uporabi za izris vnosnih polj za urejanje ali dodajanje podatkov. Razred *EditorView* je po zgradbi podoben razredoma *GenericForm* in *EditorBuilder* pri spletnih in namiznih aplikacijah. Vpisovanje aktivnosti v datoteko *AndoridManifest.xml*, bomo v bližnji prihodnosti avtomatizirali, saj želimo programirati aplikacije izključno v programskem jeziku Java.

Kakšen je izgled mobilne aplikacije, ki je nastal na podlagi anotiranih strežniških java komponent, nam bosta prikazali sliki 5.6 in 5.7.



Slika 5.6: Prikaz seznama strank in gumbov v aplikaciji Android

Na prvi sliki 5.6 lahko vidimo seznam strank in gumbe, ki se izrišejo, ko s pritiskom na zaslon izberemo neko stranko. Druga slika 5.7 pa nam prikazuje masko za urejanje stranke, ki vsebuje vnosna tekstovna polja in različne tipe izbirnih menijev.



Slika 5.7: Prikaz maske in njenih komponent za urejanje strank v aplikaciji Android

Poglavje 6

Zaključek

V uvodu diplomske naloge smo si za cilj zadali pokazati prednosti razvoja poslovnih aplikacij z uporabo ogrodja za avtomatsko kreiranje grafičnih vmesnikov. Smatrali smo, da so prednosti razvoja z ogrodjem predvsem v enotnejšem razvoju aplikacij za različna izvajalna okolja, bogatem naboru avtomatsko kreiranih grafičnih komponent, hitrejšem razvoju aplikacij, enotni strežniški kodi, pripadajočih poslovnih pravilih in validacijski logiki, neodvisnosti obnašanja in oblike grafičnih komponent ne glede na število razvijalcev, enostavnem spletnem oblikovanju in razvoju izključno v programskem jeziku Java.

Da bi dokazali omenjene prednosti ogrodja pri razvoju poslovnih aplikacij, je bilo potrebno ogrodje preizkusiti na praktičnem primeru. Preden pa smo začeli s praktičnim preizkusom, je bilo potrebno spoznati, kaj ogrodje je ter kje in kako deluje. Najprej smo predstavili izvajalna okolja, v katerih se danes izvajajo poslovne aplikacije. Znotraj posameznega izvajalnega okolja smo izbrali okolje, v katerem bomo poslovno aplikacijo razvili. V posameznem razvojnem okolju smo izbirali med odprtokodnimi ogrodji, ki pomagajo pri razvoju aplikacij. Tako smo izbrali ogrodja, na podlagi katerih bo ogrodje za avtomatsko kreiranje grafičnih vmesnikov risalo komponente na odjemalcu. Za spletne aplikacije smo izbrali ogrodje Vaadin, za namizne aplikacije Eclipse RCP in za mobilne aplikacije Android SDK. Nato je bilo nujno potrebno, da se spoznamo s temelji ogrodja, okoljem Java EE in pripadajočimi funkcionalnostmi, kot so označbe, objektno relacijske preslikave, EJB, itd. Poznavanje temeljev nam je omogočilo, da se spoznamo z delovanjem ogrodja in njegovo uporabo. Spoznali smo anotacije ogrodja, s katerimi opisujemo entitete, na podlagi katerih zna ogrodje izrisati grafične vmesnike za posamezno izvajalno okolje. Preden smo se lotili praktične uporabe ogrodja, smo spoznali še, kdaj in kje se anoti-

rane entitete interpretirajo in kakšna je komunikacija aplikacij z aplikacijskim strežnikom. V praktičnem preizkusu smo naredili tri aplikacije za posamezno izvajalno okolje. V vsaki smo enostavno uporabili ogrodje in predhodno anotirane entitete s pripadajočimi poslovnimi pravili, iz katerih so se izrisale maske za upravljanje s podatki. S tem smo dokazali, da je razvoj enoten na različnih izvajalnih okoljih, kjer vse aplikacije uporabljajo skupno strežniško kodo in pripadajoča poslovna pravila. Čas, ki smo ga porabili za anotiranje entitet, je v primerjavi s časom, ki bi ga porabili za izdelavo vseh mask na vseh izvajalnih okoljih, bistveno krajši. Ker ogrodje za risanje uporablja komponente že uveljavljenih ogrodij, kot so Vaadin, Eclipse RCP in Android, lahko rečemo, da omogoča bogat nabor grafičnih komponent. Razvijalec določi samo kje, ne pa tudi kako se komponenta izriše, zato je izgled mask aplikacije enoten kljub velikemu številu razvijalcev, ki istočasno delajo na aplikaciji. V praktični uporabi smo dokazali tudi, da je za razvoj aplikacij potrebno samo programiranje v programskem jeziku Java.

Kljub številnim prednostim, ki jih k razvoju prinaša ogrodje za avtomatsko kreiranje grafičnih uporabniških vmesnikov, obstaja še veliko možnosti za napredek in nadaljnji razvoj. Ogrodje ni produkt, ki bi ga lahko uporabljali vsi tega željni razvijalci poslovnih aplikacij, poleg tega pa njegova uporaba zahteva odlično poznavanje javanskih tehnologij, predvsem dela EJB, in privajanje na razvoj z uporabo anotacij ogrodja. Čas učenja je zato mogoče daljši kot pri razvoju v drugih ogrodjih. Kot pomislek za njegovo izboljševanje in nadgradnjo obstaja tudi možnost, da bi ponudili ogrodje skupnosti odprtokodnih razvijalcev, ki bi zagotovo pripomogli pri ustvarjanju boljšega, bolj prepoznavnega produkta v prihodnosti.

Slike

Seznam izvorne kode

Literatura

[1] M.Gronross, Book of Vaadin: Vaadin 6.4, Vadin Ltd, Turku, Finland, 2010, pogl. 3, Dodatek A

[2] N. Li, H.Li, J.Wu, AUTOSAR Based Automatic GUI Generation, ISORC 2010, 13th IEEE International Symposium, str. 156-162. Dostopno na: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5479085

[3] T.Lindholm, F.Yellin, The Java Virtual Machine Specification Second Edition, Addison-Wesley, Palo Alto, California, 1999, pogl. 1

[4] M.Monteiro, P.Oliveira, R.Goncalves, A source code based model to generate GUI, ICEIS 2008, Volume DISI, str. 449-452. Dostopno na: http://iconline.ipleiria.pt/bitstream/10400.8/148/1/ICSOFT2008_Marco-Monteiro.pdf

[5] E.Roman, G.Brose, R.P.Sriganesh, Mastering Enterprise JavaBeans, 3rd Edition, Wiley Publishing, Inc., Indianapolis, 2005, pogl. 6

[6] R.Rouvoy, P.Merle, Leveraging component programming with attribute-oriented programing, ECOOP 2006, Nantes, France

[7] A.L.Rubinger, B.Burke, Enterprise JavaBeans 3.1, Sixth Edition, O'Reilly Media, Inc., California, 2010, pogl. 1,2,9

[8] R.P.Sriganesh, G.Brose, M.Silverman, Mastering Enterprise JavaBeans 3.0, Wiley Publishing, Inc., Canada, 2006, dodatek B

[9] Sun Microsystems, The Java EE 6Tutorial, Volume I, Specification: JSR-000220, Palo Alto, California, 2006, pogl.1

[10] B.Štok, Application Framework For Management Nodes Software, Record No. in SOPRAN 238447, Iskratel, Ljubljana, Slovenia, 2004

Spletni viri:

[11] (2011) Android applications. Dostopno na:
http://www.androidapps.com/

[12] (2011) Android developers. Dostopno na:
http://developer.android.com

[13] (2011) The AndroidManifest.xml File. Dostopno na:
http://developer.android.com/guide/topics/manifest/manifest-intro.html

[14] (2011) Ecplise RCP. Dostopno na:
http://wiki.eclipse.org/index.php/Rich_Client_Platform

[15] (2010) EJB 3 Specification. Dostopno na:
http://java.sun.com/products/ejb/docs.html

[16] (2011) Google Web Toolkit, Dostopno na:
http://code.google.com/intl/sl-SI/webtoolkit/overview.html

[17] (2010) Hibernate Anotations. Dostopno na:
http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/

[18] (2011) Aplikacijski strežnik JBoss. Dostopno na:
http://www.jboss.org/jbossas/

[19] (2004) JDK 5.0 Documentation. Dostopno na:
http://download.oracle.com/javase/1,5.0/docs/index.html

[20] (2011) Introducing JSON. Dostopno na:
http://www.json.org/

[21] (2009) Oracle Forms 11g. Dostopno na:
http://www.oracle.com/technetwork/developer-tools/forms/overview/index.html

[22] (2010) Oracle Designer 10g Release 2. Dostopno na:
http://www.oracle.com/technetwork/developer-tools/designer/overview/index.html

[23] (2011) Restlet Framework. Dostopno na:
http://wiki.restlet.org/docs_2.1/13-restlet/21-restlet.html

[24] (2011) Java Remote Method Invocation, Oracle. Dostopno na:
http://download.oracle.com/javase/6/docs/technotes/guides/rmi/index.html

[25] (2009) Seam Framework. Dostopno na:
http://seamframework.org/