UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Mitja Krebelj

Zasnova in izvedba ogrodja za razvoj spletnih aplikacij

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentorica: doc. dr. Mojca Ciglarič

Ljubljana, 2010

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: 01698/2010
Datum: 01.09.2010

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

Kandidat: **MITJA KREBELJ**

Naslov: **ZASNOVA IN IZVEDBA OGRODJA ZA RAZVOJ SPLETNIH APLIKACIJ**

DESIGN AND IMPLEMENTATION OF THE FRAMEWORK FOR WEB APPLICATIONS DEVELOPMENT

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Preučite nekaj najpogostejših ogrodij za razvoj spletnih aplikacij, analizirajte njihove prednosti in pomankljivosti ter utemeljite, zakaj je v vašem primeru izdelava lastnega ogrodja boljša od uporabe že izdelanih ogrodij. Predstavite arhitekturo ogrodja, ocenite zmogljivosti ter izpostavite prednosti in slabosti vašega ogrodja na konkretnih praktičnih primerih. Na podlagi izkušenj podajte predloge za nadaljnji razvoj in izboljšave ogrodja.

Mentor:
M. Ciglarič
doc. dr. Mojca Ciglarič

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani **Mitja Krebelj**,

z vpisno številko **24930448**,

sem avtor diplomskega dela z naslovom:

Zasnova in izvedba ogrodja za razvoj spletnih aplikacij

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom **doc. dr. Mojce Ciglarič**,
- 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 10.11.2010 Podpis avtorja:

Zahvala

Zahvaljujem se mentorici, doc. dr. Mojci Ciglarič, za pomoč pri izdelavi diplomske naloge in dr. Matjažu Pančurju za dodaten vsebinski pregled naloge. Zahvaljujem se podjetju 3 PORT, d. o. o. in nadrejenim za zaupanje pri izvedbi projekta, brez katerega ne bi bilo te diplomske naloge. Hvala tudi vsem sodelavcem, ki so me spodbujali med študijem. Hvala staršema, ki sta mi omogočila študij in v vseh letih nista obupala nad mano. Hvala tudi tebi, Andreja, ker si skrbela zame med zaključevanjem študija.

Kazalo

Povzetek **1**

Abstract **2**

1 Uvod **3**

2 Ogrodje **6**
2.1 Skupne in vroče točke ogrodja 8
2.1.1 Omogočanje vročih točk z dedovanjem 8
2.1.2 Omogočanje vročih točk s kompozicijo 10
2.2 Klasifikacija ogrodij 11
2.2.1 Ogrodja po načelu bele škatle 12
2.2.2 Ogrodja po načelu črne škatle 13
2.2.3 Ogrodja po načelu sive škatle 14
2.3 Prednosti in slabosti ogrodij 15

3 Spletna ogrodja **16**
3.1 Klasifikacija spletnih ogrodij 19
3.2 Primera spletnih ogrodij 23
3.2.1 Apache Wicket 23
3.2.2 GWT 25
3.3 Odločitev za lastno ogrodje 27
3.3.1 Splošne zahteve 28

4 Razvoj lastnega spletnega ogrodja **31**
4.1 Izbira razvojnega okolja 31
4.2 Glavni moduli 32
4.3 Jedro ogrodja 3iOS 32
4.3.1 Arhitektura kode jedra 34
4.3.2 Priprava in dostop do jedra 36

4.3.3 Različne implementacije jedra 37
4.3.4 Vmesnik modula 3iFS in organizacija map 38
4.3.5 Avtentikacija in avtorizacija 41
4.3.6 Sistemske aplikacije 43
4.3.7 Potek razvoja 45
4.4 Predložni stroj 46
4.4.1 Objektno usmerjene predloge 47
4.4.2 Tipi značk 49
4.4.3 Izvajanje aplikacij 50
4.4.4 Potek razvoja 50
4.5 Aplikacije in uporabniški vmesniki 3iGUI 51
4.5.1 Glavne značilnosti ogrodja 3iGUI 52
4.5.2 Izrisovalci 62
4.5.3 Življenjski cikel komponent 64
4.5.4 Potek razvoja 67

5 Testiranje in prehod v produkcijo 69
5.1 Zmogljivost ogrodja 70
5.2 Diagnostika komponent in drugih objektov ogrodja 71
5.3 Produkcijsko okolje, kopija produkcije in stik aplikacij z okolico 72

6 Primerjava ogrodij 74
6.1 Razvojno okolje in priprava prve aplikacije 74
6.2 Izvajanje aplikacij in njihovo stanje 76
6.3 Razvoj uporabniških vmesnikov 78
6.4 Povezovanje ogrodij 81

7 Sklepne ugotovitve 84
7.1 Ogrodje v praksi 85
7.1.1 Prednosti 85
7.1.2 Slabosti 86
7.2 Možne izboljšave 87

Seznam slik 89
Seznam tabel 90

Literatura 91

Seznam uporabljenih kratic in simbolov

AJAX	Asynchronous JavaScript and XML - asinhroni JavaScript in XML
API	Application Programming Interface - aplikacijski programski vmesnik
ASP	Active Server Pages - aktivne strežniške strani
CSS	Cascading Style Sheets - predloge za oblikovanje izgleda spletnih strani
CVS	Concurrent Versions System - sistemu za kontrolo hkratnih različic
DI	Dependency injection - vrivanje odvisnosti
DOM	Document Object Model - dokumentni objektni model
EJB	Enterprise JavaBeans - javanska strežniška zrna
GIS	Geographic information system - geografski informacijski sistem
GUI	Graphical user interface - grafični uporabniški vmesnik
GWT	Goolge Web Toolkit - Googlov nabor orodij za razvoj obogatenih spletnih aplikacij (RIA)
HTML	Hypertext Markup Language - jezik za označevanje hiperteksta
HTTP	Hypertext Transfer Protocol - protokol za prenos hiperteksta
IDE	Integrated Development Environment - razvojno okolje
IoC	Inversion of control - obrat kontrole
JAR	Java ARchive - javanska arhivska datoteka
Java EE	Java Platform, Enterprise Edition - poslovna različica javanske podlage
JDBC	Java Database Connectivity - javanska povezljivost podatkovnih baz
JSON	JavaScript Object Notation - zapis podatkov, ki uporablja sintakso objektov v JavaScriptu
JSP	JavaServer Pages - javanske strežniške strani
JVM	Java virtual machine - javanski navidezni stroj

MVC	Model-view-controller - model-pogled-krmilnik
MVP	Model-view-presenter - model-pogled-predstavitelj
ORM	Object-relational mapping - objektno-relacijsko preslikovanje
POJO	Plain Old Java Object - običajni javanski objekt
RIA	Rich Internet Application - obogatena spletna aplikacija
RPC	Remote procedure call - klic oddaljenje procedure
SQL	Structure Query Language - strukturirani jezik za poizvedovanje
URL	Uniform Resource Locator - naslov vira v enotni obliki
XML	Extended Markup Language - razširljiv označevalni jezik

Povzetek

Svetovni splet je hitro razvijajoči se sistem, ki je postal pomembna podlaga za razvoj in izvajanje aplikacij. Ponovna uporaba programske opreme je eden izmed pomembnejših načinov za izgradjo nove programske opreme. Spletna ogrodja so dober primer ponovne uporabe programske opreme. Gre za generatorje aplikacij, ki nam z abstraktnimi razredi, predpripravljenimi komponentami in dobro definiranimi programskimi vzorci pomagajo pri izgradnji spletnih aplikacij.

V diplomski nalogi sem opisal zasnovo in izvedbo lastnega spletnega ogrodja, ga primerjal z dvema obstoječima odprtokodnima ogrodjema in opisal možne povezave med njimi. Zaključil sem, da povezovanje teh ogrodij ponudi dve večji koristi. Razširi lahko množico spletnih aplikacij, ki jih podpira lastno ogrodje in izboljša njegovo zmogljivost. V zaključku diplomske naloge sem navedel prednosti in slabosti razvoja lastnega ogrodja in predlagal možne izboljšave ogrodja.

Ključne besede:

ponovna uporaba programske opreme, ogrodje, zasnova, razvoj, Java

Abstract

The World Wide Web is a rapidly evolving system that has become an important platform for developing and running applications. Software reuse is one of the most important ways to build new software. Web frameworks are a good example of software reuse. They are application generators that help us build web applications by using abstract classes, prefabricated components, and well defined software patterns.

In the thesis I described the design and implementation of a custom web framework, I compared it with two existing open source web frameworks, and described possible bindings among them. I concluded that binding these frameworks offers two major benefits. It may widen the set of web applications that the custom framework supports, and it may increase the custom framework's performance. In the conclusion of the thesis I listed the advantages and disadvantages of custom framework development and suggested possible enhancements of the framework.

Key words:

software reuse, framework, design, development, Java

Poglavje 1

Uvod

Razvoj spletnih aplikacij je v razmahu. Časi, ko je bil splet le skupek statičnih strani HTML in povezav med njimi, so zdavnaj minili. Tekma za prevlado spletnih tehnologij je na začetku bolj ali manj zaposlovala ponudnike spletnih brskalnikov. Rezultat pa so bile velikokrat lastne razširitve narečja HTML. Čeprav je tekma otežila izdelavo združljivih spletnih strani, je hkrati doprinesla nekaj pomembnih novosti, brez katerih si danes ne moremo predstavljati sodobnega spleta. Med njimi zagotovo najbolj izstopa skriptni jezik JavaScript.

Danes je splet podlaga (angl. platform) za naraščajoče število storitev. Povezljivost, sodelovanje, obogatene spletne aplikacije (angl. rich internet application - RIA), spletne storitve ipd. so del tako imenovanega spleta 2.0 (angl. Web 2.0) [5]. Spletni brskalnik je ostal pomemben a hkrati tudi samoumeven del programske opreme osebnih računalnikov in večjega števila drugih naprav. Najdemo ga v dlančnikih, pametnih telefonih, igralnih konzolah, televizijskih aparatih in še kje. Boljša podpora odprtim standardom, ki jim sledijo zadnje generacije spletnih brskalnikov, pa olajša razvoj aplikacij, do katerih lahko dostopamo iz omenjenih naprav. Tekmi spletnih brskalnikov se je pridružila tudi tekma spletnih podlag in orodij, ki jih te ponujajo, kar vodi do večne tekme za naklonjenost razvijalcev.

Tudi klasične namizne (angl. desktop) poslovne aplikacije, ki so namenjene končnemu številu znanih uporabnikov, se selijo v spletni brskalnik. Uporaba odjemalca, ki je privzeto nameščen v vsak operacijski sistem, ima seveda prednosti tudi, ko aplikacija teče zgolj v lokalnem omrežju (na primer namestitev in nadgradnje aplikacije na enem mestu, boljši pregled nad delovanjem, manj težav pri dostopu do zalednih sistemov itd.). Prehod aplikacij iz

namizja v spletni brskalnik je vse prej kot enostaven. Spletni brskalniki niso bili zamišljeni kot podlaga za izvajanje aplikacij, a so se morali temu prilagoditi. Posledice teh prilagoditev so velikokrat nočna mora razvijalcev. Prav zato skupaj z razvojem spleta nastajajo in se razvijajo primerna orodja. Skupna značilnost teh orodij je omogočanje lažjega razvoja aplikacij z uporabo abstrakcije in ponovne uporabe programske kode (angl. software reuse). Zadnja leta smo tako priča razcvetu spletnih ogrodij (angl. web application framework).

V podjetju 3 PORT, d. o. o., razvijamo aplikacije za informacijsko podporo procesom na več področjih, kot so občinski informacijski sistemi, geografski informacijski sistemi (angl. geographic information system - GIS), logistika, transport in druga področja. Ker ponujamo celovite rešitve obsežnih področij, ki večinoma temeljijo na lastnem razvoju, je število različnih aplikacij veliko. Ključna točka pri razvoju aplikacij je zagotovo njihova povezljivost. Pri tem se srečujemo z izzivi povezovanja različnih tehnologij. Pomembna značilnost aplikacij je tudi, da so si v resnici velikokrat sorodne, saj vsebujejo veliko skupnih točk. Sorodnost aplikacij je eden od razlogov za dolgoročno vlaganje v lasten razvoj večjih in manjših ogrodij. V podjetju poleg lastnega razvoja uporabljamo tudi veliko odprtokodnih rešitev, pa naj gre za ogrodja, knjižnice ali končne izdelke, vendar smo se v primeru razvoja spletnih aplikacij odločili za lastno ogrodje. Najpomembnejši razlogi za to so bili:

- Načrtovanje razvoja večjega števila spletnih aplikacij na dolgi rok, ki zahteva stabilno ogrodje oziroma čim manj ključnih sprememb ogrodja, predvsem tistih razredov in vmesnikov, na katere je **neposredno** vezana aplikativna koda.

- Želja po **ponovni uporabi** lastnih rešitev, ki so se že izkazale v praksi in so nastale pred „eksplozijo“ večjega števila sicer primerljivih odprtokodnih spletnih ogrodij v Javi, ki smo ji priča zadnja leta.

- Imeli smo kar nekaj **izkušenj** pri razvoju ogrodij za namizne uporabniške vmesnike, tudi takih, ki omogočajo ločeno (oddaljeno) izvajanje uporabniškega vmesnika od pripadajoče aplikativne kode, torej ogrodij z arhitekturo, na kateri temeljijo tudi spletna ogrodja.

- Izkušnje pri **neposredni** uporabi nekaterih odprtokodnih rešitev so govorile v prid njihovi uporabi, vendar se je izkazalo, da je smiselno zaradi boljšega obvladovanja sprememb novejših različic **ograditi** take rešitve z uporabo lastnih vmesnikov. Tam kjer tako razslojevanje kode ni bilo mogoče (ogrodje za uporabniške vmesnike), smo se odločili za lastno

rešitev, ki bi se v zaledju še vedno lahko povezovala z drugimi (odprtokodnimi) rešitvami.

V diplomski nalogi sem **predstavil razvoj** lastnega spletnega ogrodja, ogrodje sem **primerjal** z dvema odprtokodnima rešitvama, nakazal sem možne **povezave** med ogrodji ter opisal njihove prednosti in slabosti. Za vsa opisana in primerjana ogrodja je značilno, da omogočajo razvoj spletnih aplikacij z uporabo pristopov, ki so sicer značilni za razvoj klasičnih namiznih aplikacij.

Drugo poglavje diplomske naloge vsebuje razlago ogrodij in možne klasifikacije, tretje pa opis spletnih ogrodij s primerjavami različnih arhitektur. To poglavje zaključi še opis dveh odprtokodnih spletnih ogrodij ter podrobnejših razlogov za razvoj lastnega spletnega ogrodja. Osrednja poglavja opisujejo razvoj in arhitekturo spletnega ogrodja ter njegovo vpeljevanje v produkcijo. Diplomsko nalogo zaključita poglavji s primerjavo ogrodij, z možnostmi povezovanja med njimi in z opisom prednosti in slabosti, ki so se pokazale pri uporabi ogrodja v praksi. Zaključek diplomske naloge vsebuje tudi opis možnih izboljšav lastnega ogrodja.

Poglavje 2

Ogrodje

Izgradnje novega sistema si skoraj ne moremo predstavljati brez **ponovne uporabe**. Pri tem gre tako za ponovno uporabo standardnih gradnikov nekega sistema kot tudi za uporabo znanja oziroma izkušenj. Objektno usmerjeno programiranje, knjižnice, komponente, vzorci (angl. pattern) in ne nazadnje tudi ogrodja so tipični primeri omogočanja oziroma spodbujanja ponovne uporabe pri razvoju programske opreme. Cilj, ki ga želimo doseči s ponovno uporabo, je lažji in hitrejši razvoj ter vzdrževanje in s tem zmanjšanje stroškov ter povečanje kvalitete končnih izdelkov.

Ogrodje je abstraktna zasnova (angl. design), ki je sestavljena iz **komponent** in pripadajočih **abstraktnih razredov**. Za razliko od knjižnic (ki predstavljajo večinoma neodvisen skupek splošno uporabne kode) so ogrodja abstraktna zasnova, ki je prilagojena točno določenemu tipu aplikacij [3]. Markiewicz in Lucena [4] definirata ogrodja kot generatorje aplikacij, ki so neposredno vezani na določeno domeno (družino povezanih problemov).

Komponenta je ponovno uporaben programski gradnik, ki ga lahko povezujemo z drugimi komponentami na tak način, da z njimi sestavimo aplikacijo [23]. Enega bolj očitnih primerov uporabe komponent predstavljajo grafični uporabniški vmesniki (angl. graphical user interface - GUI). Tipičen uporabniški vmesnik sestavlja večje število komponent, kot so gumbi, vnosna polja, spustni meniji ipd. Isti tip komponente je lahko uporabljen večkrat v isti aplikaciji in tudi v več različnih aplikacijah. Prednosti uporabe standardnih komponent v tem primeru niso samo na strani razvijalcev, ki tako reciklirajo kodo, temveč tudi na strani uporabnikov, ki se uporabe standardnih komponent učijo samo enkrat.

Pomembna značilnost ogrodja je implementacija načela, ki mu pravimo **obrat kontrole** (angl. inversion of control - IoC). Pojem predstavlja načelo pri snovanju programske arhitekture, kjer je kontrola poteka izvajanja obrnjena v primerjavi s klasičnim proceduralnim programiranjem [7]. V primeru klasično razvite aplikacije je koda aplikacije tista, ki nadzira potek programa, tako da sama gradi instance objektov in kliče njihove metode. Aplikacija ima torej kontrolo nad potekom izvajanja. V primeru uporabe ogrodja pa je le-to zadolženo za grajenje instanc in klice specifičnih metod, ki jih implementira aplikacija. Obrat kontrole je pomemben dejavnik, ki ločuje knjižnice od ogrodij. Primerjava knjižnice z ogrodjem je podana v tabeli 2.1.

knjižnica	**ogrodje**
množica razredov, ki jih instancira uporabnik[1]	omogoča prilagajanje z izpeljevanjem razredov
uporabnik kliče metode	kliče uporabnikove metode
ni v naprej določenega poteka izvajanja	ima kontrolo nad potekom izvajanja
ni v naprej določenega medsebojnega delovanja (angl. interaction)	določa medsebojno delovanje objektov
nima privzetega obnašanja	vsebuje privzeto obnašanje

Tabela 2.1: Primerjava knjižnice z ogrodjem [17].

Obratu kontrole pravimo tudi **hollywoodsko načelo** (angl. Hollywood principle) [8]. Ime je dobilo po znanem odgovoru: „Ne kličite nas, mi vas bomo poklicali,“ ki ga dobijo amaterski igralci na filmskih avdicijah v Hollywoodu. Namen uporabe obrata kotrole je predvsem [7]:

- odpraviti sklopljenost (angl. coupling) med izvajanjem in implementacijo določenega opravila,
- sistem se lahko osredotoči samo na to, za kar je bil zasnovan,
- sistem ne predvideva, kaj počnejo oziroma kaj bi morali početi drugi sistemi,
- zamenjava določenih sistemov nima vpliva na druge sisteme.

[1] Kot uporabnik je tukaj mišljena koda, ki uporablja knjižnico oziroma koda aplikacije, ki temelji na uporabi ogrodja.

Posebna oblika obrata kontrole je **vrivanje odvisnosti** (angl. dependency injection - DI) [9]. Gre za način „oskrbovanja“ komponent z drugimi komponentami, kjer prve potrebujejo storitve drugih. Glavni cilj vrivanja odvisnosti je zmanjšanje sklopljenosti sistema.

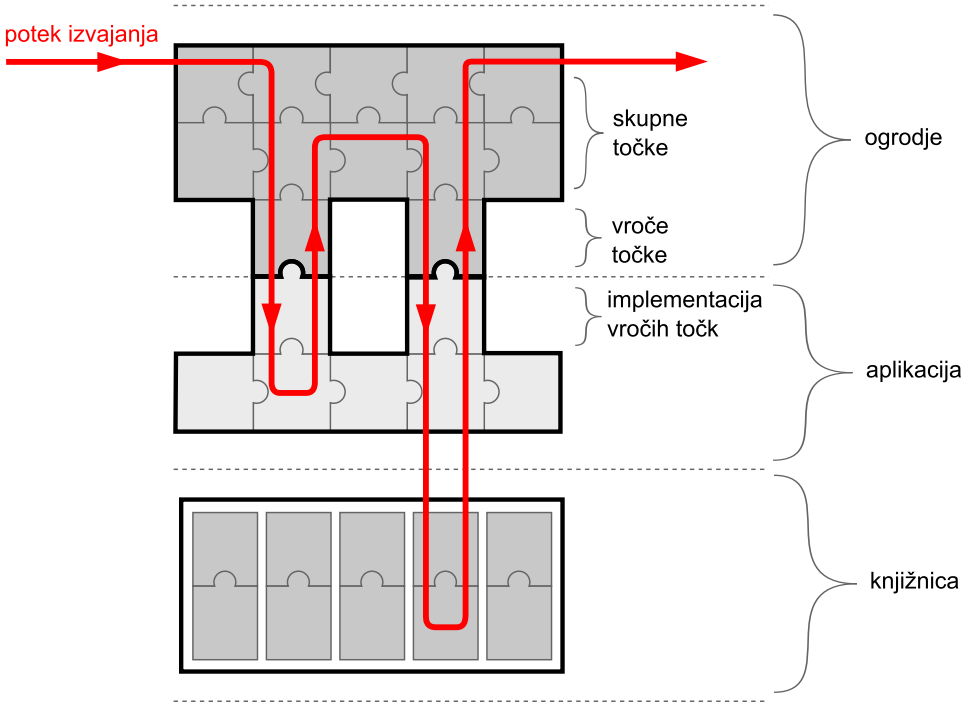
2.1 Skupne in vroče točke ogrodja

Ogrodje predstavlja zametek aplikacije, zato ga ni mogoče izvajati samostojno. Za izvajanje potrebujemo instanco aplikacije. Pripravimo jo z implementacijo metod in vmesnikov, ki jih v ta namen ponuja ogrodje. Točkam v kodi, ki omogočajo razširitev ogrodja, pravimo **vroče točke** (angl. hot spot) [4, 20]. Vroče točke so v naprej pripravljeni deli ogrodja, ki jih „zapolnimo“ s specifično aplikativno kodo. Implementacija vročih točk omogoča **prilagoditev oziroma spremembo obnašanja objektov ogrodja**, ne da bi bilo potrebno za to spreminjati njihovo kodo. Poleg vročih točk pa ogrodje vsebuje tudi točke, ki predstavljajo njegov nespremenljivi del. Te točke imenujemo **skupne točke** (angl. common spot) [20] ali tudi **zamrznjene točke** (angl. frozen spot) [4]. Skupne točke so deli kode, ki se v določeni družini aplikacij pojavljajo vedno v isti obliki. Prav zato se te točke pri snovanju ogrodja izluščijo iz množice podobnih aplikacij in preselijo v ogrodje. Skupne točke preprečujejo ponavljanje kode v aplikacijah in so pomemben dejavnik ponovne uporabe kode.

Na sliki 2.1 je vidna vloga skupnih in vročih točk v povezavi z obratom kontrole. Skupne in vroče točke so si vsebinsko gledano nasprotne. Če prve vsebujejo kodo, ki je skupna določeni družini aplikacij, druge predstavljajo prazna mesta v ogrodju, ki so namenjena specifični aplikaciji. Zasnova skupnih in vročih točk je ena pomembnejših nalog, ki jih mora opraviti snovalec ogrodja. Pri tem pa je eno glavnih vprašanj, na katerega mora odgovoriti, kako omogočiti vroče točke. Možna sta dva pristopa, in sicer z uporabo **dedovanja** ali **kompozicije** [20].

2.1.1 Omogočanje vročih točk z dedovanjem

Pristop omogočanja vročih točk z uporabo **dedovanja** predvideva, da ogrodje vsebuje enega ali več abstraktnih razredov, ki so osnova za izgradnjo komponent. Naloga razvijalca aplikacije je, da izpelje predvidene abstraktne razrede in implementira vse abstraktne metode oziroma da „zapolni“ manjkajoče dele ogroda s specifično aplikativno kodo.



Slika 2.1: Vloga skupnih in vročih točk v povezavi z obratom kontrole

Abstraktni razredi ogrodja so navadno sestavljeni iz končnih in abstraktnih metod. Končne metode so lahko **šablone** (angl. template method) [20]. Šablona vsebuje kodo, ki določa sodelovanje med posameznimi metodami. Ta koda kliče metode ogrodja in vroče točke v točno določenem vrstnem redu. Kot je opisano v nadaljevanju[2], ni potrebno, da so vroče točke vedno pripravljene v obliki abstraktnih metod. Snovanje in uporaba ogrodij imata nekaj zanimivih značilnosti:

- Pri razvoju aplikacije oziroma njenih komponent se osredotočimo na vroče točke. Pri tem nas ne zanima, kdaj in kako se bodo izvedli klici šablon in posledično aplikativne kode.
- Šablone, ki kličejo implementirane vroče točke, so del ogrodja. Klicatelj šablon je ogrodje, ki ima kontrolo nad potekom izvajanja. Prednost opisanega obrata kontrole je v tem, da lahko ogrodje na tak način **pripravi idealne pogoje** pred klicem vroče točke in poskrbi za druge

[2]razdelek 2.1.2

potrebne akcije po klicu vroče točke. Na ta način je aplikativna koda velikokrat razbremenjena „navlake“, ki je tipično prisotna v aplikacijah, a nima neposredne veze s problemom, ki ga rešujemo. Seveda pa je cena, ki jo plačamo za omenjeno razbremenitev, bolj „toga“ koda. Ni si težko predstavljati, da si lahko s slabo zasnovo ogrodja hitro zavežemo roke.

- Tretja značilnost je možnost nadgradnje ogrodja, ki se pokaže šele pri njegovi uporabi. Pri razvoju aplikacije lahko še vedno odkrijemo izseke splošno uporabne kode, ki jih premaknemo v nove metode. Te lahko kasneje postanejo del ogrodja. Način izgradnje ogrodja od spodaj navzgor, kjer izhajamo iz konkretnih primerov in iz njih poskušamo izluščiti skupne značilnosti, ki se nato preselijo v ogrodje, je tipičen pristop pri izgradnji ogrodij [3].

Pri izpeljevanju zapletenih komponent lahko podrazredi dedujejo tudi večje število lastnosti svojega starša. Nekatere abstraktne komponente lahko celo zahtevajo, da izpeljani razredi uporabijo podedovane lastnosti na točno določen način. Posledično mora razvijalec dobro razumeti „notranje“ delovanje abstraktnih razredov, iz katerih izpeljuje svoje razrede. Tak način dela zato tipično zahteva več privajanja in boljše razumevanje delovanja samega ogrodja, kot bi to sicer želeli. Možno rešitev predstavlja pristop k zasnovi ogrodja, ki namesto dedovanja izkorišča kompozicijo objektov.

2.1.2 Omogočanje vročih točk s kompozicijo

Pri uporabi **kompozicije** se tako kot v primeru uporabe dedovanja poslužujemo značilnosti objektno usmerjenega programiranja. Kompozicija omogoča sestavljanje enostavnejših objektov v bolj zapletene. Prednost, ki jo obljublja uporaba kompozicije, je zakrivanje podrobnosti, ki so del „notranjega“ stanja ogrodja.

Pri uporabi kompozicije **so vroče točke določene z množico vmesnikov** (angl. interface), ki so dobro povezani z ogrodjem. Razvijalci pripravijo nove komponente z implementacijo teh vmesnikov. Take komponente lahko nato povežejo z ogrodjem in s tem prilagodijo njegovo delovanje. Ogrodje v ta namen predvidi točke povezovanja [20].

Pri implementaciji komponent je pomembna razlika med uporabo dedovanja in kompozicije v tem, da pri slednji implementiramo komponento v celoti in ne

samo določenega dela. Tako se pri implementaciji ne srečujemo s kodo, ki bi jo naše komponente sicer podedovale od delov ogrodja. Razumevanje uporabe komponent je osredotočeno samo na nekaj dobro definiranih vmesnikov, s samo implementacijo ogrodja pa se ni potrebno ukvarjati.

Ko s pomočjo ogrodja pripravimo primerne komponente, jih je potrebno povezati z izvajalnim delom ogrodja. Za to nalogo potrebujemo objekt, ki mu pravimo **vsebnik** (angl. container). Vsebnik je razred, čigar instance predstavljajo zbirke drugih objektov [6]. Vsebnik vzdržuje seznam komponent in jim nudi okolje za izvajanje. Pri tem je lahko tudi sam komponenta ogrodja, kar privede do hierarhije komponent povezanih v drevesno strukturo.

2.2 Klasifikacija ogrodij

Ogrodja lahko klasificiramo glede na domeno (angl. problem domain), za katero so bila razvita, in glede na njihovo notranjo zgradbo. Glede na **domeno** jih klasificiramo na [17]:

- **Aplikacijska ogrodja** (angl. application framework) - obsegajo širše uporabne funkcionalnosti na različnih uporabniških domenah. Tipični predstavniki so ogrodja za izdelavo uporabniških vmesnikov, ki omogočajo izgradnjo širokega spektra različnih aplikacij za različna področja uporabe.

- **Domenska ogrodja** (angl. domain framework) - zajemajo strokovno znanje ožje določenih domen (na primer ogrodja za multimedijo, ogrodja za dostop do podatkov, orodja za varno trgovanje itd.)

- **Podporna ogrodja** (angl. support framework) - nudijo sistemske storitve (na primer ogrodja za dostop do datotek, podpora porazdeljenim računalniškim sistemom, gonilniki naprav itd.).

Klasifikacija ogrodij glede na njihovo **notranjo zgradbo** loči ogrodja glede na to, kako omogočajo vroče točke. Ogrodja razdeli po načelih [3, 4, 20] **bele škatle** (angl. white-box), **črne škatle** (angl. black-box) in **sive škatle** (angl. gray-box).

2.2.1 Ogrodja po načelu bele škatle

Ogrodja po načelu bele škatle temeljijo na množici abstraktnih razredov. Za omogočanje vročih točk uporabljajo pristop z **dedovanjem**. Naloga razvijalca pri uporabi takega ogrodja je izpeljati abstraktne razrede ogrodja in implementirati njihove abstraktne metode. Zaradi uporabe dedovanja pa mora razvijalec dobro poznati delovanje ogrodja in arhitekturo razredov. Posledično taka ogrodja imenujemo tudi **arhitekturno gnana ogrodja** (angl. architecture-driven framework) [17].

Prednosti te vrste ogrodij so:

- Razvoj ogrodij je enostavnejši, saj ni potrebno predvideti strogih vmesnikov in vseh možnih povezav med komponentami, ki jih sicer potrebujemo pri sestavljanju komponent s kompozicijo.
- Aplikacije vsebujejo manj kode, ki služi za medsebojno povezovanje komponent.
- Omogočajo bolj prilagodljiv pristop k razvoju aplikacij in nadgradnjam ogrodja. Z izpeljevanjem razredov namreč omogočajo dodajanje novih funkcionalnosti, ki se lahko kasneje preselijo v ogrodje. Pri teh ogrodjih ni potrebno v naprej predvideti strogo določenih vmesnikov, temveč se ti lahko razvijejo postopoma glede na potrebe, ki se pokažejo v praksi.

Slabosti te vrste ogrodij so:

- Razvijalci morajo dobro poznati delovanje ogrodja. Posledica je bolj strma učna krivulja uporabe ogrodja.
- Uporaba končnih šablon, ki kličejo vroče točke, lahko privede do bolj „toge“ kode. Končne šablone ne dovolijo izpeljevanja, zato lahko preveč omejijo vrstni red klica metod in sodelovanja med objekti, ki ga ne moremo obiti samo z drugačno implementacijo vročih točk.
- Odpiranje „notranjosti“ razredov ogrodja lahko privede do aplikacij, ki izkoriščajo ogrodje na nepredviden način. To lahko oteži nadaljnji razvoj ogrodja, ki želi biti združljivo z obstoječimi aplikacijami, saj mora tako ogrodje upoštevati tudi nepredvidene scenarije, katerih uporaba ni bila dobro načrtovana od samega začetka. V primeru, da pri nadgradnji ogrodja ne ohranimo združljivosti, pa lahko povzročimo množico

nedelujočih aplikacij, ki jih je potrebno prilagoditi novi različici ogrodja. To lahko v skrajnem primeru popolnoma izniči prednosti uporabe ogrodja.

- Za taka ogrodja je težje razviti nabor pomožnih orodij (angl. toolkit) oziroma generatorjev kode.

2.2.2 Ogrodja po načelu črne škatle

Ogrodja po načelu črne škatle predvidevajo razvoj aplikacij z omogočanjem vročih točk s **kompozicijo**. Pri tem vsebuje posamezno ogrodje množico v naprej pripravljenih komponent, ki jih s povezovanjem sestavimo v končno aplikacijo. Razvoj aplikacije tako sestoji iz sestavljanja različnih komponent in dodajanja lastnih objektov, ki implementirajo predvidene vmesnike. S takim sestavljanjem lahko razvijalec določi značilnosti aplikacije, medtem ko **ogrodje določa pravila pri sestavljanju komponent**. Pomembna razlika glede na ogrodja po načelu bele škatle je, da pri teh ogrodjih razvijalcem ni potrebno poznati „notranjega“ delovanja ogrodja oziroma njegovih komponent. Ker koda aplikacij, ki so zgrajene s takimi ogrodji, temelji na sestavljanju in nastavljanju komponent, se ta ogrodja imenujejo tudi **podatkovno gnana ogrodja** (angl. data-driven framework) [17].

Prednosti te vrste ogrodij so:

- Uporaba ogrodja je tipično lažja, saj predvideva sestavljanje že pripravljenih komponent. Pri tem se lahko razvijalec osredotoči samo na dobro definirane javne vmesnike, ni mu pa potrebno poznati „notranjega“ delovanja komponent.

- Omogočajo lažji razvoj pomožnih orodij. Tipični primer so orodja za izris uporabniških vmesnikov. Če imamo v naprej pripravljen nabor standardnih komponent z znanimi vmesniki, lahko razvijemo pomožno orodje, ki bo končnemu uporabniku omogočalo izbiro komponent in definiranje povezav med njimi.

- Ker taka ogrodja ne izpostavljajo svoje „notranjosti“ končnim aplikacijam, so z vidika nadgrajevanja bolje obvladljiva. Razvijalci ogrodja imajo lahko z uporabo načela črne škatle veliko boljšo kontrolo nad pravilno uporabo ogrodja, kot bi jo imeli, če bi končnim aplikacijam omogočili izpeljevanje in posledično dedovanje lastnosti objektov, ki so del ogrodja.

Slabosti te vrste ogrodij so:

- Razvoj ogrodij je težji, saj je potrebno že v naprej dobro predvideti vmesnike vseh komponent ter njihovo medsebojno sodelovanje in povezovanje. Nasprotno pri uporabi dedovanja to ni tako nujno, saj lahko aplikacije do neke mere same dodajajo morebitne manjkajoče funkcionalnosti.

- Uporaba ogrodij je lahko bolj „toga“, saj je razvijalec omejen s končnim naborom vmesnikov in zato nima toliko možnosti za lastne razširitve funkcionalnosti.

- Razvijalec aplikacije mora tipično povezati večje število komponent in drugih objektov, da bi z njimi dosegel željeno obnašanje aplikacije. S tem se velik del razvoja osredotoči na kodo, ki predstavlja „ožičenje“ komponent. Prav nasprotno pa ogrodja po načelu bele škatle že privzeto vsebujejo kodo za „ožičenje“, ki je ni potrebno pisati razvijalcem aplikacij.

2.2.3 Ogrodja po načelu sive škatle

Oba predhodno opisana tipa ogrodij sta v resnici skrajna primera. V praksi se večinoma srečujemo z ogrodji, ki uporabljajo tako načelo bele kot črne škatle. Tem ogrodjem pravimo ogrodja po načelu sive škatle. Navadno združujejo enostavnost uporabe (uporaba kompozicije) in razširljivost (možnost izpeljevanja razredov). Taligent [17] na primer omenja pristop k razvoju ogrodij, ki imajo za osnovo predvideno kodo po načelu bele škatle, nad njo pa sloj kode po načelu črne škatle. Seveda se pri uporabi obeh pristopov poskuša izkoristiti čim več njunih prednosti oziroma zmanjšati njune slabosti.

Johnson in Foote [3] omenjata uporabo obeh pristopov razvoja ogrodij v časovnem kontekstu. Začetne različice ogrodja naj bi bile bolj usmerjene k načelu bele škatle. Ko še nimamo dovolj izkušenj z uporabo ogrodja, prepustimo več možnosti za širitev funkcionalnosti končnim aplikacijam. Na začetku je potrebno pazljivo določati vmesnike, da ne bi prehitro omejili razvoja ogrodja. Sčasoma se splošno uporabni deli nove kode preselijo iz aplikacij v ogrodje. Skozi posamezne iteracije razvoja aplikacij in nadgradenj ogrodja nastaja knjižnica dobro definiranih vmesnikov in splošno uporabnih komponent. Končni cilj, ki ga omenjata avtorja, je ogrodje po načelu črne škatle, ki bolje izkorišča ponovno uporabo kode. Načelo črne škatle označita kot ideal, h kateremu naj bi se približal vsak sistem.

2.3 Prednosti in slabosti ogrodij

Preden se odločimo za uporabo ogrodja ali celo za njegov razvoj, je potrebno pretehtati prednosti in slabosti takega pristopa. Če razvijamo zelo enostavno aplikacijo, se bomo morda lahko izognili uporabi ogrodja in prihranili čas, ki bi bil sicer potreben za njegovo uvajanje. V večini primerov pa se bomo že pri razvoju ene same aplikacije morali soočiti z uporabo enega ali več splošno uporabnih ogrodij (na primer pri grajenju uporabniškega vmesnika in dostopu do podatkovne baze). V primeru, da imamo v načrtu razvoj večjega števila sorodnih aplikacij, pa se lahko pojavi tudi vprašanje razvoja lastnega ogrodja. To vprašanje je še težje, saj bomo za razvoj ogrodja porabili najmanj toliko časa kot bi ga za razvoj aplikacije brez ogrodja. Poleg tega je potrebno predvideti še dokumentacijo, učne aplikacije (angl. tutorial), čas za uvajanje razvijalcev in ne nazadnje tudi bodoče vzdrževanje in nadgrajevanje ogrodja. V nadaljevanju so opisane glavne prednosti in slabosti pri uporabi in razvoju ogrodij [1].

Prednosti ogrodij:

- Omogočajo ponovno uporabo predpripravljene in stestirane kode. S tem povečajo zanesljivost novih aplikacij in zmanjšujejo trud pri njihovem programiranju in testiranju.
- Lahko pripomorejo k vzpostavitvi dobre prakse programiranja ter primerne uporabe programskih vzorcev in novih pomožnih orodij. Nadgradnja ogrodja lahko doprinese nove funkcionalnosti, boljšo zmogljivost ali boljšo kvaliteto, brez da bi bilo za to potrebno spreminjati kodo aplikacij.
- Omogočajo razširitev privzetega obnašanja.

Slabosti ogrodij:

- izgradnja ogrodja je lahko zahtevna in časovno potratna (draga),
- strma učna krivulja privajanja na novo ogrodje,
- zapletenost ogrodja se lahko s časom povečuje.

Poglavje 3

Spletna ogrodja

Spletna ogrodja omogočajo lažji in hitrejši razvoj spletnih aplikacij. Gre za aplikacije, do katerih dostopamo prek računalniškega omrežja (na primer internet ali intranet) s pomočjo brskalnika [10]. Za razliko od klasičnih namiznih aplikacij jih ni potrebno posebej nameščati na odjemalce. Podobno je z njihovim vzdrževanjem, saj je (po nadgradnji aplikacije na strežniku) nova različica takoj dostopna vsem uporabnikom. Prav tako so take aplikacije večinoma neodvisne od operacijskega sistema končnega uporabnika, kar predstavlja še eno izmed prednosti, zaradi katerih so te aplikacije vse bolj razširjene.

Spletna aplikacija se lahko izvaja na strežniku ali v brskalniku (odjemalcu). V primeru izvajanja na **strežniku** se v brskalnik prenašajo samo njeni pogledi (angl. view), ki so največkrat sestavljeni z označevalnim jezikom HTML. Brskalnik poskrbi za prenos uporabnikovega zahtevka z novimi ali spremenjenimi podatki na strežnik, kjer jih ta posreduje v obdelavo primerni instanci spletne aplikacije. Rezultat obdelave podatkov je nov pogled, ki ga aplikacija s pomočjo strežnika pošlje nazaj v brskalnik. Opisan „pogovor“ poteka po vzorcu zahtevek-odgovor (angl. request-response pattern) z uporabo protokola HTTP.

Spletne aplikacije, ki se izvajajo **v brskalniku**, se pred začetkom izvajanja delno ali v celoti naložijo iz strežnika v brskalnik. Med potekom izvajanja lahko dostopajo do strežnika, ko potrebujejo dodatne podatke za obdelavo, osvežitev uporabniškega vmesnika ali celo del izvajalne kode (dodatni moduli aplikacije). V takih primerih največkrat govorimo o obogatenih spletnih aplikacijah, ki poskušajo posnemati delovanje klasičnih namiznih aplikacij. Pri tem lahko za izvajalno okolje uporabijo kar spletni brskalnik z jezikom JavaScript. Z

JavaScriptom imajo omogočen dostop do programskega vmesnika (angl. application programming interface - API), ki omogoča dinamično spreminjanje HTML-ja. Dokument HTML je v brskalniku predstavljen z drevesno podatkovno strukturo, ki ji pravimo dokumentni objektni model (angl. document object model - DOM). Ta omogoča pregledovanje vsebine dokumenta in njeno spreminjanje. Obogatene spletne aplikacije lahko uporabijo DOM za izris svojih uporabniških vmesnikov in njihovo ažuriranje. Odzivnost uporabniškega vmesnika je veliko boljša, če sproti ažurirajo manjše število vozlišč DOM-a. Pri tem lahko sodeluje tudi strežnik, od katerega pridobijo ažurne podatke. Pri razvoju takih aplikacij uporabljamo množico tehnik, ki so znane po imenu asinhroni JavaScript in XML (angl. asynchronous JavaScript and XML - AJAX). Kratica AJAX se navezuje na asinhroni način povezovanja aplikacij s strežnikom, kjer ostane uporabniški vmesnik aplikacije odziven tudi med časom obdelave zahtevka na strežniku. XML predstavlja zgolj enega od možnih formatov za prenos podatkov, ki jih uporabljata brskalnik in strežnik. Strežnik lahko podatke vrne v več možnih formatih, kot so na primer XML, HTML in JSON. Namesto JavaScripta lahko obogatene spletne aplikacije uporabijo tudi izvajalna okolja, ki so del razširitev brskalnika. Primeri takih razširitev so javanski navidezni stroj (angl. Java virtual machine - JVM), Flash in Silverlight.

Posebno poglavje pri razvoju spletnih aplikacij je **vzdrževanje stanja**. Gre za problem, ki je večinoma tuj namiznim aplikacijam in je posledica delovanja protokola HTTP. Ta je namreč brez stanja (angl. stateless), kar pomeni, da se vsak zahtevek izvede neodvisno od prej izvedenih zahtevkov. Vzdrževanje stanja aplikacije pomeni možnost, da ima aplikacija pri obdelavi zahtevka dostop do podatkov, ki so bili obdelani v prejšnjih zahtevkih. Pri tem je pogovor med odjemalcem in strežnikom navadno vezan na nek kontekst. Primer takega konteksta je **seja**. Seja je pogovor med odjemalcem in strežnikom, ki se lahko začne s prijavo uporabnika v sistem in traja dokler je strežnik ne zaključi. Strežnik lahko to stori ob točno določenem dogodku (na primer odjava uporabnika) ali pa po daljši neaktivnosti odjemalca. Na strežniku je seja implementirana kot vsebnik, ki vzdržuje množico objektov. Ti objekti so dostopni vsem zahtevkom v seji in se sprostijo šele ob njenem zaključku.

Vzdrževanje stanja aplikacije ni mogoče brez sodelovanja odjemalca. Ta mora podpirati vsaj eno od tehnik, ki omogočajo trajnost stanja. Tehnike se lahko razlikujejo in so odvisne tudi od tipa spletne aplikacije. Aplikacije, ki se izvajajo na odjemalcu, so glede stanja zelo podobne namiznim aplikacijam. Se-

stavljajo jih objekti, ki se med izvajanjem aplikacije naložijo v pomnilnik in tam po potrebi ostanejo do zaključka izvajanja. Posebne skrbi za vzdrževanje stanja v takih primerih ni. Klici na strežnik, ki jih opravijo take aplikacije, so tipično klici oddaljenih procedur (angl. remote procedure call - RPC), kjer aplikacija v zahtevku vedno poda vso potrebno informacijo, s pomočjo katere lahko strežnik vrne odgovor. Zahtevki lahko v takih primerih vsebujejo tudi del trenutnega stanja aplikacije, če ga strežnik potrebuje za pripravo odgovora.

Spletne aplikacije, ki se izvajajo na strežniku, so nasprotno popolnoma odvisne od tehnik za vzdrževanje stanja. Pri tem je stanje prav tako sestavljeno iz množice objektov, ki pa brez pravega mehanizma ne obstanejo v pomnilniku med posameznimi zahtevki. Ogrodja morajo rešiti dve osnovni težavi, in sicer **kam shraniti** stanje aplikacije in **kako povezati** naslednji zahtevek z enim izmed predhodno shranjenih stanj. Ogrodje lahko med drugim shrani stanje aplikacije v pomnilnik, v zaledno podatkovno bazo, datotečni sistem ali pa stanje zakodira in pošlje na odjemalca skupaj z rezultatom zahtevka. Pred obravnavanjem naslednjega zahtevka mora ogrodje določiti pravo stanje, ga po potrebi naložiti nazaj v pomnilnik in aplikaciji omogočiti dostop do stanja. Še najmanj težav pri iskanju pravega stanja (teh je v določenem trenutku na strežniku lahko zelo veliko) predstavljajo stanja, ki se sproti pošiljajo na odjemalca. Odjemalec mora vsakemu zahtevku priložiti tudi zakodirano stanje, ki ga je nazadnje dobil od strežnika. Strežniku tako ni potrebno iskati stanja, saj je to vedno prisotno v zahtevku. Prav tako mu ni potrebno skrbeti za shrambo stanj. Ta pristop pa ima tudi nekaj slabosti, ki so predvsem pretakanje večje količine podatkov, stanje ne sme vsebovati varnostno kritičnih podatkov, preslikovanje stanja iz objektov v zakodirano obliko in nazaj je časovno in procesorsko potratno. Drugi načini shranjevanja predvidevajo, da se stanje označi z enolično identifikacijsko oznako. K odgovoru strežnika se doda tudi oznaka stanja, ki jo odjemalec kasneje „vrača“ z vsakim zahtevkom. Ta oznaka je lahko shranjena kot piškotek (angl. cookie), del naslova URL ali na primer kot skrito polje spletnega obrazca (angl. web form). Odvisno od konteksta pogovora je lahko identifikacija stanja vezana tudi na druge informacije o uporabniku, kot je na primer številka uporabnikovega spletnega digitalnega potrdila.

Spletna ogrodja podpirajo razvoj ključnih elementov spletnih aplikacij, ki med drugim zajemajo [11]: varnost (avtentikacija in avtorizacija uporabnikov), enostaven dostop do različnih podatkovnih baz, preslikovanje naslovov URL, sisteme za upravljanje s spletnimi predlogami, predpomnjenje pogosto zahtevanih

virov, podporo tehniki AJAX, samodejno upravljanje z nastavitvami in implementacijo spletnih storitev (angl. web service).

3.1 Klasifikacija spletnih ogrodij

Jezikov in podlag, v katerih lahko razvijamo spletne aplikacije, je veliko. Med njih spadajo na primer: Java, PHP, Perl, .NET, Ruby, Python, Groovy, JavaScript itd. Velika večina je prisotnih na različnih operacijskih sistemih. Za vsakega izmed opisanih jezikov je v splošnem na voljo večje število spletnih ogrodij. Velikokrat se avtorji ogrodij zgledujejo po rešitvah napisanih v drugih jezikih, zato so si ogrodja pogosto tudi podobna. To je še posebej značilno za odprtokodna ogrodja. Ena vodilnih podlag po številu spletnih ogrodij je zagotovo Java. Ker je Java programski jezik, s katerim je bilo razvito ogrodje, ki ga opisujem v naslednjih poglavjih, se bom v nadaljevanju navezoval na primere, ki so vezani na Javo.

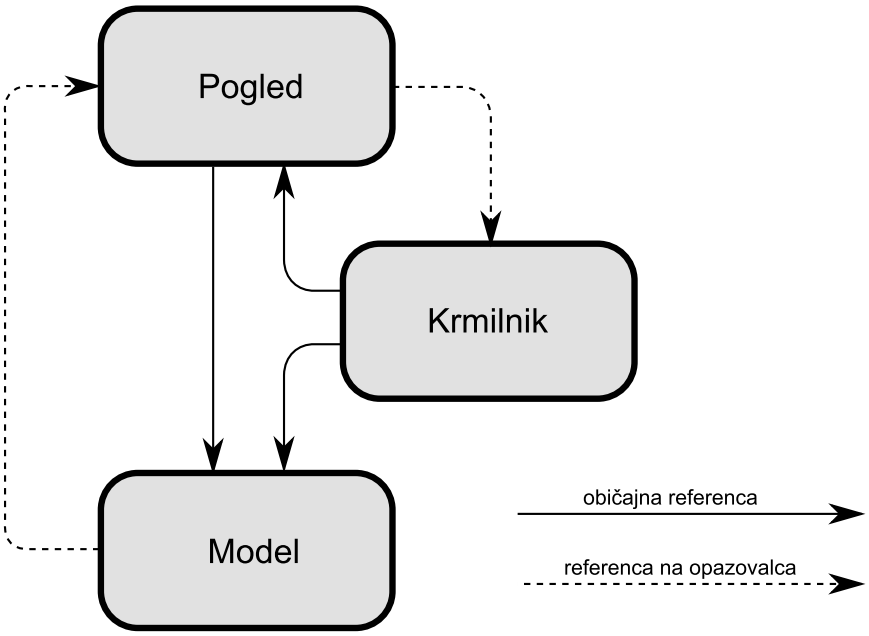
Spletna ogrodja lahko klasificiramo glede na arhitekturo, ki določa način in mesto procesiranja zahtevkov. Ločimo dva pomembnejša modela arhitekture model 1 in model 2. **Model 1** [19] je enostavnejša arhitektura, ki je tesno vezana na vzorec zahtevek-odgovor. Zahtevek se večinoma obdela na enem mestu, kjer koda poskrbi za branje in interpretacijo vhodnih parametrov zahtevka, izvajanje aplikacijske logike in generiranje odgovora (na primer generiranje dokumenta HTML). V Javi lahko tako arhitekturo izkoristimo z enostavno uporabo **servletov** (Java servlet) in strani **JSP**. Arhitektura model 1 je primerna za razvoj spletnih aplikacij sestavljenih iz dinamičnih strani, ki vsebujejo enostavne in večinoma statične prehode med stranmi. Za prehode se v takih primerih uporabljajo kar običajne povezave (angl. link) v dokumentih HTML.

Model 1 je sicer enostaven, a lahko z večanjem aplikacije hitro zaidemo v težave, ki se navadno kažejo kot ponavljanje enake kode na več mestih, prevelika sklopljenost med podatki in njihovo predstavitvijo ter zapletena in nepregledna logika prehajanja med stranmi. Rešitev težav omogoča uporaba vzorca **model-pogled-krmilnik** (angl. model-view-controller - MVC) [12]. Gre za vzorec arhitekture, ki omogoča šibko sklopljenost (angl. loose coupling) med posameznimi „sloji“ aplikacije, ki so razdeljeni na model, pogled, krmilnik [12, 18], kot je prikazano na sliki 3.1.

- **Model** - predstavlja poslovne podatke in pravila za dostop do njih

oziroma za njihovo spreminjanje. Model lahko obvešča poglede o spremembah podatkov (z uporabo vzorca „opazovalec“) ter omogoča poizvedovanje po podatkih.

- **Pogled** - riše vsebino modela. Dostopa do podatkov iz modela in določa njihovo predstavitev. Pogled ažurira predstavitev podatkov vsakokrat, ko pride do sprememb v modelu. Poleg tega je pogled zadolžen za obveščanje o uporabnikovih akcijah krmilniku.
- **Krmilnik** - definira obnašanje aplikacije. Zadolžen je za razpošiljanje vhodnih podatkov in izbiro primernih pogledov za predstavitev podatkov. Interpretira uporabnikove vhodne podatke ter jih nato preslika v akcije, ki jih je potrebno opraviti na modelu.



Slika 3.1: Vzorec MVC

Model 2 [19] temelji na vzorcu MVC. Tam, kjer model 1 uporabi množico med seboj odvisnih dinamičnih strani, model 2 vpelje **krmilnik** kot osrednji modul, ki določa obnašanje aplikacije. Krmilnik je navadno servlet, ki je postavljen

med brskalnik in dinamične strani (poglede). Slednje lahko predstavljajo JSP-ji ali pa generirani podatki iz servletov. Krmilnik vsebuje logiko razpošiljanja zahtevkov naslednjemu pogledu. Logika razpošiljanja lahko upošteva naslov URL, vhodne podatke in stanje aplikacije. Z uporabo krmilnika je logika aplikacije centralizirana, s tem pa je zmanjšana tudi sklopljenost med JSP-ji in servleti. JSP predstavlja zgolj pogled, zadolžen za predstavitev podatkov iz modela, ne vsebuje pa več logike izvajanja akcij nad modeli in logike prehajanja med stranmi. Centralni krmilnik je zadolžen za delo z modeli in izbiro naslednjega pogleda, z njegovo uporabo pa je olajšana tudi implementacija varnostne logike oziroma njeno obvladovanje. Delo krmilnika lahko strnemo v **tri točke**: interpretiranje vhoda, izvedba akcije in izbira naslednjega pogleda.

Pogledi so v Javi implementirani z JSP-ji ali neposredno s servleti. Slednji omogočajo poljubno implementacijo pogledov tako neposredno (generiranje pogleda v servisni metodi servleta) kot tudi posredno z uporabo raznih označevalnih jezikov (tipično različna narečja XML). Format podatkov, ki predstavljajo nek pogled, ni nujno samo HTML, je pa velikokrat nek besedilni format. Za take formate se večkrat uporabljajo tehnologije, ki med drugim omogočajo delo s **predlogami**. Ena glavnih zahtev, ki jo postavijo oblikovalci videza spletnih aplikacij, je gotovo točno določena zasnova videza. Ker tipično spletno aplikacijo sestavlja večje število pogledov, je vsakega od njih potrebno prirediti skupni zasnovi. Ta navadno določa standardno skupino gradnikov in njihovo postavitev (angl. layout) na pogledu. V ta namen spletna ogrodja predvidijo uporabo **predložnega stroja** (angl. template engine), ki skrbi za sestavljanje posameznih gradnikov v končni pogled. Za predloge je na primer možno uporabiti kar JSP-je. Veliko ogrodij ponuja lastne rešitve za sestavljanje in uporabo predlog. Pri implementaciji pogledov ogrodja razlikujemo med dvema modeloma arhitektur [11]; to sta **model potiska** (angl. push model) in **model povleka** (angl. pull model). V primeru uporabe prvega modela, ogrodje pred pripravo pogleda potisne primerne podatke v pogled. Uporaba modela povleka pa predvideva, da pogled samostojno povleče podatke iz modela.

Model spletne aplikacije vsebuje poslovne podatke in poslovno logiko. Ker so poslovni podatki največkrat shranjeni v zaledni podatkovni bazi, zajema najosnovnejši način implementacije modela tudi uporabo tehnologije JDBC. Ta omogoča neposredno uporabo jezika SQL, s pomočjo katerega upravljamo s podatki baze. Ker je lahko neposredna uporaba JDBC-ja nerodna in nepregledna, so bila tudi za to področje razvita primerna ogrodja. „Naravna“ ob-

lika predstavitve podatkov v Javi so objekti. Objektno-relacijsko preslikovanje (Object-relational mapping - ORM) je primer priljubljenega koncepta, ki razvijalcem omogoča neposredno delo z objekti, ki se s pomočjo primernega ogrodja brezšivno (angl. seamless) nalagajo, shranjujejo in ažurirajo v povezavi z zaledno podatkovno bazo. Gre za tako imenovano **trajnost objektov** (angl. object persistence). Pomembnejši standardni tega področja so del podlage Java EE, ki je namenjena razvoju poslovnih informacijskih rešitev. Z vidika modelov sta med bolj zanimivimi standardi gotovo JPA in EJB. Prvi je namenjen poenotenju dela z raznimi ogrodji ORM in črpa iz prakse znanih ogrodij, kot je na primer Hibernate. Drugi je starejši in večkrat posodobljeni standard, ki omogoča delo s tako imenovanimi poslovnimi javanskimi zrni (angl. enterprise java beans).

Ogrodja, ki podpirajo arhitekturo Model 2, imajo navadno vnaprej pripravljene in nastavljive krmilnike. Poleg tega predvidijo poseben razred za implementacijo akcij. Koda akcij je zadolžena za dostop do podatkovnega modela. Arhitektura model 2 je še vedno tesno vezana na vzorec zahtevek-odgovor.

Ogrodja, ki poskušajo bolje posnemati rešitve za razvoj namiznih aplikacij, temeljijo na uporabi **komponentnega modela**, kjer so komponente predstavljene z **javanskimi objekti**. Ta arhitektura ne predvideva več vloge centralnega krmilnika, ki bi po vsakem zahtevku odločal o naslednjem pogledu (spletni strani) aplikacije, temveč omogoča uporabo vzorca MVC na vsaki komponenti posebej. Navadno taka ogrodja ne temeljijo več na arhitekturi spletnih strani in prehodih med njimi, saj se približajo uporabi komponent in vsebnikov uporabniškega vmesnika, kot jih poznamo pri razvoju namiznih aplikacij. Akcije na modelih temeljijo na proženih dogodkih in njihovih poslušalcih.

Pri izibri spletnega ogrodja lahko glede na vse do sedaj napisano izbiramo med:

- različnimi **programskimi jeziki** oziroma podlagami spletnih ogrodij,
- izvajanjem aplikacije na **strežniku** ali na **odjemalcu**,
- **„klasičnim“** videzom spletne aplikacije ali **obogateni spletni aplikaciji**,
- arhitekturami **model 1**, **model 2** ali **komponentni model**,
- uporabo ogrodja s **popolnim skladom** (angl. full stack) storitev ali uporabo in povezovanjem **specializiranih ogrodij**,

- različnimi **tipi podpore** pri implementaciji modelov (JPA, EJB, ...) ali **brez podpore**.

3.2 Primera spletnih ogrodij

V nadaljevanju sta opisani dve znani spletni ogrodji. Odločil sem se za Apache Wicket [15] in GWT [21]. Razlogov za izbiro je bilo več, in sicer:

- Obe ogrodji sta dobro poznani, preizkušeni in dokumentirani.
- Razvoj aplikacij temelji na objektno usmerjenem programiranju v Javi. Obe ogrodji uporabljata komponentni model, ki je primerljiv z ogrodjem, ki ga opisujem v naslednjih poglavjih.
- Izvajalno okolje je lahko navaden vsebnik servletov (ne potrebujeta aplikacijskega strežnika).
- Ogrodji sta si dovolj različni, da pokrivata širok spekter pristopov in rešitev, ki so zanimive za primerjavo.
- Pri izbiri vsakega izmed ogrodij sem med drugim poskušal najti ogrodje, ki bi bilo na enostaven način povezljivo z lastnim ogrodjem.

3.2.1 Apache Wicket

Apache Wicket je odprtokodno spletno ogrodje napisano v Javi. Aplikacije, napisane s tem ogrodjem, se v celoti izvajajo na strežniku. Gre za ogrodje, ki omogoča razvoj aplikacij z uporabo prvorazrednih (angl. first-class) javanskih komponent uporabniškega vmesnika. Arhitektura komponent temelji na vzorcu MVC, kjer je potek izvajanja določen z dogodki in njihovimi poslušalci. Čeprav so aplikacije sestavljene iz množice spletnih strani, razvoj z Wicketom zelo spominja na razvoj namiznih aplikacij, saj ogrodje dobro poskrbi za skrivanje vzorca zahtevek-odgovor. To pomeni, da Wicket ne temelji na sicer pogosto uporabljeni arhitekturi model 2. Kljub temu še vedno uporablja „spletne pojme“ in med drugim omogoča tudi delo s spletnimi obrazci, kjer so ti predstavljeni kot vsebniki. Pomembna značilnost ogrodja so komponente z lastnim stanjem (angl. stateful component) ter modeli, sestavljeni iz običajnih javanskih objektov POJO. Wicket transparentno poskrbi za vzdrževanje stanja spletnih strani in njihove celotne hierarhije komponent in modelov. Stanje navadno vzdržuje v aktualni seji uporabnika, del identifikacije stanja pa zakodira v naslov URL. Razvoj spletnih strani in komponent poteka samo v

Javi, zato je možno uporabiti poljubno orodje oziroma urejevalnik za Javo. Za postavitev komponent na straneh in vsebnikih uporablja Wicket predloge HTML (angl. HTML template). Pri tem je posamezna predloga strogo vezana na svoj vsebnik (in obratno). Pomembna razlika glede na tipične predstavnike arhitekture model 2 je uporaba predlog, ki so hkrati vedno veljavni dokumenti HTML. To pomeni, da Wicket ne predvideva uporabe dodatnih značk (angl. tag) ali celo neposrednega mešanja kode in HTML-ja, kar je na primer značilno za JSP-je. Ogrodje vsebuje privzeto podporo tehniki AJAX in skuša v večji meri skriti in olajšati uporabo te tehnike. Razen predlog HTML Wicket ne predvideva drugih zunanjih nastavitev in s tem prepreči tako imenovani pekel XML-jev (angl. XML hell). Izraz se nanaša na veliko število datotek XML, ki jih predvidevajo določena spletna ogrodja. Ko je takih datotek preveč, postane njihovo obvladovanje oteženo, še posebej v primeru reorganiziranja (angl. refactoring) programske kode, katere delovanje je odvisno tudi od datotek XML.

Wicket je ogrodje po načelu sive škatle, saj za sestavljanje hierarhije komponent uporablja kompozicijo in dedovanje. Že osnovna različica ogrodja vsebuje množico predpripravljenih komponent, s katerimi sestavimo uporabniški vmesnik. Na voljo pa so tudi razširitve, ki vsebujejo naprednejše komponente. Wicket nudi dobro podporo lastnemu razvoju komponent ter njihovi ponovni uporabi. Ker uporaba in razvoj komponent temeljita na objektnem programiranju, je ta pristop uporabljen tudi pri delu s predlogami HTML. Wicket ne vsebuje ogrodja, ki bi omogočalo trajnost domenskih objektov (ki spadajo v model aplikacije). Za izkoriščanje trajnosti objektov dopušča možnost uporabe poljubnega namenskega ogrodja, kot je na primer Hibernate. Cilji, ki jih skušajo doseči razvijalci tega ogrodja, so predvsem [15]:

- **Enostavnost** - uporaba običajnih javanskih objektov, celotna koda je napisana v Javi in se zgleduje po znanem ogrodju Swing [22], izogibanje pogosti uporabi nastavitev XML itd.
- **Ponovna uporaba** - vse komponente so ponovno uporabne, njihovo razpečevanje je enostavno in predvideva uporabo standardnih arhivskih datotek JAR.
- **Nevsiljivost** (angl. non-intrusive) - HTML in ostali označevalni jeziki niso pomešani s programsko kodo.
- **Varnost** - koda je že privzeto varna, stanje spletnih strani lahko izpostavimo samo izrecno skozi določene naslove URL, enostavna povezava

s standardnimi javanskimi varnostnimi storitvami itd.

- **Učinkovitost in skalabilnost** (angl. scalability) - učinkovit in lahek (angl. lightweight), omogoča izvajanje v gruči (angl. cluster) itd.
- **Popolnost** - razvojna skupina obljublja izdelek z vsemi kjučnimi elementi, ki je pripravljen za takojšnjo uporabo.

3.2.2 GWT

Google je podjetje, ki stavi na splet kot podlago za storitve v povezavi z brskalnikom kot podlago za izvajanje aplikacij. GWT je Googlov nabor orodij za razvoj obogatenih spletnih aplikacij. Ta je prilagojen vzorcu, ki narekuje odjemalca s stanjem in strežnik brez stanja (angl. stateful client - stateless server). Velika prednost take delitve je skalabilnost sistema, kjer lahko povečujemo število strežnikov po potrebi brez posegov v kodo in organizacijo aplikacije na strežniku. V resnici se zasnova takega sistema ne razlikuje veliko od zasnove odjemalec-strežnik, ki se uporablja pri klasičnih namiznih aplikacijah. Velika razlika pa je v **dostopnosti aplikacij**, ki so lahko brez namestitve na voljo takoj in na različnih napravah, ki poganjajo različne operacijske sisteme. Trenutno smo priča tekmi prenosnih podlag, kot sta Applov iOS 4 in Googlov Android. Ni naključje, da avtorji omenjenih podlag posvečajo veliko pozornosti optimizaciji kode brksalnikov, še posebej pa hitrosti izvajanja JavaScripta in zgodnji podpori prihajajočemu standardu HTML 5.

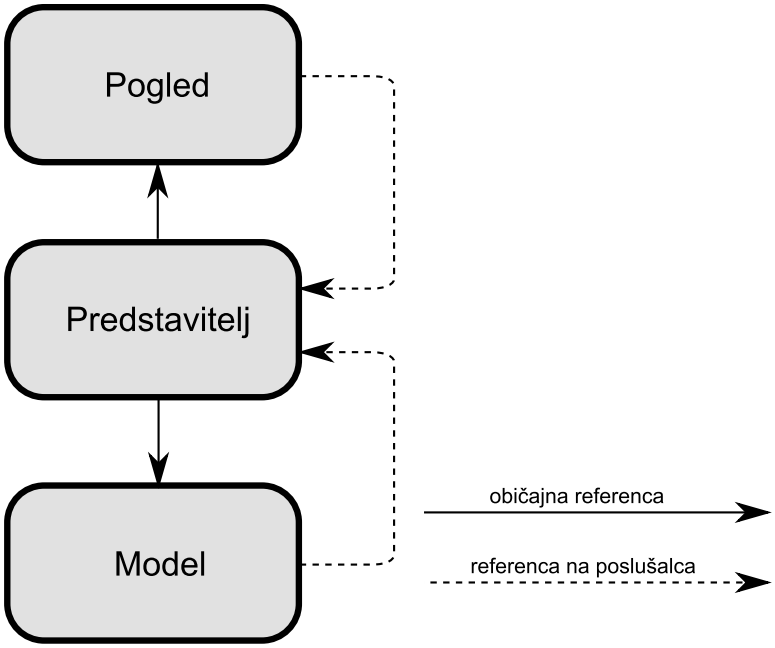
GWT omogoča razvoj obogatenih spletnih aplikacij, ki se izvajajo z JavaScriptom v **brskalniku**, razvoj aplikacij pa poteka v Javi. Lahko ga razdelimo na tri osnovne dele: prevajalnik z ostalimi razvojnimi orodji, izvajalno okolje in ogrodje za razvoj uporabniških vmesnikov. GWT-jev **prevajalnik** omogoča prevajanje javanske kode v JavaScript. Prednost uporabe Jave je v tem, da za ta jezik obstaja že veliko dobrih orodij, uporablja ga zelo veliko razvijalcev, omogoča lažje razhroščevanje itd. Prevajalnik omogoča tudi uporabo domorodne (angl. native) kode napisane v JavaScriptu, ki omogoča navezavo na obstoječe knjižnice in ogrodja napisana v tem skriptnem jeziku. Nabor **razvojnih orodij** omogoča oziroma olajša razvoj aplikacij. K tem orodjem spadata tudi vtičnik za razvojno okolje Eclipse in vtičnik za brskalnik, ki omogoča razhroščevanje in testiranje aplikacij v stvarnem času (angl. real time). **Izvajalno okolje** vsebuje lasten vmesnik za dostop do DOM-a. Ta omogoča enoličen dostop in delo z DOM-om in s tem pripomore k izdelavi kode, ki ima enak učinek v različnih brskalnikih, saj GWT poskrbi za razreševanje

morebitnih razlik v domorodnih implementacijah DOM-a. Drugi pomemben del izvajalnega okolja v GWT-ju je emulicija knjižnice, ki je del standardnega javanskega izvajalnega okolja. Brskalniki še vedno nudijo omejen nabor možnosti, če jih primerjamo z namizjem, zato GWT poskrbi za emuliranje zgolj podmnožice standardne javanske knjižnice. V največji meri so podprti predvsem razredi iz javanskih paketov `java.lang` in `java.util`.

Ogrodje GWT se za izgradnjo uporabniških vmesnikov zgleduje po sorodnih ogrodjih prirejenih za namizje. Največkrat je omenjena podobnost z ogrodjem Swing, čeprav se Google ni odločil za neposredno uporabo vzorca MVC, temveč je podprl sorodni vzorec model-pogled-predstavitelj [2] (angl. model-view-presenter - MVP). Gre za vzorec, ki v primerjavi z vzorcem MVC predvideva večjo vlogo krmilnika (preimenovanega v predstavitelja), ki popolnoma loči pogled od modela, tako da se vrine med njiju. Vzorec je prikazan na sliki 3.2. Prednosti tega vzorca sta predvsem manjša sklopljenost posameznih modulov sistema in lažje testiranje kode. Predstavitelj ima referenco na pogled in model, medtem ko ga lahko pogled in model obveščata o spremembah s proženjem dogodkov. Naloga predstavitelja je odzivanje na dogodke, izvajanje aplikativne logike in prenašanje podatkov iz pogleda v model ter obratno. Pri večjem številu različnih komponent sistema pa se lahko pokaže tudi slabost vzorca MVP. V takih primerih lahko postane obvladovanje „omrežja“ poslušalcev in dogodkov zapleteno in nepregledno.

Ogrodje GWT temelji na komponentah uporabniškega vmesnika, vsebnikih, dogodkih in poslušalcih dogodkov. Komponente so podobno kot v primeru ogrodja Wicket prvorazredni javanski objekti. Predstavljene so z razredi, ki omogočajo razširitev funkcionalnosti z uporabo dedovanja ali kompozicije. Tudi tukaj govorimo o ogrodju po načelu sive škatle. Aplikacije so v osnovi popolnoma samostojne in ne potrebujejo strežnika za izvajanje. Večina zahtevnejših aplikacij pa potrebuje dostop do strežnika. Ta je lahko v primeru GWT-ja poljuben, podpirati mora le način klica oddaljenih procedur, ki ga uporablja aplikacija. Če uporabljamo javanski strežnik (vsebnik servletov), nam GWT omogoči tudi enostaven način klicanja storitev na strežniku, kjer ogrodje poskrbi, da je implementacija strežniških storitev enostavna, klici storitev pa zakrijejo kompleksnost pogovora med odjemalcem in strežnikom.

V osnovi predvideva GWT **programsko** deklariranje in postavitev komponent na vsebnikih. Zadnje različice poleg tega omogočajo tudi uporabo XML predlog, ki omogočajo **deklarativni** pristop k dodajanju komponent na vsebnike.



Slika 3.2: Vzorec MVP

Za ta pristop se je Google odločil, da bi olajšal delo oblikovalcem, ki jim je deklarativni pristop bolj domač. Pomembna značilnost GWT-ja je tudi dobra podpora ponovni uporabi kode, saj je na novo nastale komponente in pripadajoče datoteke enostavno združiti v tako imenovane module, ki jih je možno enostavno uporabiti v drugih projektih. Tudi zato je na spletu na voljo veliko število modulov, s katerimi lahko nadgradimo osnovne zmožnosti ogrodja GWT. Pri snovanju ogrodja je Google mislil tako na razvijalce kot na končne uporabnike. Prvi lahko razvijajo v znanem okolju, ki do določene mere zakrije kompleksnost spletnih tehnologij, drugi pa lahko uporabljajo aplikacije, ki so takoj dostopne in odzivne z bogatim naborom funkcionalnosti.

3.3 Odločitev za lastno ogrodje

Prve spletne aplikacije smo razvijali s tehnologijo ASP. Arhitektura aplikacij je bila večinoma model 1, ki smo jo do neke mere nadgradili z uporabo zunanjih komponent. Nadgrajevanje in vzdrževanje spletnih aplikacij v ASP-ju je postopoma postajalo vse bolj zahtevno, izvorna koda dinamičnih strani pa

nepregledna. Zaradi omenjenega smo v podjetju začeli iskati novo okolje, v katerem bi lahko razvijali za splet. Odločili smo se za Javo in spletni strežnik, ki je hkrati vsebnik servletov. V času spoznavanja novega okolja smo vanj prenesli tudi nekaj starejših spletnih aplikacij. Končni cilj je bil poenostaviti razvoj do te mere, da bi bil ta primerljiv z razvojem namiznih aplikacij. Za zgled smo vzeli znani ogrodji AWT in Swing. Želeli smo vzpostaviti tudi širše zasnovano okolje, ki bi nudilo dodatne storitve, kot so enostaven dostop do podatkovnih baz z uporabo tehnologije JDBC ali drugih namenskih ogrodij, enostavna namestitev aplikacij, uporabniški računi in dovoljenja ipd.

Skozi večletni razvoj v podjetju smo se opirali na različne odprtokodne rešitve, kot so strežniški operacijski sistem, spletni strežnik, programske knjižnice, baza podatkov, strežnik za GIS storitve in ne nazadnje tudi razvojno okolje (angl. integrated development environment - IDE). Pri iskanju ogrodja za izgradnjo uporabniških vmesnikov smo naleteli na omejitve pri izbiri odprtokodnih rešitev. Koda aplikacij je navadno tesno povezana s takim ogrodjem, zato je lahko izbira le-tega tvegano opravilo. Predvsem v primeru, ko želimo s takim ogrodjem na dolgi rok razviti večje število aplikacij. V času razvoja prvega spletnega ogrodja na žalost ni bilo tako velike izbire ogrodij, ki bi omogočala razvoj aplikacij, primerljiv s tistimi za klasično namizje. Še najbolj se je našim željam približalo ogrodje Echo 2, ki se zgleduje po Swingu, a ni bilo toliko uveljavljeno, prav tako pa nam niso bile všeč določene rešitve glede implementacije ogrodja. Šlo je predvsem za način vzdrževanja stanja aplikacij in na prvi pogled težji razvoj lastnih komponent. V času, ko je med spletnimi ogrodji večinoma „kraljeval“ model 2, smo se zato odločili za lasten razvoj spletnega ogrodja po vzorcu MVC na podlagi prvorazrednih javanskih komponent, modelov, dogodkov in poslušalcev. Del aktualnega ogrodja, ki ga opisujem v nadaljevanju diplomske naloge, temelji na opisani implementaciji (starejšega) ogrodja in predstavlja logično nadaljevanje le-te. V času, ko je število sorodnih odprtokodnih ogrodij vedno večje, si postavljamo predvsem vprašanje, ali bi bilo povezovanje takih ogrodij z lastno rešitvijo koristno in kako bi tako povezavo lahko izvedli. Tudi na to vprašanje sem poskušal odgovoriti v nadaljevanju diplomske naloge.

3.3.1 Splošne zahteve

Pred snovanjem ogrodja smo postavili nekaj splošnih zahtev:

- Lastno ogrodje naj bo modularno in dovolj prilagodljivo, da ga je mogoče povezati z drugimi ogrodji. Posamezni moduli naj bodo zamenljivi.

- Aplikacije naj bodo podobne klasičnim namiznim aplikacijam. Podobnost s klasičnimi aplikacijami naj velja tudi za njihov razvoj (MVC, Swing) in namestitev na strežnik.
- Razvijalci naj se v čim manjši meri srečujejo s spletnimi tehnologijami kot so HTML, CSS, JavaScript in AJAX.
- Ogrodje naj nudi podporo različnim vlogam sodelujočih na projektu, kot so: razvijalec / programer, odgovoren za dostop do podatkovne baze (optimizacija poizvedb SQL), oblikovalec videza, odgovoren za nameščanje aplikacij in odgovoren za nemoteno delo sistema v produkciji.
- Vpeljevanje novih ogrodij oziroma knjižnic v uporabo naj bo enostavno. Ogrodje naj v največji možni meri zakrije zapletenost novih vmesnikov oziroma naj jih **nadomesti** z lastnimi, ki so po možnosti enostavnejši.
- Ogrodje naj bo dovolj „stabilno“, da skozi nadgradnje **vzdržuje združljivost** s prejšnjimi različicami. Prilagaja naj se potrebam, ki se pojavijo na konkretnih projektih.

Pričakovane prednosti

Prva pričakovana prednost lastnega ogrodja je bila zagotovo dobra podpora razvoju spletnih aplikacij, ki bi dolgoročno omogočila krajše razvojne cikle in kvalitetnejšo kodo. Pričakovali smo tudi, da bo vpeljevanje novih tehnologij in drugih ogrodij lažje in bolj razumljivo, če bo kontrola nad njimi centralizirana. Predvideli smo, da bomo s popolnim obladovanjem komponent uporabniškega vmesnika lahko razvili uporabnikom bolj prijazne aplikacije. Obetali smo si tudi boljšo organizacijo dela in podporo notranjim standardom dela. Določene standarde smo želeli skupaj z razvojem ogrodja tudi prilagoditi novejšim trendom, ki so na primer predvidevali poenostavljanje uporabniških vmesnikov.

Predvidene slabosti

Prva slabost lastnega razvoja, o kateri zagotovo ni bilo potrebno preveč razmišljati, je bila cena razvoja lastnega ogrodja. Druga pa, kako doseči postopen razvoj, ki bo vzporedno sledil potrebam projektov in pri tem ne bo uveljavljal preveč „kratkovidnih“ rešitev. Ko smo določili tip spletnih aplikacij, ki jih bo podpiralo ogrodje, smo se zavestno omejili na predvidene potrebe podjetja. Pri tem je bilo vseeno možno, da bi bodoči projekti zahtevali tudi razvoj „bolj klasičnih“ spletnih aplikacij. Ogrodje, ki smo ga pripravljali,

je moralo biti zato dovolj razslojeno, da bi ga lahko v največji možni meri uporabili tudi v drugih primerih oziroma ga po potrebi povezali z drugimi ogrodji. Skrajni primer bi bila seveda tudi uporaba popolnoma neodvisnih rešitev. Če bi bilo takih primerov več, bi bila odločitev za lastno ogrodje povsem napačna. Ena od slabosti, ki smo se jih zavedali, je bilo potrebno znanje in uvajanje kadrov. Pri uporabi lastnega ogrodja je seveda nemogoče iskati kadre, ki bi ogrodje že poznali, kot je to mogoče pri uporabi znanih in uveljavljenih rešitev. Ker naj bi se ogrodje zgledovalo po znanih rešitvah (Swing), je bilo poznavanje teh rešitev najboljši približek, ki smo ga lahko zahtevali od novih kadrov (kasneje se je izkazalo, da je to dovolj dobra rešitev).

Poglavje 4

Razvoj lastnega spletnega ogrodja

Kot sem že omenil, smo se odločili za razvoj v **Javi**. Glavni razlog za tako izbiro je bilo gotovo predhodno poznavanje tega programskega jezika in njegova široka uprabnost. Drugi razlog je bilo produkcijsko okolje, ki temelji na strežnikih z nameščenim operacijskim sistemom **GNU/Linux**. Prenosljivost javanskih programov med različnimi operacijskimi sistemi je bila v tem primeru ključna, saj v večji meri še vedno razvijamo v Ms. Windows.

4.1 Izbira razvojnega okolja

Za potrebe razvoja smo izbrali razvojno okolje **Eclipse** in takoj izkoristili vgrajeno podporo sistemu za **kontrolo hkratnih različic** (angl. Concurrent Versions System - CVS). Kasneje smo razvojno okolje razširili še z zelo uporabnima vtičnikoma **Mylyn** in **FindBugs**. Prvega smo povezali s spletnim sistemom za upravljanje razvoja **Bugzilla** in tako dobili dobro rešitev za upravljanje z opravili (angl. task) in sledenje napakam. Drugega smo izbrali zaradi opozarjanja na skrite napake in kot opomnik v primerih uporabe „sumljivih“ vzorcev programske kode. V primeru vsebnika servletov smo iskali samostojen strežnik, ki bi bil hiter in varčen s porabo pomnilnika. Izbrali smo strežnik **Jetty**, ki smo ga dodatno „okrnili“ tako, da smo odstranili vse module, ki jih nismo potrebovali (med njimi je bila tudi podpora JSP-jem). Nazadnje smo v Eclipse dodali še vtičnik **„Run Jetty Run“**, ki omogoča hitro izvajanje strežnika Jetty neposredno iz razvojnega okolja.

4.2 Glavni moduli

Ogrodje, ki ga opisujem v nadaljevanju, sestavljaja sistem treh šibko sklopljenih modulov: Jedro ogrodja 3iOS, predložni stroj in kompnentno ogrodje za uporabniške vmesnike 3iGUI. **3iOS** je podlaga, ki nudi podporo ostalim modulom. **Predložni stroj** je namenjen sestavljanju dinamičnih spletnih strani s pomočjo predlog. **3iGUI** je najpomembnejši in tudi največji modul celotnega ogrodja. Gre za samostojno ogrodje, ki omogoča razvoj spletnih aplikacij, ki temeljijo na komponentnem modelu in dogodkih. Zgleduje se po javanskih ogrodjih AWT in Swing in je glavni modul, s katerim imajo opravka razvijalci.

Vsi moduli se izvajajo v standardnem vsebniku servletov. Del implementacije modulov zato sestavljajo tudi strežniški filtri in servleti. Razvijalcem teh fitrov in servletov ni potrebno poznati. Prav tako je za nameščanje v vsebnik predvidena v naprej pripravljena struktura map, ki vsebujejo standardno in že pripravljeno opisno datoteko (angl. deployement descriptor) **web.xml**. 3iOS se namesti v vsebnik kot **ena spletna aplikacija**, kot jo določa standard Java EE. S pomočjo modulov jedra pa 3iOS tipično poganja večje število „svojih“ spletnih aplikacij (v nadaljevanju aplikacije).

4.3 Jedro ogrodja 3iOS

3iOS omogoča izvajanje aplikacij, servletov, cikličnih opravil in spletnih storitev. Prve so že omenjene aplikacije razvite na podlagi ogrodja 3iGUI. **Servleti** so določeni s standardom Java EE, 3iOS pa jim dodatno omogoči še dostop do lastnih storitev, zaščito dostopa z lastnim sistemom avtorizacije in lastna pravila pri preslikovanju URL-jev v klice servletov. **Ciklična opravila** predstavljajo objekti, ki jih 3iOS izvaja ciklično v določenih časovnih intervalih. Uporabna so za obdelovanje sporočil v raznih čakalnih vrstah, občasne klice določenih zunanjih spletnih storitev, arhiviranje, čiščenje začasnih map, sporočanje o trenutnem stanju sistema itd. **Spletne storitve** v 3iOS-u so objekti, ki jih lahko prek protokola HTTP pokličemo s parametri po metodah GET ali POST. Na klic lahko odgovorijo z vsebino poljubnega formata. Večinoma uporabljamo različne besedilne formate, kot so XML, CSV in „multipart/mixed“. V določenih primerih uporabljamo tudi binarne formate (na primer storitev, ki generira slikovno datoteko).

Med 3iOS-ove glavne naloge spadajo:

- instanciranje aplikacij in drugih pomembnih objektov v sistemu,
- **povezovanje in ograjevanje** drugih ogrodij in knjižnic,
- upravljanje z viri podatkov, opozarjanje na nepravilno rabo virov in omogočanje skupnega in relativno enostavnega pogleda na množico virov,
- upravljanje z uporabniškimi računi in implementacija varnostnih mehanizmov,
- diagnostika delovanja aplikacij in sprotno obveščanje o morebitnih napakah aplikacij v produkciji,
- izvajanje različnih sistemskih storitev, ki so nujno potrebne za nemoteno izvajanje aplikacij (na primer upravljanje s stanji spletnih aplikacij in sproščanje pomnilnika),
- upravljanje sistema z uporabo sistemskih aplikacij.

3iOS temelji na množici javanskih vmesnikov, ki aplikativni kodi omogočajo dostop do jedra in njegovih modulov, ko potrebuje določene storitve. Predvideno je, da se v sklopu ene instance 3iOS-a izvaja večje število aplikacij in drugih storitev. 3iOS poskuša razvijalcem prikriti tipično spletno okolje in ga nadomestiti z okoljem, ki spominja na namizje. Tukaj ne gre samo za podobnost pri razvoju uporabniških vmesnikov, temveč tudi za podobnost pri organizaciji datotek in map. Poleg izvajalne kode sestavlja vsako aplikacijo tudi določeno število datotek. Te datoteke se tipično nahajajo v mapi, v katero smo namestili aplikacijo. Mednje spadata vsaj namestitvena in izvajalna datoteka, navadno pa se jima pridružijo še razne nastavitvene datoteke. Aplikacije lahko dostopajo do datotek in map ter jih v stvarnem času tudi spreminjajo. Tipični predstavniki datotek, ki se spreminjajo v stvarnem času, so na primer dnevniške datoteke (angl. log file). Dostop do map in datotek je omogočen prek modula 3iFS, do katerega lahko dostopamo prek jedra. Jedro in modul 3iFS poskušata z organizacijo in imenovanjem razredov in datotek „pričarati“ okolje, ki bi spominjalo na klasičen operacijski sistem. Čeprav večino klicev vmesnika API preusmerita nižje ležečim ogrodjem in knjižnicam, pa njuna vloga ni samo ograjevanje drugih storitev, temveč v določenih primerih prevzameta bolj aktivno vlogo z grajenjem objektov in neprestanim vzporednim izvajanjem določenih opravil.

4.3.1 Arhitektura kode jedra

Pri snovanju arhitekture jedra sem želel doseči šibko sklopljenost med aplikativno kodo in „okoliškimi“ storitvami (knjižnicami, ogrodji, spletnimi storitvami ipd.). Cilja, ki sem ju želel doseči prek šibke sklopljenosti, sta bila predvsem prenosljivost ogrodja med različnimi okolji in standardizacija dostopa do večine storitev, ki jih potrebuje aplikativna koda. Če na primer razvijamo servlet, bomo za informacijo o trenutno prijavljenem uporabniku morali uporabiti popolnoma drug objekt, kot če razvijamo namizno aplikacijo. Podobno moramo v namizni aplikaciji, ki dostopa do datotek, predvideti različna pravila imenovanja poti, če bo aplikacija tekla na različnih operacijskih sistemih. Takih primerov je še več in pri snovanju ogrodja sem jih reševal z lastnim naborom vmesnikov, za katerimi se lahko nahaja poljubna implementacija storitev. Ta je lahko prirejena namizju, spletu, različnim virom podatkov in drugim storitvam. Zaradi širokega spektra storitev, ki naj bi jih nudilo ogrodje, sem moral le-te zakriti za smiselno imenovanimi vmesniki in določiti osrednjo točko dostopa do njihovih implementacij. Ta osrednja točka dostopa, naj bi bila vedno enaka za vse aplikacije ne glede na okolje, za katero bi bile razvite.

Osnova 3iOS-a je 19 javnih vmesnikov, ki jih implementirajo jedro in njegovi moduli. Najpomembnejši med njimi so:

- `IOperatingSystem` - glavni vmesnik, ki ga implementira jedro. Ta vmesnik omogoča dostop do vseh javno dostopnih modulov jedra.
- `IFileSystem` - omogoča delo z datotekami in mapami.
- `IObjectFileSystem` - omogoča preslikavo med datotekami in objekti.
- `IRegistry` - omogoča dostop in spreminjanje sistemskih nastavitev.
- `IApplicationManager` in `IApplicationStateManager` - omogočata dostop do namestitvenih podatkov o aplikacijah in upravljanje s stanji aplikacij.
- `IAuthSession` - omogoča prijavljanje v sejo in odjavljanje iz seje, vzdrževanje objektov na seji in pridobivanje podatkov o odjemalcu. Omogoča tudi dostop do trenutno prijavljenega uporabnika, ki je določen z vmesnikom `IUser`.
- `ILogger` - omogoča dodajanje zapisov v dnevniške datoteke.

Aplikacije, spletne storitve, servleti in ciklična opravila imajo dostop do jedra in njegovih modulov samo prek uporabe omenjenih vmesnikov. Aplikativna koda se ne navezuje na nobeno izmed implementacij (poseben primer so sistemske aplikacije, ki se lahko neposredno vežejo na implementacijo jedra in modulov). Implementacijo jedra je potrebno določiti in nastaviti šele v izvajalnem okolju na strežniku. Omenjeni vmesniki in njihova implementacija izkoriščajo vzorec fasade (angl. facade pattern) [13].

Jedro je vsebnik množice specializiranih modulov. Ti se lahko glede na vrsto jedra razlikujejo. Najpomembnejši se pojavljajo v vseh različicah jedra in so:

- **3iFS** - omogoča dostop in delo z datotekami in mapami.
- **3oFS** - nadgradi 3iFS z možnostjo neposrednega preslikovanja med datotekami in objekti.
- **Application_manager** - skrbi za nastavitve vseh nameščenih aplikacij.
- **Application_state_manager** - modul, ki je zadolžen za upravljanje s stanji aplikacij.
- **Auth_session** - predstavlja aktualno sejo. Omogoča dostop do objektov, ki so shranjeni v seji, dostop do informacij o aktualnem uporabniku, njegovi IP številki, spletnem digitalnem potrdilu itd. Gre za osrednji modul, ki omogoča izvajanje varnostnih mehanizmov v stvarnem času.
- **Authorization_manager** - skrbi za vloge (angl. role) uporabnikov v sistemu.
- **Db_connection_pool** - vsebuje bazen povezav na poljubno množico podatkovnih baz.
- **Group_manager** - skrbi za nalaganje definicij skupin uporabnikov in preverja njihovo pravilnost.
- **Kernel_logger** - osnovni modul, ki omogoča pisanje sistemskih dnevniških datotek.
- **Log_manager** - omogoča pisanje dnevniških datotek. Namenjen je modulom jedra in aplikativni kodi za dodajanje novih zapisov v dnevniške datoteke. Modul temelji na uporabi vzorca proizvajalec-porabnik (angl. producer-consumer pattern) [16].

- **Realm_manager** - skrbi za nalaganje definicij področij uporabnikov in preverja njihovo pravilnost.
- **Registry** - omogoča dostop in spreminjanje osnovnih nastavitev sistema.
- **Service_manager** - je vsebnik za ciklična opravila. Izvaja se v svoji niti, ki vsebuje enostaven razporejevalnik opravil (angl. task scheduler).
- **User_manager** - skrbi za nalaganje definicij uporabnikov, pripravo njihovih vlog ter razreševanje odvisnosti med skupinami, področji in uporabniki. Poleg tega skrbi za osveževanje pravic uporabnikov (ob vsaki prijavi) ter preverjanje uporabniških računov.
- **Web_service_manager** - skrbi za namestitve spletnih storitev, njihovo instanciranje, izvajanje in obveščanje o morebitnih napakah.

4.3.2 Priprava in dostop do jedra

Jedro ni strogo vezano na spletno okolje. Vmesniki ne zahtevajo izvajanja v spletnem okolju temveč jih je možno implementirati na tak način, da jedro lahko uporabimo tudi pri razvoju klasičnih namiznih aplikacij. Pri uporabi v spletnem okolju jedro potrebuje **prilagoditev vzorcu zahtevek-odgovor**, hkrati pa mora omogočiti **svojo dostopnost aplikativni kodi**. V času uporabe ogrodja 3iOS na spletu smo razvili več implementacij jedra, ki pa temeljijo na skupnem vzorcu uporabe. V sistemu je vedno ena sama instanca jedra, ki jo na strežniku pripravi in vsebuje filter `IOSFilter`. Ob zagonu poda filter jedru **pot do korenske mape**, iz katere se nato zažene modul 3iFS. Če je omenjena pot pravilna, se v jedru v točno določenem vrstnem redu zaženejo vsi preostali moduli. Jedro sporoča vse informacije o lastnem stanju prek sistemskih dnevniških datotek.

Kot pove že njegovo ime, je `IOSFilter` zadolžen za filtriranje zahtevkov. Filter je postavljen pred vse sistemske servlete, ki skrbijo za razpečevanje zahtevkov aplikativni kodi. Pri tem filter poskrbi, da aplikativna koda ob vsakem zahtevku dobi dostop do jedra. Ker se lahko zahtevki **prožijo in obravnavajo vzporedno**, je potrebno poskrbeti, da vsaka nit, na katero je v določenem trenutku vezan nek zahtevek, omogoči pravilen dostop do jedra. Filter med drugim poskrbi tudi za vpeljavo konteksta. Na strežniku je lahko v določenem trenutku prijavljenih večje število uporabnikov, vsak z lastno sejo. Filter zato poskrbi, da se vsakemu zahtevku priredi različica jedra, ki ustreza kontekstu zahtevka. Za to filter uporablja vzorec dekoratorja (angl. decorator

pattern) [14], ki **ogradi jedro** z objektom, ki je tudi implementacija vmesnika `IOperatingSystem`. Dekorator je „objavljen“ na tak način, da lahko do njega neposredno dostopa vsa aplikativna koda. V ta namen se uporablja razred `OS`, ki vsebuje statično metodo `getCurrent()`. Ta metoda temelji na uporabi objekta tipa `ThreadLocal`, ki omogoča varno uporabo „globalnih“ spremenljivk v večnitnem okolju. Primer dostopa do jedra ogrodja, je prikazan v sledečem izseku kode:

```
IOperatingSystem os = OS.getCurrent();
```

Ko je dekorator pripravljen in objavljen, filter spusti zahtevek v nadaljnjo obdelavo. Za filtrom so na vrsti sistemski servleti. Teh je več in pokrivajo vse možne zahtevke, ki jih predvideva sistem. Naloga servletov je dodatna priprava vsebine konteksta, grajenje instanc objektov in posredovanje zahtevkov v nadaljnjo obdelavo.

4.3.3 Različne implementacije jedra

Ker je arhitektura jedra pogojena samo z nekaj javnimi vmesniki (in njihovo dokumentacijo), je možnih implementacij veliko. Trenutno imamo implementirane tri glavne različice jedra, ki imajo lahko dodatne podrazličice. Osnovna različica je namenjena zgolj potrebam razvoja. Vsebuje implementacijo vseh potrebnih modulov. **Razvojno jedro** ne omogoča upravljanja uporabniških računov in privzeto vsebuje samo tri nespremenljive uporabnike, ki so namenjeni testiranju. Razvojno jedro ni namenjeno samo razvijalcem aplikacij temveč tudi razvijalcem novih jeder, ki želijo izkoristiti že napisane module. Jedro gradi svoje module z uporabo tovarniških metod (angl. factory method) in omogoča njihovo prekrivanje v podrazredih. Tako lahko podrazličice jedra na novo vpeljejo samo specifične module, ostale pa podedujejo. Modula, ki ju največkrat zamenjajo podrazredi, sta na primer `authorization_manager` in `user_manager`.

Produkcijsko jedro, ki ga uporabljamo v večini namestitev, ima ločen sistem upravljanja z uporabniki in pravicami. Gre za spletno storitev, ki se izvaja v ločenem spletnem strežniku. Razvita in uporabljena je bila pred ogrodjem 3iOS za potrebe starejših spletnih aplikacij. Ker se je rešitev izkazala za učinkovito, smo na njeni podlagi razvili novo različico modula `user_manager`, ki je postal del produkcijskega jedra.

Tretje jedro, ki ga uporabljamo, je vezano na elektronsko poslovanje občin. Izpeljano je iz razvojnega jedra, uporabniške račune pa povezuje z zalednim sistemom elektronskega poslovanja. Modul `user_manager` je v tem primeru povezan s sistemom tako imenovanih e-strank, ki se v sistem prijavljajo s spletnimi digitalnimi potrdili.

Bistvo vseh implementacij jedra je možnost uporabe že razvitih in preizkušenih rešitev. Tako je omogočeno recikliranje kode in povezovanje različnih storitev, ki jih 3iOS „skrije“ za svojimi vmesniki.

4.3.4 Vmesnik modula 3iFS in organizacija map

Z implementacijo modulov vmesnika 3iFS sem ogradil objekte, ki smo jih uporabljali že v starejših ogrodjih in predstavljajo dodaten vmesni sloj med aplikativno kodo in viri podatkov. Želel sem doseči enostaven dostop do podatkov prek sistema, ki razvršča vire po „navideznih“ mapah in datotekah in lahko poskrbi za dodatne akcije nad podatki, če je to potrebno. Cilj uporabe takih „navideznih“ virov je bil poenotenje dostopov do različnih datotečnih sistemov, dokumentov v podatkovnih bazah, spletnih storitev, drugih „oddaljenih“ virov, čakalnih vrst in še česa. Poleg tega sem s prevzemom kontrole nad dostopom do podatkov želel omogočiti boljšo diagnostiko in morebitno posredovanje sistema pri nepravilnem delu z viri (predvsem v zvezi s pravilnim zaključevanjem objektov, ki se povezujejo z viri).

Modul 3iFS spada med pomembnejše module jedra ogrodja 3iOS. Omogoča dostop do datotek in map sistema. Za dostop uporablja sistem imenovanja, ki spominja na imenovanja v znanih operacijskih sistemih. Datoteke so razporejene po mapah, ki so hierarhično povezane v drevesno strukturo. Aplikativna koda lahko dostopa do storitev tega modula prek vmesnika `IFileSystem`. Glavni metodi, ki ju določa ta vmesnik, sta:

- `IWritableResource getResource(String absolutePath)` - omogoča dostop do datotek in map. Če datoteka ali mapa ni zaščitena, je moč vanjo tudi pisati. Vmesnik `IWritableResource` predstavlja mapo ali pa datoteko, ki je vir ali ponor podatkovnega toka.

- `String resolveAbsolutePath(String path)` - omogoča preslikovanje bližnjic v absolutne poti.

Pot vsake datoteke sestavljajo štirje gradniki. Prikazal jih bom na primeru datoteke, ki se nahaja v javno dostopni mapi, in sicer `/[INETPUB]/test/primer.html?p1=test`. Gradniki poti so sledeči:

- `/[INETPUB]/` - koren poti. Vsaka pot v 3iFS-u se vedno začne s korenom. 3iFS loči **tri tipe korenov**: statični, navidezni in dinamični. **Statični** koreni so neposredno vezani na nek zaledni vir podatkov. **Navidezni** koreni so bližnjice do poljubnih map. **Dinamični** koreni so koreni, ki jih pripravijo sistemski servleti in so vezani na trenutni kontekst aplikacije. Primeri takih korenov so:
  - `/[CURRENT_HOME]/` - mapa, v kateri se izvaja trenutna aplikacija, servlet, ciklično opravilo ali spletna storitev,
  - `/[CURRENT_USER]/` - domača mapa trenutno prijavljenega uporabnika,
  - `/[CURRENT_REALM]/` - domača mapa področja, v katero spada trenutno prijavljeni uporabnik.
- `test/` - pot do mape. Vsa imena map se v 3iFS-u obvezno končajo z znakom „/“.
- `primer.html` - ime datoteke.
- `?p1=test` - parametri imena datoteke. Znak „?“ določa začetek parametrov v imenu datoteke. Če je v imenu več parametrov, jih lahko ločimo z znakom „&“. Format parametrov je zelo podoben formatu metode GET protokola HTTP. Parametri v imenu datotek se lahko uporabljajo za pošiljanje sporočil nižje ležečim slojem kode, kot so na primer filtri v modulu 3iFS.

Modul 3iFS nudi storitve aplikativni kodi in večini modulov sistema. Ti ga večinoma uporabljajo za dostop do svojih nastavitev, nekateri pa vanj tudi pišejo (dnevniške datoteke, ažuriranje nastavitev, shranevanje stanj aplikacij itd.). Z vidika implementacije spada 3iFS med **bolj zahtevne** module. Število klicev vmesnika je lahko v enem zahtevku HTTP veliko. Poleg tega so klici metod konkurenčni, saj lahko do vmesnika v nekem trenutku dostopa več niti. Po drugi strani so dostopi do virov podatkov, s katerimi dela 3iFS, tipično počasni. V večini primerov gre za dostope do datotečnega sistema na strežniku, izvor datotek pa je lahko tudi podatkovna baza ali spletna storitev. Zaradi tega ima modul 3iFS razslojeno arhitekturo, ki med klicatelja in vir podatkov vrine

še **sloj filtrov**. Gre za objekte, ki lahko prestrežejo zahtevke po datotekah oziroma s pomočjo dekoratorja ogradijo objekte `IWritableResource`, ki jih vrača nižje ležeči sloj. Pri tem so filtri postavljeni v zaporedno verigo, kjer eden za drugim filtrirajo podatke. Filtri lahko omogočajo spreminjanje vsebine prebranih datotek v stvarnem času, predpomnjenje pogosto dostopanih datotek, beleženje dostopov, zaščito pred branjem ali pisanjem, sporočanje, diagnostiko itd. Določeni filtri se odzivajo tudi na parametre, ki so pripeti imenu datoteke. Tako je na primer možno izklopiti vse filtre z uporabo parametra `raw=true`. Filtri se odzivajo samo na določene poti datotek. V nastavitvah modula 3iFS je za to predvidena uporaba **regularnih izrazov** (angl. regular expression).

Koreni in mape v modulu 3iFS so poljubno nastavljivi. Da bi bila ogranizacija in imenovanje map čim bolj razumljiva, smo določene mape preimenovali po vzoru znanih operacijskih sistemov. Osnovna struktura map, ki jo uporabljamo, je zato:

- `/[HOME]/` - koren vseh domačih map.
- `/[INETPUB]/` - javno dostopna mapa, do katere lahko dostopajo vsi uporabniki (tudi neprijavljeni).
- `/[NATIVE]/` - koren, ki vsebuje „navidezne“ mape, ki se preslikajo na poljubno mesto izven predvidene korenske mape za 3iFS.
- `/[PROGRAM_FILES]/` - koren, ki vsebuje namestitve vseh 3iOS-ovih aplikacij.
- `/[SERVICES]/` - koren, ki vsebuje namestitve vseh cikličnih opravil.
- `/[SYSTEM]/` - koren, ki vsebuje večino sistemskih nastavitev.
- `/[TEMP]/` - koren, ki vsebuje začasne datoteke.
- `/[WEB_SERVICES]/` - koren, ki vsebuje namestitve vseh spletnih storitev.

V nadaljevanju sem podal še vzorec kode, ki prikazuje pravilno uporabo objektov pri branju datoteke. Prikazana je priprava vhodnega toka iz besedilne datoteke, ki se nahaja v domači mapi trenutno prijavljenega uporabnika. Gre za enega od vzorcev uporabe ogrodja 3iOS, ki sem jih opisal v sklopu priprave dokumentacije ogrodja.

```
IFileSystem fs = OS.getCurrent().getFileSystem();
IWritableResource myRes = fs.getResource(
  "/[CURRENT_USER]/application_data/test/test.txt"
);

// preberemo vsebino datoteke, če obstaja
if (myRes.exists()) {
  InputStream in = null;
  try {
    in = myRes.getInputStream();
    // tukaj preberemo vsebino in z njo nekaj naredimo
    // ...
  }
  finally {
    if (in!=null) try {in.close();} catch (Exception ex) {}
  }
}
```

4.3.5 Avtentikacija in avtorizacija

Pri snovanju vmesnika za avtentikacijo in avtorizacijo sem želel doseči enostavnost pri delu z uporabniškimi računi, s pravicami dostopov, z vlogami uporabnikov, pri dostopu do informacij o aktualnem uporabniku itd. Predvsem pa sem poskušal doseči, da bi bilo točk, v katerih aplikacija potrebuje podrobne podatke o aktualnem uporabniku, čim manj. Ogrodje 3iOS že „ve“, kdo je aktualni uporabnik, zato sem poskušal doseči, da bi to ogrodje v večji meri samo poskrbelo za opravila, ki so vezana na varnost.

Dostop do aplikacij, servletov, spletnih storitev in zaščitenih podatkov se izvaja v povezavi z modulom auth_session. Vsi uporabniki so v sistemu predstavljeni z implementacijami vmesnika `IUser`. Ta prek svojih metod omogoča dostop do osnovnih podatkov uporabnika, kot so uporabniško ime, področje, tip uporabnika, vloge uporabnika v sistemu itd. Auth_session poskrbi za pridobivanje objektov `IUser` v sodelovanju z drugimi moduli.

3iOS omogoča uporabo deklarativne in programske varnosti. **Deklarativna varnost** omogoča nastavljanje pravic dostopov izven programske kode. Pravice dostopov nastavimo v namestitvenih datotekah aplikacij in spletnih

storitev. Te datoteke med drugim vsebujejo seznam izvajalnih[3] datotek z vlogami, ki omogočajo njihovo izvajanje. Spodaj je prikazan primer namestitvene datoteke aplikacije. Ta se navadno imenuje `index.app.xml`.

```
<application id="LOGIN">

  <info>
    <title>Prijava v sistem</title>
  </info>

  <authorization>
    <declare_resources>
      <application_resource role="SYS_inetpub">
        index.layout.xml
      </application_resource>
      <servlet_resource role="SYS_inetpub">
        logout.srv.xml
      </servlet_resource>
    </declare_resources>
  </authorization>

</application>
```

Prikazana je uporaba vloge `SYS_inetpub`, ki se privzeto dodeli vsem uporabnikom in omogoča dostop do javnih aplikacij in podatkov. Namestitvena datoteka vsebuje deklaraciji dveh izvajalnih datotek, ki glede na uporabo omenjene vloge omogočata izvajanje vsem uporabnikom. Podoben pristop deklarativne varnosti se uporablja tudi v primeru namestitvenih datotek spletnih storitev.

Programska varnost se uporablja takrat, ko samo deklarativna varnost ne zadostuje. Vključena je v kodo aplikacij in večinoma temelji na uporabi metod vmesnika `IUser`. Te so predvsem:

- `isAnonymous()` - vrne informacijo, ali je uporabnik prijavljen.
- `isSystem()` - vrne informacijo, ali je uporabnik sistemski uporabnik z vsemi pravicami.

[3]Gre za nastavitvene datoteke XML, s pomočjo katerih lahko ogrodje prebere ime razreda, pripravi njegovo instanco in jo s klicem metod uporabi.

- `isInRole(String role)` - vrne informacijo, ali ima uporabnik določeno vlogo. Zgoraj omenjeni sistemski uporabnik ima vse vloge, zato v takem primeru ta metoda vedno vrne pozitiven odgovor.

Kot je razvidno iz do sedaj opisanega, pri avtentikaciji in avtorizaciji uporabnikov sodeluje več modulov skupaj z aplikativno kodo. Deklarativna varnost omogoča neodvisno nastavljanje pravic dostopa, ki ob času razvoja niso nujno znane. Programska varnost pa razvijalcem omogoča bolj podrobno obvladovanje dostopov do kritičnih delov kode.

4.3.6 Sistemske aplikacije

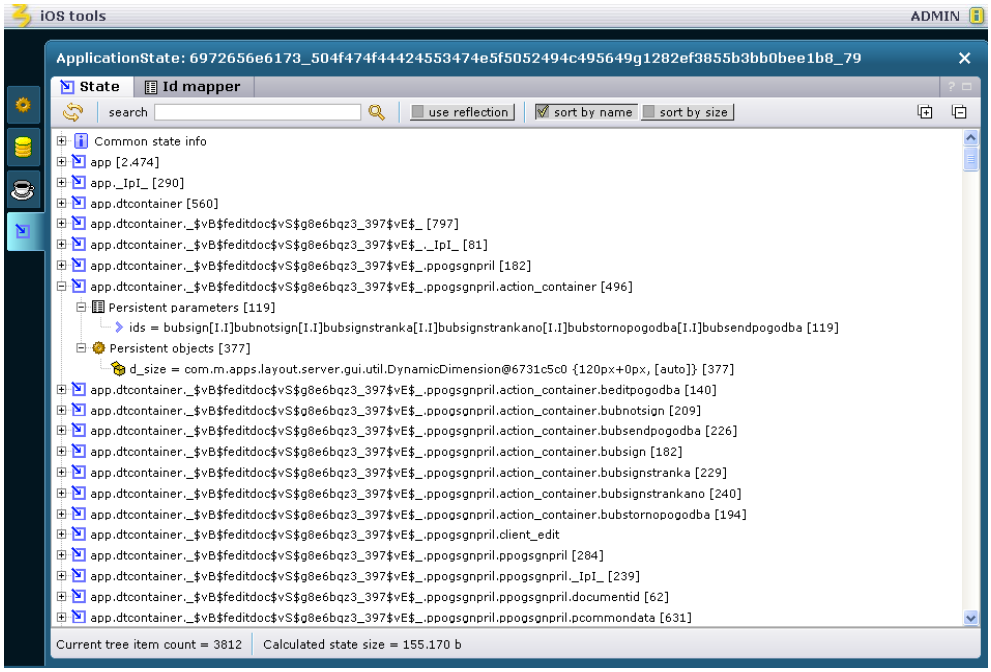
Del ogrodja 3iOS so sistemske aplikacije. Gre za aplikacije, ki omogočajo uporabo storitev ter upravljanje in preverjanje delovanja ogrodja v produkciji. Vsaka storitev, ki jo 3iOS **neposredno** nudi končnemu uporabniku, je pripravljena kot sistemska aplikacija. Večinoma gre za spletne aplikacije razvite z ogrodjem 3iGUI, v določenih primerih pa se uporabljajo tudi obogatene spletne aplikacije, ki se v brskalniku izvajajo v javanskem navideznem stroju (uporaba tehnologije Java Applet). Sistemske aplikacije so lahko namenjene vsem uporabnikom ali pa samo upravljalcem. Za vse sistemske aplikacije velja, da so zamenljive. 3iOS ima lahko za isto storitev hkrati nameščenih več aplikacij. Od običajnih aplikacij se sistemske razlikujejo v tem, da so velikokrat vezane na določeno implementacijo 3iOS-ovega jedra in lahko zaobidejo javne vmesnike ogrodja. Pri tem imajo dostop do vmesnikov in modulov, ki običajnim aplikacijam niso na voljo. Število in vrsta sistemskih aplikacij se razlikujeta glede na implementacijo jedra, vendar večinoma zajemajo sledeče skupine:

- aplikaciji za **prijavo** in **odjavo** iz sistema,
- aplikacija za **menjavo gesla** (če jedro predvideva uporabo gesel),
- aplikacije za **pošiljanje zahtevkov** upravljalcu (pozabljena gesla, prijave napak, predlogi za izboljšave itd.),
- aplikacija **osnovni meni**, ki vsebuje povezave na ostale aplikacije, do katerih ima dostop uporabnik,
- **upravljalske** aplikacije za urejanje uporabniških računov, pregled delovanja sistema, pregled dnevniških datotek itd.,

- aplikacije za **podporo razvoju**, ki na primer omogočajo spreminjanje videza aplikacij v stvarnem času,

- **aplikacije-ogrodja**, ki jih predstavljajo hitro razvite in ozko specializirane aplikacije. Največkrat gre za majša ogrodja, ki jih razširimo z uporabo nastavitvenih datotek XML in s tem dobimo „poizvedovalnike“ po specifičnih podatkih, enostavne čarovnike za izdelavo dokumentov in podobne „pomožne“ aplikacije. Tako izdelane aplikacije so uporabne tudi za hitro izdelavo prototipov in predstavitev.

Na sliki 4.1 je prikazan primer sistemske aplikacije, ki je namenjena upravitelju sistema in omogoča pregled vseh aktualnih stanj aplikacij, registra, bazena povezav, aktualne porabe pomnilnika itd.



Slika 4.1: Primer sistemske aplikacije v 3iOS-u, ki je bila razvita z ogrodjem 3iGUI

4.3.7 Potek razvoja

Pred začetkom snovanja ogrodja sem na podlagi pogovorov s sodelavci pripravil seznam zahtev oziroma funkcionalnosti, ki naj bi jih ogrodje podpiralo. Sprva je bilo ogrodje 3iOS mišljeno samo kot nadgradnja aktualnega spletnega ogrodja, ki smo ga takrat uporabljali, a je bilo to ogrodje preveč togo in ozko usmerjeno, da bi ga bilo mogoče enostavno prilagoditi novim zahtevam in željam. Največ težav sem imel s poskusi definicije modulov novega ogrodja, saj so bile zahteve raznolike. Poleg tega pa nisem želel zavreči obstoječih rešitev, ki so bile dobro preizkušene v praksi. Odločil sem se, da staro ogrodje ohranim in ga poskusim povezati z novo nastajajočim 3iOS-om.

Po pripravi zasnove, je bilo potrebno določiti potek razvoja. Bistvo ogrodja 3iOS je zaledna podpora aplikacijam. Če teh še ni, je ogrodje težko testirati in preveriti uporabnost v praksi. Navadno pa tudi velja, da se prav pri uporabi ogrodja pokažejo podrobne zahteve glede vmesnikov in njihovih implementacij. Zaradi tega sem se po predstavitvi grobe zasnove ogrodja odločil, da pred implementacijo ogrodja razvijem predložni stroj in 3iGUI. Ker sta bila oba omenjena modula predvidena kot glavna uporabnika storitev 3iOS-a, sem moral poiskati najenostavnejšo zamenjavo zanj. Pri tem sem predvidel, da predložnega stroja in ogrodja 3iGUI v prvem koraku ne bom razvil do potankosti, temveč dovolj, da bomo v podjetju lahko preizkusili delovanje celote in kasneje odločili smer postopnega razvoja. Kot nadomestek 3iOS-a sem pripravil definicijo enega samega vmesnika, ki je znal graditi določene vrste objektov in pridobivati podatkovne tokove. Implementacija vmesnika je bila zelo enostavna in se je večinoma naslanjala na obstoječe rešitve, ki smo jih že uporabljali v praksi. Celoten sistem v prvem koraku ni imel predvidenih varnostnih mehanizmov, nikakršne optimizacije, omejen nabor gradnikov uporabniškega vmesnika, omejen nabor predlog skupnega videza aplikacij, enostavnejšo razdelitev map in omejen obseg nastavitvenih datotek.

Predložni stroj sem razvil in testiral neodvisno od preostalih dveh modulov. Ker pa so se na aktualnih projektih pokazale potrebe za njegovo uporabo, smo se odločili, da ga uporabimo v produkciji pred zaključkom prve različice 3iOS-a. Predložni stroj sem moral zato povezati z obstoječim (starejšim) ogrodjem in nastajajočim 3iOS-om. Razvoj ogrodij 3iGUI in 3iOS je potekal v zaporednih ciklih. Pri tem sem postopoma razvil vse pomembnejše module ogrodja in sproti pripravljal dokumentacijo.

Delovanje ogrodja in njegovo uporabnost sem sproti testiral z razvojem sistemskih aplikacij. Prav sprotni razvoj aplikacij je pomagal pri odkrivanju pomankljivosti in dodajanju novih funkcionalnosti. Prav tako sem v posameznih razvojnih ciklih dodelal glavne vmesnike. Ti so najpomembnejši del ogrodja 3iOS, zato sem jim posvetil največ časa. Implementacija vmesnikov je bila na začetku manj pomembna in je služila zgolj potrditvi koncepta. Ko so bili najpomembnejši vmesniki dovolj dodelani, sem implementaciji posvetil več pozornosti in nekatere module dodelal, druge pa razvil na novo. Povsod, kjer je bilo možno, sem uporabil že razvite razrede in storitve obstoječih rešitev.

Razvoj 3iOS-a se je po zaključku prve različice umiril. Vse naslednje različice so gradile na obstoječi kodi in niso predstavljale večje količine dela. Kot se je izkazalo kasneje, sta predložni stroj in 3iOS predstavljala manjši del celotnega razvoja, saj je daleč največ dela zahteval 3iGUI.

4.4 Predložni stroj

Pri zasnovi predložnega stroja sem želel predvsem določiti enostavna pravila za sestavljanje predlog in omogočiti uporabo poljubnih besedilnih formatov. Pri tem sem se želel izogniti mešanju kode in vsebine predlog, kot to omogočajo strani JSP. Za ta modul ogrodja sem predvidel tudi dodaten vmesnik za dostop do predlog, ki je zakril neposredne klice modula 3iFS. Cilj je bila šibka sklopljenost s 3iOS-om (in modulom 3iFS), saj smo predložni stroj želeli uporabiti tudi brez 3iOS-a.

Predložni stroj je modul ogrodja, ki omogoča uporabo besedilnih predlog za sestavljanje različnih tipov besedilnih dokumentov. Omogoča sestavljanje samostojnih dinamičnih strani HTML in posameznih pogledov aplikacij. V primerih sestavljanja pogledov ga uporabljamo za postavitev komponent na vsebnikih. Glavne značilnosti predložnega stroja so:

- sestavljanje dokumentov sledi načelom objektno usmerjenega programiranja,
- predloge so shranjene v 3iFS-u,
- končne dokumente sestavlja v stvarnem času,
- dokumenti so lahko sestavljeni iz statičnih predlog in dinamično generiranih podatkov,

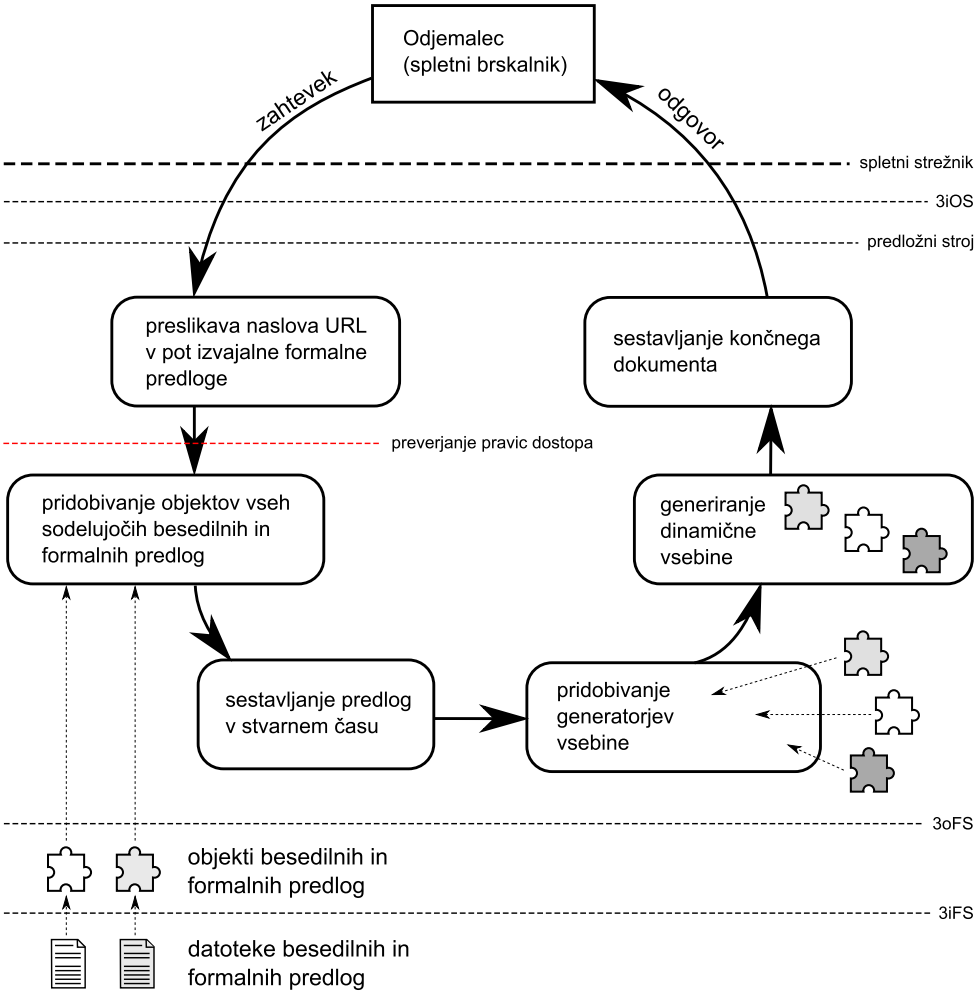
- mešanje programske kode in vsebine predlog ni omogočeno.

Predložni stroj je popolnoma samostojen modul, ki se lahko izvaja tudi brez navezave na 3iOS (3iFS). Temelji na dveh tipih datotek. **Prve** so **besedilne predloge**. Največkrat so to datoteke, ki vsebujejo del dokumenta HTML. Te datoteke imajo navadno končnico `*.template.html`. Gre za običajne besedilne datoteke, v katere lahko vnesemo posebne značke. Značke predstavljajo mesta v dokumentu, v katera bo predložni stroj vrinil vsebino. Oblika značk ni strogo predpisana, a se pri njihovi deklaraciji držimo skupnega dogovora glede označevanja, ki predvideva značke oblike `[%TO_JE_ZNACKA%]`. **Drugi** tip datotek so datoteke XML, ki jih imenujemo **formalne predloge**. Te vsebujejo deklaracije značk, njihovo povezavo z generatorji vsebine in pravila za sestavljanje predlog. Formalne predloge imajo končnico `*.layout.xml`. **Generatorji vsebine** so javanski objekti, ki jih predložni stroj zgradi in uporabi, ko želi zamenjati značko s končno vsebino.

Predložni stroj podpira dva osnovna načina uporabe, in sicer aktivnega in pasivnega. **Aktivni** način uporabe omogoča neposredno izvajanje predložnega stroja brez dodatnega programiranja. Končni uporabnik lahko dostopa neposredno do formalne predloge prek naslova URL. Pri tem predložni stroj poskrbi za preslikavo URL-ja v pot do formalne predloge v 3iFS-u. Pod določenimi pogoji lahko predložni stroj izvede najdeno formalno predlogo. To stori tako, da glede na vsebino datoteke sestavi vse potrebne besedilne predloge in uporabi generatorje vsebine. Rezultat sestavljanja je navadno dokument HTML, ki ga predložni stroj vrne klicatelju. Opisan postopek je prikazan na sliki 4.2. **Pasivni** način uporabe predložnega stroja je namenjen drugim modulom. Predložni stroj omogoča dostop do svojih storitev prek programskega vmesnika API. Ta omogoča dostop do formalnih predlog v 3iFS-u ter njihovo sestavljanje in izvajanje v stvarnem času. Formalne predloge so predstavljene z javanskimi objekti, ki jih s klicem posebne metode lahko izvedemo. Rezultat izvajanja je seznam bajtov, ki predstavljajo končni dokument. Pasivni način uporabe izkoriščajo nekateri vsebniki uporabniškega vmesnika v ogrodju 3iGUI.

4.4.1 Objektno usmerjene predloge

Pri sestavljaju dokumentov omogoča predložni stroj uporabo načel objektno usmerjenega programiranja. Dokumente lahko sestavimo s pomočjo večjega števila formalnih predlog (in pripadajočih besedilnih predlog). Pri tem si



Slika 4.2: Shema izvajanja predložnega stroja

lahko pomagamo z dedovanjem in kompozicijo. Dedovanje je omogočeno z izpeljevanjem formalnih predlog. Kompozicijo pa omogočajo posebne značke - vsebniki. Te značke lahko vsebujejo gnezdene formalne predloge. Formalne predloge ločimo na korenske, izpeljane, izvajalne in abstraktne. **Izvajalne** so namenjene neposrednemu izvajanju prek predložnega stroja, abstraktne pa so namenjene nadaljnjemu izpeljevanju. **Abstraktne** formalne predloge vsebujejo vsaj eno abstraktno značko, kar pomeni, da ta nima določenega generatorja vsebine.

Za **korenske formalne predloge** velja, da njihova vsebina ni izpeljana iz nobene druge formalne predloge. Taka formalna predloga je vedno vezana ne neko besedilno predlogo. Ta predstavlja najbolj splošen del dokumenta. Primer je predloga s standardno glavo in repom. Korenske formalne predloge so največkrat abstraktne, saj iz njih navadno izpeljemo večje število izvajalnih formalnih predlog.

Izpeljane formalne predloge izpeljejo eno od korenskih formalnih predlog ali drugo izpeljano predlogo. Pri tem podedujejo vse starševe značke. Imajo natanko enega starša in nimajo svoje besedilne predloge, saj tudi to podedujejo od starša. Pri izpeljevanju značk lahko popolnoma prekrijejo starševe značke, lahko pa jim dodajo svojo vsebino (lahko jo vrinejo pred starševo značko ali za njo). Posamezna izpeljana formalna predloga je lahko abstraktna ali izvajalna. Abstraktna je v primeru, če ima deklarirano vsaj eno abstraktno značko ali pa je vsaj eno tako značko podedovala od starša in je ni prekrila.

4.4.2 Tipi značk

Formalne predloge vsebujejo deklaracije značk, ki so lahko različnih tipov:

- `[abstract]` - abstraktna značka, ki nima določenega generatorja vsebine. Če formalna predloga vsebuje vsaj eno tako značko, potem ni zmožna izvajanja.
- `[empty]` - značka, ki se pri generiranju dokumenta izbriše iz predloge.
- `[container]` - značka, ki omogoča uporabo kompozicije z vsebovanjem druge formalne predloge. Globina gnezdenja formalnih predlog ni omejena.
- `[resource]` - značka, ki se pri generiranju dokumenta zamenja z vsebino nekega podatkovnega vira. V praksi sta podprta dva vira, in sicer neposredno vneseno besedilo in besedilo iz poljubne datoteke v 3iFS-u.
- `[byte_producer]` - značka, ki generira svojo vsebino s pomočjo posebnega objekta, ki implementira vmesnik `ByteProducer`. Gre za uporabo zelo enostavnih objektov, ki s pomočjo vhodnih parametrov generirajo seznam bajtov. Navadno jih uporabljamo tam, kjer je potrebno generirati zelo enostavno dinamično vsebino, kot sta na primer aktualni datum ali ime trenutno prijavljenega uporabnika.

- `[servlet]` - značka, ki je po delovanju podobna znački `[byte_producer]`, le da je generator vsebine v tem primeru kar servlet. Ta značka se uporablja v primerih, ko je generirane vsebine več ali je generiranje le-te bolj zapleteno (na primer navezava na trenutno sejo).
- `[web_application]` - značka, ki omogoča izvajanje aplikacije. Ta značka uporablja najbolj kompleksen generator vsebine, katerega generirana vsebina predstavlja aktualen pogled aplikacije, ki je „izrisan“ z označevalnim jezikom HTML.

4.4.3 Izvajanje aplikacij

Vstopna točka za uporabo vsake aplikacije v 3iOS-u je njena formalna predloga. Gre za izvajalno formalno predlogo, ki je navadno izpeljana iz ene od sistemskih formalnih predlog. Navadno se imenuje `index.layout.xml` in se nahaja v domači mapi aplikacije. Pravice dostopa do nje so določene v že opisani namestitveni datoteki `index.app.xml`, ki se nahaja v isti mapi. Z uporabo formalnih predlog je možno enostavno spremeniti osnovni videz aplikacije tako, da zamenjamo pot do „starša“. Možno je tudi zamenjati vsebino sistemskih formalnih predlog in tako spreminiti postavitev in videz vsem aplikacijam hkrati. Ker 3iFS omogoča preslikovanje vsebine map v stvarnem času, lahko to v praksi izkoristimo za sprotno spreminjanje videza določenih aplikacij. To pride prav v času razvoja oziroma priprave zasnove videza (hitro testiranje postavitve), v primeru morebitnih nujnih popravkov v produkciji in v primeru večjih sprememb videza, kjer je možno v prehodnem obdobju izvajati isto aplikacijo s starim ali novim videzom.

4.4.4 Potek razvoja

Zasnova predložnega stroja je v primerjavi z ogrodjema 3iOS in 3iGUI enostavnejša. Pri snovanju predložnega stroja sem poskušal doseči enostavnost priprave predlog in hkrati izkoristiti „moč“, ki jo ponuja ideja objektno usmerjenega programiranja. Predvsem sem želel doseči čim večjo stopnjo **ponovne uporabe predlog**. Predložni stroj je bil od začetka predviden kot samostojen modul, ki omogoča neposredno uporabo predlog prek preslikovanja URL naslovov. Pri tem je bilo potrebno rešiti vprašanje shrambe predlog, njene organizacije in preslikovanja naslovov URL v akcije predložnega stroja. Za shrambo je bil dolgoročno predviden bodoči modul 3iFS, ki pa v času razvoja predložnega stroja še ni bil implementiran. Zato sem za potrebe predložnega

stroja začasno pripravil vmesnik, ki je predstavljal okrnjeno funkcionalnost modulov 3iFS in 3oFS. Implementacija vmesnika je bila zelo enostavna, a je omogočala nadaljnje delo in testiranje predložnega stroja.

Razvoj predložnega stroja ni predstavljal večjega izziva. Za najbolj zahtevne so se pokazali le objekti, ki skrbijo za predpomnjenje predlog s podporo spremembam v stvarnem času. Po zaključenem razvoju prve različice predložnega stroja sem ga (začasno) priredil in povezal s tedaj aktualnim ogrodjem (predhodnikom 3iOS-a). S kasnejšo vpeljavo 3iOS-a sem delo na predloženem stroju dokončno zaključil.

4.5 Aplikacije in uporabniški vmesniki 3iGUI

Pri snovanju ogrodja za izgradnjo uporabniških vmesnikov sem poskušal posnemati sorodna ogrodja, ki so namenjena namiznim aplikacijam. Razlog je bil v predhodnih izkušnjah z razvojem namiznih aplikacij in „klasičnih“ spletnih aplikacij. Pristopi pri razvoju obeh tipov aplikacij so se zelo razlikovali, kjer mi je bil razvoj za namizje veliko bolj všeč, saj sem lahko več časa posvetil vsebinskim vprašanjem ne pa skrbi za vzdrževanje stanja in neprestanemu prilagajanju in testiranju videza. Drugi razlog za posnemanje namizja je bila „objektnost“ ogrodij za namizje, ki omogočajo ponovno uporabo. Za posnemanje znanih ogrodij sem se odločil tudi zato, ker sem ta ogrodja dobro poznal in sem lahko uporabil že znane in preizkušene vzorce, ki sem jim zaupal.

3iGUI je ogrodje, ki omogoča razvoj in izvajanje aplikacij v povezavi s 3iOS-om. Aplikacije se izvajajo **na strežniku**. Ogrodje predvideva izgradnjo uporabniških vmesnikov z uporabo standardnih gradnikov, kot so komponente uporabniškega vmesnika (v nadaljevanju komponente), vsebniki uporabniškega vmesnika (v nadaljevanju vsebniki), dogodki, poslušalci dogodkov in podatkovni modeli (v nadaljevanju modeli). Ogrodje se po načinu uporabe komponent zgleduje po ogrodju Swing. V večji meri temelji na uporabi vzorca MVC. Pomembna značilnost ogrodja je zmožnost **vzdrževanje stanja** komponent. 3iGUI je ogrodje po načelu sive škatle, ki za implementacijo vročih točk predvideva uporabo kompozicije (sestavljanje hierarhije komponent) in dedovanja (izpeljevanje komponent). Vsebuje večje število predpripravljenih komponent, kot so običajna vnosna polja, spustni meniji, gumbi, podatkovne tabele, dialogna okna itd. Razvoj aplikacije zajema sestavljanje obstoječih komponent v hierarhijo, izpeljevanje vsebnikov in implementacijo poslušalcev

dogodkov. Ogrodje omogoča tudi razvoj lastnih komponent in njihovo ponovno uporabo.

4.5.1 Glavne značilnosti ogrodja 3iGUI

Kot sem že omenil, razvoj aplikacij z ogrodjem 3iGUI spominja na razvoj z ogrodjem Swing. Pri tem je zaradi drugačnega okolja nekaj pomembnih razlik. **Prva** očitna razlika je zahteva po enolični oznaki vseh komponent, ki so vsebovane v istem vsebniku. Oznaka komponent je enostaven niz znakov (String). Ta omogoča povezavo komponent z značkami v formalnih predlogah, ko se 3iGUI povezuje s predložnim strojem. Poleg tega je možno iz vsebnika vedno priklicati instanco komponente tudi s pomočjo njene oznake. **Druga** opazna razlika je način grajenja komponent, saj 3iGUI privzeto vsebuje tudi mehanizme za grajenje komponent na zahtevo. **Tretja** razlika pa so mehanizmi za vzdrževanja stanja komponent. Ti so v večini primerov „skriti“ in se z njimi ni potrebno posebej ukvarjati, v primeru zahtevnejših aplikacij ali razvoja lastnih komponent, pa jih razvijalec mora uporabiti.

Čeprav je bilo ogrodje 3iGUI razvito za splet, programski vmesnik API to v večji meri zakrije. Razvijalci aplikacij se tako ne srečujejo s tipičnimi spletnimi tehnologijami, kot so HTML, CSS, JavaScript, AJAX itd. Prav tako se tudi pri razvoju lastnih komponent ni potrebno vedno ubadati z omenjenimi tehnologijami. Navezava na spletne tehnologije se pojavi zgolj v enem od slojev ogrodja, ki mu pravimo **kontekst izrisa**. Ta je predstavljen z objektom, ki vsebuje tako imenovane izrisovalce. V nadaljevanju poglavja sem dodatno razdelal omenjene značilnosti ogrodja 3iGUI skupaj s primeri kode.

Hierarhija komponent

Komponente v ogrodju 3iGUI delimo na primitivne komponente in na vsebnike. **Primitivne komponente** so tiste, ki ne vsebujejo drugih komponent. Največkrat gre za enostavne komponente, kot so na primer gumbi in vnosna polja. **Vsebniki** so komponente, ki lahko vsebujejo druge komponente. Z njihovo pomočjo zgradimo celotno hierarhijo komponent uporabniškega vmesnika. Od primitivnih komponent se razlikujejo tudi po tem, da navadno potrebujejo dodatno informacijo o **postavitvi** vsebovanih komponent. Poseben primer vsebnika je komponenta `WebApplication`, ki vedno predstavlja koren hierarhije. Sledi izsek iz kode, ki predstavlja najenostavnejši način sestavljanja hierarhije komponent:

```
// priprava vsebnika
WebPanel panel = new WebPanel("panel");
panel.setTitle("Primer vsebnika z enim gumbom");

// priprava gumba
WebButton button = new WebButton("button");
button.setLabel("pritisni me");

// gumb dodamo vsebniku
panel.addComponent(button);
```

Implementacija vzorca MVC

Aplikacija v ogrodju 3iGUI je sestavljena iz komponent in aplikativne kode. Aplikativna koda se večinoma izvaja ob poslušanju dogodkov, ki jih prožijo komponente. Vzorec MVC je uporabljen v vsaki komponenti posebej, kjer komponenta predstavlja pogled in krmilnik, vsebuje pa še referenco na model. 3iGUI pozna dva osnovna tipa modelov, in sicer enostavne in kompleksne. **Enostavni modeli** so objekti, ki implementirajo vmesnik IDataModel. Definicija vmesnika je sledeča:

```
public interface IDataModel<T> {

  public T getMValue();
  public void setMValue(T mValue);

}
```

Parameter `T` se razlikuje glede na zahteve posamezne komponente. Poleg prikazanega vmesnika igra med enostavnimi modeli pomembno vlogo tudi abstraktni razred `ReadOnlyDataModel`, ki je namenjen implementaciji modelov, iz katerih lahko komponente samo berejo. Enostavni modeli se večinoma uporabljajo v primerih, ko model vrača eno samo vrednost. Za vzdrževanje stanja enostavnih modelov so zadolžene njihove komponente. Komponente lahko uporabljajo tudi **implicitne** modele. To navadno velja za zelo enostavne komponente in tiste, ki jim nismo posebej določiti modela. Sledeči izsek kode predstavlja uporabo dveh instanc komponente `WebLabel` s pripravljenim modelom in implicitno brez njega:

```
// primer s pripravo modela
```

```
WebLabel rLabel = new WebLabel("random");
rLabel.setLabel(
  new ReadOnlyDataModel<String>() {
    public String getMValue() {
      return String.valueOf(Math.random());
    }
  }
);

// primer z uporabo implicitnega modela
WebLabel simpleLabel = new WebLabel("simple");
simpleLabel.setLabel("to je primer vsebine");
```

Kompleksne modele uporabljajo komponente, ki omogočajo delo z večjo količino podatkov. Primera takih komponent sta komponenti `WebGrid` in `WebTree`. Prva predstavlja tabelaričen seznam podatkov, ki jih lahko pregledujemo s premikanjem po straneh. Druga pa predstavlja podatke združene v drevesno strukturo (primer komponente prikazuje slika 4.1). Ker je dostop do podatkov bolj zahteven, imajo kompleksni modeli tipično večje število metod. Prav tako ti modeli nimajo strogo predpisanega skupnega vmesnika, saj se ta prilagaja vsaki komponenti posebej. Za razliko od enostavnih modelov morajo kompleksni modeli sami poskrbeti za vzdrževanje svojega stanja.

Za določene komponente smo razvili večje število kompleksnih modelov. Tako ima na primer komponenta `WebGrid` kar nekaj modelov, ki med drugim omogočajo prikaz in delo s podatki neposredno iz podatkovnih baz, iz map in datotek modula 3iFS ali pa neposredno iz seznama objektov v pomnilniku.

Izpeljevanje komponent

Poleg možnosti uporabe kompozicije pri sestavljanju hierarhije komponent, predvideva 3iGUI tudi uporabo dedovanja z izpeljevanjem komponent. Večino komponent je možno izpeljati in jih nadgraditi. Ogrodje lahko v določenih primerih celo zahteva izpeljevo točno določene komponente. Primer je že omenjena komponenta `WebApplication`, ki jo moramo obvezno izpeljati, ko začnemo z razvojem nove aplikacije. Največkrat izpeljujemo vsebnike, v katerih nato pripravimo kodo za grajenje vsebovanih komponent in poslušanje njihovih dogodkov. Izpeljevanje pride prav tudi v primeru primitivnih komponent. Navadno jih izpeljemo takrat, ko želimo pridobiti ožje usmerjene (spe-

cializirane) komponente. Kot primer lahko navedem vnosno polje za enotno matično številko občana (EMŠO), ki ima za razliko od splošnega vnosnega polja že omejeno dolžino vnosa in predvideno preverjanje veljavnosti vsebine (angl. validation), ki zahteva trinajstmestno število zaključeno s pravilno kontrolno številko.

Vzdrževanje stanja komponent

Vzdrževanje stanja aplikacije je v ogrodju 3iGUI definirano kot vzdrževanja stanja hierarhije komponent. Vsako stanje aplikacije je vedno **vezano na eno okno (ali zavihek) brskalnika**. Vsaka komponenta v aplikaciji ima lastno stanje, ki ga v odločenem trenutku shrani in kasneje zopet prebere. Stanje celotne aplikacije je tako sestavljeno iz množice stanj posameznih komponent. Stanje vsake komponente je množica objektov tipa String ali pa takih, ki implementirajo vmesnik `Externalizable`. Ta omogoča preslikavo objekta v seznam bajtov (serializacija) oziroma grajenje objekta iz seznama bajtov. Stanje celotne aplikacije je zato možno shraniti v poljuben podatkovni tok in ga kasneje prebrati nazaj v pomnilnik. Komponente in njihova stanja je možno popolnoma ločiti. Pri tem ogrodje poskrbi samo za trajnost stanj, komponente pa zavrže.

Ločevanje stanj in komponent omogoča lažjo prenosljivost stanj. Shranjeno stanje aplikacije pa je manjše, če ne vsebuje celotne hierarhije komponent temveč samo podatke, za katere zahtevamo trajnost. Pri vzdrževanju svojega stanja sodeluje aplikacija z že omenjenim modulom application_state_manager. Vmesnik tega modula je sledeč:

```
public interface IApplicationStateManager {

  public void addAndUnlockState(String id, Serializable state);
  public Object getAndLockState(String id);
  public void unlockState(String id);

}
```

Vmesnik predstavlja odlagališče stanj za vsako aplikacijo v 3iOS-u. Vse implementacije vmesnika morajo pri delu s stanji upoštevati določena pravila. Gre predvsem za nadzorovanje konkurenčnih dostopov do množice stanj v večnitnem okolju in omejevanje konkurenčnih dostopov do posameznega stanja.

Osnovo pravilo vzdrževanja stanj določa, da se mora vsako stanje aplikacije vedno nahajati v natanko enem od sledečih „scenarijev“:

- stanje je v pomnilniku, je v uporabi (s strani aplikacije) in je zaklenjeno,
- stanje je v pomnilniku, ni v uporabi in je odklenjeno,
- stanja ni v pomnilniku, ni v uporabi in je odklenjeno.

3iGUI predvideva izvajanje aplikacij, ki dostopajo do stanja z eno nitjo naenkrat. Zato mora modul application_state_manager poskrbeti za pravilno zaklepanje stanj. Zagotoviti mora, da bo pri morebitno večjem številu konkurenčnih niti le ena naenkrat dostopala do stanja aplikacije, vse ostale pa bodo po potrebi čakale v vrsti. To velja za vsako posamezno instanco aplikacije (okno ali zavihek brskalnika) posebej. Zahtevki posameznih aplikacij se seveda lahko obravnavajo vzporedno. Osnovna implementacija modula deluje na sledeči način:

- Aplikacija poskuša na začetku obdelave zahtevka HTTP pridobiti shranjeno stanje (če obstaja) s klicem metode `getAndLockState`. Če je stanje zaklenjeno, mora počakati v vrsti. Ko dobi stanje v uporabo, je to hkrati že zaklenjeno za druge niti. Stanje pridobi s pomočjo identifikacijske oznake, ki jo prebere iz množice poslanih parametrov s strani odjemalca. Če identifikacije ne dobi, pripravi novo stanje.
- Pred koncem obdelave zahtevka aplikacija shrani novo stanje s klicem metode `addAndUnlockState`. Pri tem se stanje odklene in je na voljo drugi niti. Po tej točki aplikacija ne sme več uporabiti stanja - navadno se instanca aplikacije zavrže.

Application_state_manager, ki ga uporabljamo v produkciji, deluje tako, da vzdržuje stanje določene aplikacije v pomnilniku samo določen čas. Če je stanje več časa neakativno (nihče ne dostopa do njega), ga shrani iz pomnilnika v posebno mapo prek modula 3iFS. Po potrebi stanje iz 3iFS-a prebere nazaj v pomnilnik, kjer zopet omogoči njegovo neposredno uporabo. Če je stanje shranjeno (neaktivno) v 3iFS-u daljše obdobje, ga zbriše.

Grajenje komponent

V ogrodju 3iGUI lahko komponente gradimo neposredno ali na zahtevo. **Neposredno grajenje** komponent je grajenje, kjer instanco komponente

navadno zgradimo v konstruktorju nekega vsebnika. Ta način grajenja komponent se uporablja tudi v drugih primerljivih ogrodjih, kot je na primer Swing.

V praksi se pokaže, da je v vsakem zahtevku HTTP potrebno graditi samo določeno vejo celotne hierarhije komponent. Pri uporabi zavihkov je na primer potrebno zgraditi samo vsebino izbranega zavihka. Grajenje vseh ostalih zavihkov ter branje in shranjevanje njihovega stanja bi bilo v večini primerov potratno. V ta namen 3iGUI uporablja **grajenje na zahtevo**. Pri grajenju na zahtevo je potrebno uporabiti vmesnik „graditelja“:

```
public interface GUIBuilder {

  public SimpleWebComponent createComponent(String id);

}
```

Implementacija vmesnika mora prepoznati podano identifikacijsko oznako in za njo zgraditi instanco komponente. Največkrat so graditelji kar vsebniki sami.

Uporaba grajenja na zahtevo se je v praksi izkazala za zelo učinkovito, saj se je z njeno vpeljavo število zgrajenih komponent na zahtevek zelo zmanjšalo. Pri sledenju aktivnostim na stanju tipične aplikacije pa se je pokazala „lokalnost“ dostopov do stanja. Zaporedni zahtevki povzročajo spremembe stanja na omejenem številu komponent, kjer gre večinoma za iste komponente ali pa za komponente, ki so si dovolj blizu[4].

Dogodki in poslušalci

Komponente lahko prožijo različne dogodke. Gumbi prožijo dogodke ob pritisku, izbirna polja ob izbiri podatka itd. Veliko večino dogodkov v ogrodju 3iGUI prožijo uporabniki z uporabo komponent (klik miške, pritisk tipke itd.). Poleg opisanih neposrednih dogodkov 3iGUI predvideva tudi programsko proženje dogodkov. V takem primeru se dogodek proži ob klicu določene metode komponente. Zaradi tega nekateri dogodki vsebujejo tudi informacijo o viru proženja (uporabnik ali koda). Za prestrezanje dogodkov je potrebno implementirati enega ali več poslušalcev.

[4]Na primer dve komponenti, ki sta vsebovani v istem vsebniku.

Proženje dogodkov je pomemben korak v življenjskem ciklu komponent. Ogrodje mora zagotoviti, da se proženje dogodkov zgodi ob pravem trenutku, ko je hierarhija komponent veljavna in vsebina modelov ažurna. Za pripravo dogodkov so odgovorne komponente same. Komponenta mora pri branju odjemalčevih podatkov ugotoviti, ali je potrebno proženje dogodka (na primer, ali je uporabnik pritisnil na gumb). Ko komponenta ugotovi, da je potrebno prožiti dogodek, pripravi dogodek, vendar ga ne proži takoj, temveč svojo namero sporoči aplikaciji. Aplikacija shrani referenco komponente in izvede proženje dogodka šele v za to primernem trenutku.

Če ima komponenta več poslušalcev, jih obvešča po vrsti glede na njihovo mesto v seznamu. Poslušalci vsebujejo del aplikativne kode, ki največkrat kliče metode modelov. Ker imamo pri definiranju poslušalcev precej svobode, se velikokrat pojavi vprašanje, katerim razredom je smiselno dodati možnost poslušanja dogodkov. V praksi za poslušanje dogodkov uporabljamo predvsem vsebnike (poslušajo dogodke vsebovanih komponent) in objekte deklarirane z anonimnimi notranjimi razredi. V nadaljevanju je podan primer obeh tipov poslušalcev:

```
public class MyPanel extends WebPanel
                     implements WebSelectionListener {

  private WebButton button = new WebButton("button");
  private WebCombobox combo = new WebCombobox("combo");

  public MyPanel(String id) {
    super(id);

    // gumb
    button.setLabel("pritisni me");
    // poslušalec je anonimni notranji razred
    button.addWebActionListener(
      new WebActionListener() {
        public void actionPerformed(WebActionEvent ev) {
          doPush();
        }
      }
    );
    addComponent(button);
```

```
    // izbirno polje
    combo.addItem(new WebItem("id1","prva izbira"));
    combo.addItem(new WebItem("id2","druga izbira"));
    // poslušalec je vsebnik
    combo.addWebSelectionListener(this);
    addComponent(combo);
  }

  private void doPush() {
    findWebApplication().addEcho(
      new WebEchoOK("To je odziv na pritisk gumba!")
    );
  }

  public void selectionStateChanged(WebSelectionEvent ev) {
    WebItem item = (WebItem)ev.getItemInstance();
    findWebApplication().addEcho(
      new WebEchoOK("Izbrana vrednost je "+item.getValue())
    );
  }

}
```

Nastavitve komponent

Komponentam lahko nastavljamo lastnosti in jih tako prilagodimo potrebam posamezne aplikacije. Pri nastavljanju komponent lahko izbiramo med nastavljanjem v kodi ali zunanjim nastavljanjem. **Nastavljanje v kodi** tipično omogočajo metode `set*` (angl. setter). Prednost nastavljanja v kodi je v tem, da je tako nastavljanje za razvijalca zelo enostavno. Slabost tega pristopa pa je, da imamo v takih primerih mešanje kode z vsebino, kar navadno ni priporočljivo. Vsaka najmanjša sprememba besedila pomeni poseg v kodo. Prav tako pomeni neposredno vnašanje besedil v kodo težave pri morebitni vpeljavi večjezičnosti. Primer neposrednega nastavljanja komponente je prikazan v sledečem izseku kode:

```
WebLabel label = new WebLabel("my_label");
label.setLabel("Testno besedilo");
label.setTooltip("Opis komponente");
```

```
label.setDynamicSize(new DynamicDimension("100px,20px"));
label.setStyleId("bold");
```

3iGUI omogoča tudi **zunanje nastavljanje** oziroma nastavljanje komponent izven kode. Vsaka aplikacija ima v svoji namestitveni mapi predvideno mapo za nastavitve uporabniškega vmesnika. Ta mapa je `/[CURRENT_HOME]/gui/`. Vse komponente lahko nastavimo v eni sami datoteki ali pa gremo v drugo skrajnost in za vsako komponento pripravimo svojo nastavitveno datoteko. V praksi uporabljamo nastavitvene datoteke do nivoja posameznega vsebnika, primitivnim komponentam pa ne določamo svojih nastavitvenih datotek. Zunanje nastavitve navežemo na komponente s pomočjo njihove identifikacijske oznake. Spreminjamo jih lahko tudi v stvarnem času, kar pride prav tako med razvojem in oblikovanjem pogledov aplikacije kot tudi v produkciji, če so kje potrebni manjši popravki pogledov. Primer zunanjega nastavljanja komponente je prikazan v sledečem primeru nastavitvene datoteke:

```
<properties>
  <property name="my_label.p_label">Testno besedilo</property>
  <property name="my_label.p_tooltip">Opis komponente</property>
  <property name="my_label.p_d_size">100px,20px</property>
  <property name="my_label.p_style">bold</property>
</properties>
```

Zunanje nastavitve imajo **zelo pomembno lastnost**, da jih lahko uporabimo vedno, tudi ko je razvijalec že nastavil komponente v kodi. Zunanje nastavitve imajo namreč prednost pred tistimi v kodi, zato jih vedno prekrijejo. Potrebujemo samo identifikacijsko oznako in tip komponente.

Postavitev komponent

Večina komponent v ogrodju 3iGUI je vidnih - torej takih, ki se rišejo na zaslon. Vsaka komponenta je odgovorna za izris lastne vsebine. Za postavitev komponente (koordinate na zaslonu) pa je odgovoren njen vsebnik. Ker so vsebniki hkrati tudi vsebovane komponente, zanje velja isto pravilo. Izris aktualnega pogleda aplikacije temelji na risanju komponent „od zgoraj navzdol“. Najprej se prične risati korenski vsebnik. Med risanjem sproži risanje vsebovanih komponent, ki se rišejo po enakem postopku. Risanje se nadaljuje v „globino“ dokler niso izrisane vse aktulane primitivne komponente.

Vsebniki med risanjem potrebujejo informacijo o postavitvi vsebovanih komponent. Postavitev komponent na pogledih je pomembna, saj prav postavitev določi videz aplikacije in lahko zelo vpliva na preglednost uporabniškega vmesnika oziroma na njegovo prijaznost. 3iGUI omogoča programske in deklarativne postavitve. Programske postavitve podpirajo samo enostavne primere, kot so na primer v vrsto postavljeni gumbi na orodnih vrsticah. Za vse ostale primere pa 3iGUI predvideva uporabo zunanjih nastavitev, pri katerih si lahko pomagamo s predložnim strojem ali drugimi deklarativnimi rešitvami. Najpomembnejši razredi oziroma vmesniki, ki omogočajo postavitev komponent na vsebnikih, so:

- `SimpleLayout` - omogoča enostavno postavitev komponent v vrsto. Med komponente je možno vnesti tudi posebna ločila (angl. separator).
- `Layout` - predstavlja formalno predlogo iz predložnega stroja. Postavitev komponent določimo kar z označevalnim jezikom HTML.
- `IVisualLayout` - določa objekte, ki uporabljajo posebno nastavitveno datoteko XML, kjer je postavitev komponent določena veliko bolj pregledno in enostavno kot z uporabo jezika HTML. Primer kompleksnega pogleda aplikacije, ki uporablja to vrsto postavitve, prikazuje slika 4.3.

Preverjanje veljavnosti vsebine komponent

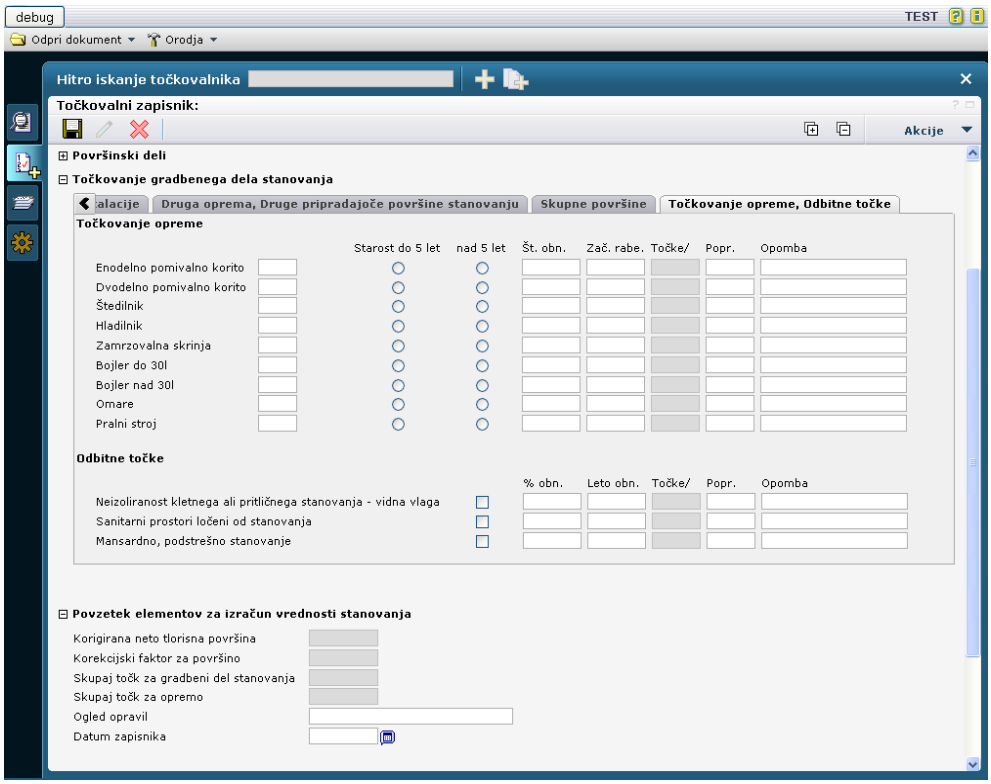
Preverjanje veljavnosti vsebine komponent oziroma njihovih modelov se v ogrodju 3iGUI rešuje z uporabo posebnih objektov - validatorjev. Vsaka komponenta ima lahko več validatorjev, ki se izvajajo po vrsti glede na njihovo aktualno mesto. Validatorji na vsebnikih se navadno ukvarjajo s preverjanjem odvisnosti med vsebovanimi komponentami. Primer iz prakse je vsebnik, ki vsebuje komponente za zajem osebnih podatkov uporabnika. Uporabnik mora obvezno vnesti davčno številko ali EMŠO. Validator na takem vsebniku vrne napako samo v primeru, ko uporabnik ni vnesel nobene od omenjenih številk.

Preverjanje vsebine posamezne komponente je potrebno izvesti s klicem namenske metode. To pomeni, da ogrodje nima predvidene točke, ki bi preverjanje izvedla samodejno. Za izvajanje preverjanja mora vedno poskrbeti razvijalec aplikacije. Po klicu preverjanja se mora sam odločiti, na kakšen način bo obveščal o napakah (obvestilo uporabniku, zapis v dnevniško datoteko itd.).

Preverjanje veljavnosti se tako največkrat izvede kot posledica proženja dogodka. Na primer, ko uporabnik v čarovniku pritisne gumb „naprej“, ali ko po spreminjanju nekega dokumenta pritisne na gumb „shrani“.

4.5.2 Izrisovalci

Izris aktualnega pogleda aplikacije se zgodi za proženjem dogodkov. Izrišejo se samo vidne komponente. Vsaka komponenta ima za izris predvideni dve metodi, in sicer `paint` in `update`. Prva se uporablja za izris pogleda komponente v celoti, druga pa za delni izris pogleda oziroma ažuriranje parametrov videza komponente na odjemalcu (navadno posledica zahtevka AJAX). Komponente lahko v teh dveh metodah neposredno izrišejo svoje poglede, na primer z generiranjem dela HTML-ja, vendar zaradi ločevanja kode komponent od njihovega videza tega v praksi ne počnemo. Komponente zato v omenjenih metodah vedno prikličejo **kontekst izrisa**, prek njega pridobijo primerno in-



Slika 4.3: Primer kompleksne postavitve komponent

stanco izrisovalca ter z njim izvedejo izris. Kontekst izrisa mora vsebovati seznam vseh aktualnih izrisovalcev. Določen je na nivoju celotne aplikacije in ga lahko s pomočjo parametrov zamenjamo tudi v stvarnem času.

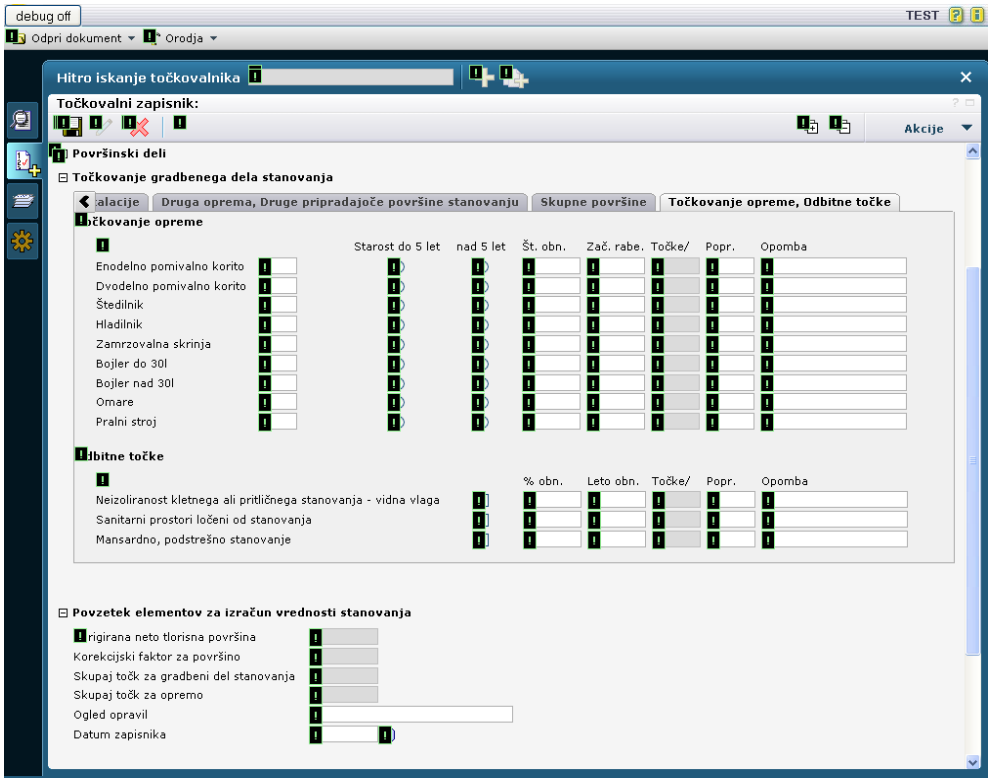
Izrisovalci primitivnih komponent so običajno enostavni. Iz dane instance komponente pridobijo vse „vidne“ podatke in iz njih sestavijo vsebino pogleda, ki jo nato „vpišejo“ v podan izhodni tok. Običajno gre za del dokumenta HTML ali XML oziroma za kodo v JavaScriptu. Rezultat izrisa komponente sicer ni strogo določen in je lahko poljuben niz podatkov, ki jih razume predviden odjemalec. Koda **izrisovalcev vsebnikov** je navadno bolj zapletena. Ti izrisovalci morajo poskrbeti za risanje pogleda vsebnika in vseh vsebovanih komponent. Vsebovanih komponent po navadi ne rišejo neposredno, temveč v določenem trenutku kličejo metodo `paint` za vsako izmed komponent.

Za vsak tip komponente se v kontekstu izrisa ustvari natanko ena instanca izrisovalca, ki riše vse komponente istega tipa. Obstajajo pa tudi primeri uporabe **dekoratorjev** izrisovalcev, ki se zgradijo za vsako instanco komponente posebej. Primer je dekorator, ki pred vsak pogled komponente izriše še posebno ikono, ki omogoča razhroščevanje aplikacije (prikazano na sliki 4.4). Brez uporabe dekoratorja, bi bilo za izris ikone potrebno prilagoditi kodo vseh izrisovalcev, dekorator pa to omogoči v nekaj vrsticah kode.

Za celoten izris pogleda komponente se uporabljajo **običajni izrisovalci**. V večini primerov gre za enostavno opravilo, ki preslika značilnosti komponente in njeno vsebino v del dokumenta HTML. Pri tem običajni izrisovalec upošteva zgolj trenutno stanje komponente, prejšnja stanja pa ga ne zanimajo.

Izrisovalci AJAX omogočajo delno ažuriranje parametrov izrisa namesto vsakokratnega ponovnega izrisa komponent na odjemalcu. Če komponento rišejo prvič, potem jo izrišejo v celoti, enako kot običajni izrisovalci. Pri vseh naslednjih izrisih pa ti izrisovalci uporabijo vsebino predhodnega izrisa in jo primerjajo z vsebino aktualnega izrisa. V primeru, da najdejo razlike, pripravijo kodo v JavaScriptu, ki bo poslana k odjemalcu. Ta koda vsebuje postopek ažururanja sprememb na odjemalcu. Če ne najdejo razlik med predhodnim in aktualnim izrisom, ne narišejo ničesar.

Uporaba tehnike AJAX je v ogrodju 3iGUI zgolj stvar izbire konteksta izrisa. Ogrodje poskuša v največji možni meri zakriti podrobnosti pogovora med odjemalcem in aplikacijo na strežniku. Razvoj aplikacije ne zahteva poglobljenega



Slika 4.4: Primer kompleksne postavitve komponent, ki so bile izrisane v razhroščevalnem načinu izvajanja aplikacije (običajen izris pogleda prikazuje slika 4.3)

znanja o lastnostih odjemalca. Kontekst izrisa je sloj kode, ki je vezan na točno določen tip odjemalca in je hkrati zadolžen za razreševanje morebitnih razhajanj pri različnih odjemalcih istega tipa. Največ težav pri razvoju izrisovalcev se na primer pojavlja zaradi razlik v brskalnikih.

4.5.3 Življenjski cikel komponent

Vsaka komponenta, ki je del hierarhije, sodeluje v življenjskem ciklu aplikacije. Gre za **vrsto vročih in skupnih točk** posamezne komponente, ki se kličejo v določenem vrstnem redu ob vsakem zahtevku odjemalca. Večino vročih točk implementirajo že obstoječe komponente, možno pa jih je prekriti z **izpeljevanjem razredov**. Druge vroče točke predvidevajo uporabo **kompozicije**. Tukaj gre na primer za sestavljanje hierarhije komponent, dodajanje last-

nih graditeljev komponent in prijavo lastnih poslušalcev. Aktualna hierarhija komponent se vedno nahaja v določenem **koraku življenjskega cikla**. Ko vse komponente zaključijo z aktualnim korakom, se življenjski cikel hierarhije premakne v naslednji korak. V času izvajanja zahtevka se lahko določene komponente zgradijo in dodajo v hierarhijo, ko je ta opravila že več korakov. Hierarhija v takih primerih „počaka“ na „pozno“ komponento, da se ob njenem dodajanju v hierarhijo opravijo vsi koraki do aktualnega. Ko komponenta „dohiti“ korake hierarhije, se življenjski cikel le-te nadaljuje.

V splošnem delimo korake življenjskega cikla v dve osnovni področji, in sicer nevarno in varno področje. V **nevarno področje** spadajo koraki, ko hierarhija še ni popolnoma zgrajena in stanje vseh komponent še ni ažurno. Razvijalci, ki prekrijejo metode tega področja, morajo razumeti delovanje ogrodja in se zavedati morebitnih pasti, ki so posledica nepopolne hierarhije in stanja komponent. V veliki večini primerov pa se razvijalci aplikacij ne srečujejo z metodami tega področja. Ogrodje v **varnem področju** zagotavlja, da je hierarhija komponent zgrajena in stanja komponent ažurna. V tem področju se izvaja velika večina aplikativne kode. Seznam najpomembnejših korakov življenjskega cikla komponent v hierarhiji je sledeč:

1. **začetek nevarnega področja**
2. **grajenje komponent in dodajanje v hierarhijo** - izvede se koda v konstruktorjih komponent ali pa koda graditeljev. Vse zgrajene komponente moramo obvezno dodati v hierarhijo.
3. **uporaba zunanjih nastavitev** - ogrodje poskrbi, da je vsaki komponenti omogočen dostop do svojih zunanjih nastavitev. Za branje in uporabo svojih zunanjih nastavitev so zadolžene komponente same.
4. **uporaba zagonskih parametrov aplikacije** - korenski vsebnik (aplikacija) pridobi dostop do svojih zagonskih parametrov, ki so podani v zagonski datoteki aplikacije. Za branje in uporabo teh parametrov je zadolžena posamezna instanca aplikacije.
5. **branje in uporaba zadnjega aktualnega stanja** - aplikacija pridobi zadnje shranjeno stanje, ki ga uporabi za ažuriranje vsebine vseh trenutno zgrajenih komponent.
6. **procesiranje odjemalčevega zahtevka** - vsaka komponenta prebere podatke, ki jih je poslal odjemalec. Podatke, ki so ji namenjeni, uporabi za ažuriranje lastnega stanja.

7. **začetek varnega področja**

8. **prva inicializacija vsebine (modelov) komponent** - če komponenta v 4. koraku ni pridobila lastnega stanja in v 5. koraku ni pridobila podatkov od odjemalca, izvede inicializacijo, ki je namenjena pripravi prvega stanja.

9. **korak pred proženjem dogodkov** - v tem koraku lahko komponente izvedejo določena opravila, ki so potrebna pred proženjem dogodkov.

10. **proženje aktualnega uporabnikovega dogodka** - izvede se proženje dogodka, ki ga je izvedel končni uporabnik (odjemalec). V tem koraku se izvede koda poslušalcev aktualne komponente. Gre za **najpomembnejši korak**, v katerem se izvaja logika aplikacije.

11. **korak pred risanjem celotne aplikacije** - v tem koraku lahko komponente izvedejo določena opravila, ki so potrebna pred njihovim izrisom. Primer je vsebnik, ki riše komponente v zavihkih. Tak vsebnik lahko v tem koraku nastavi vidnost zavihkov tako, da „skrije“ vsebino vseh zavihkov, razen trenutno označenega.

12. **risanje komponent** - komponente izrišejo svoje poglede oziroma v ta namen uporabijo aktualni kontekst izrisa.

13. **začetek nevarnega področja**

14. **korak po risanju celotne aplikacije** - v tem koraku lahko komponente izvedejo določena opravila, ki so potrebna pred shranjevanjem stanja.

15. **priprava aktualnega stanja in shranjevanje le-tega** - komponente pripravijo stanje in ga „objavijo“ v seznamu aktualnih stanj. Ogrodji 3iGUI in 3iOS poskrbita za shranjevanje celotnega stanja aplikacije.

16. **„čiščenje“ komponent** - preden se instance komponent zavržejo, morajo le-te zapreti morebitne reference na zunanje vire podatkov. Prav tako se v tem koraku razdre hierarhija komponent, kar pomeni, da komponente zavržejo referenco na svoje vsebnike, vsebniki pa izpraznijo sezname vsebovanih komponent.

17. **komponente se zavržejo** - generator vsebine v predložnem stroju, ki skrbi za pripravo instance aplikacije in krmili del njenega življenjskega cikla, to instanco ob koncu procesiranja zahtevka zavrže.

4.5.4 Potek razvoja

Ogrodje 3iGUI je v primerjavi s 3iOS-om in predložnim strojem zahtevalo največ dela. Število razredov tega modula je nekajkrat večje v primerjavi z ostalima moduloma, prav tako je bilo snovanje življenjskega cikla aplikacije, trajnosti stanj, konteksta izrisa in posameznih komponent veliko bolj zahtevno.

Prva zasnova, ki sem jo pripravil, je vsebovala enostaven življenjski cikel in vzdrževanje stanja na odjemalcu. Kasneje sem shrambo stanj premaknil na strežnik in pripravil že omenjeni modul application_state_manager. Na začetku razvoja sem sproti razvijal komponente in prvo učno aplikacijo. Razvoj je potekal vzporedno. Takoj ko sem pripravil novo komponento, sem jo uporabil v aplikaciji. Nasprotno sem pri razvoju aplikacije hitro našel pomanjkljivosti komponent in se posledično vračal k njihovemu razvoju. Ko so k razvoju aplikacij pristopili še drugi razvijalci, so se hitro pokazale dodatne potrebe po nadgradnjah kompnent in predvsem po novih komponentah.

V času razvoja sem večkrat razširil življenjski cikel aplikacije in dodajal nove vmesne korake. Prav tako sem določenim komponentam dodal možnost proženja novih dogodkov takoj, ko se je po njih pojavila potreba. Med razvojem se je tudi izkazalo, da bo delo na ogrodju 3iGUI časovno daljše, saj so se zahteve po novih komponentah, dogodkih in modelih stopnjevale. Razvijalci končnih aplikacij so velikokrat priredili oziroma sestavili lasten nabor komponent, ki sem jih kasneje posplošil in premaknil v 3iGUI. Prav tako je neodvisno od mojega dela na ogrodju 3iGUI nastal vzporedni projekt, na katerem je sodelavec razvil „višjeležeče“ ogrodje delno osnovano na ogrodju 3iGUI. V tem ogrodju so se pojavile rešitve, kot je na primer uporaba zunanjih nastavitev komponent. Veliko rešitev, ki so se izkazale v praksi, sem kasneje prevzel in premaknil v 3iGUI.

V času razvoja sem naredil veliko sprememb na ogrodju 3iGUI. Nekajkrat sem spremenil videz komponent in osrednjih vsebnikov ogrodja, pogosto uporabljenim komponentam sem dodajal nove zmožnosti in nastavitve, število različnih modelov se je z večanjem števila aplikacij povečalo itd. Modul application_state_manager sem večkrat spremenil in opitimiziral. Pojavilo se je tudi nekaj ključnih odločitev, ki so vplivale na ozadje delovanja aplikacij. Prva je bila odločitev za **enonitne instance** aplikacij. Na začetku je ogrodje dopuščalo več vzporednih zahtevkov na eno instanco aplikacije oz. instanco stanja. V praksi se je izkazalo, da taka zasnova povzroča težave. Primerov,

ko bi uporabnik v istem oknu brskalnika vzporedno sprožil večje število zahtevkov, skoraj ni bilo. Take primere je mogoče rešiti tudi drugače, zato sem se odločil za zaklepanje stanj in s tem poenostavil razvoj aplikacij. Tako ni bilo več potrebno razmišljati o sinhronizaciji stanj in kode aplikacij, ki lahko vsebuje zahrbtne napake. Druga odločitev je bila uporaba tehnike **AJAX** na tak način, da je z vidika razvoja aplikacij uporaba le-te **nevidna**. Čeprav je bilo ogrodje 3iGUI z vidika razvoja zelo „dinamično“, je bistvo zasnove ostalo nespremenjeno in združljivo s starejšimi različicami aplikacij.

Poglavje 5

Testiranje in prehod v produkcijo

Med razvojem ogrodja sem sproti testiral kjučne dele kode. Pri tem sem si pomagal z aplikacijami, ki sem jih sproti razvijal v ta namen. Del testnih aplikacij je namenjen zgolj testiranju delov ogrodja, druge aplikacije so hkrati tudi **učne aplikacije**, ki vsebujejo najbolj pogoste vzorce uporabe komponent. Kodo učnih aplikacij sem poleg običajne dokumentacije JavaDoc opremil z dodatnimi komentarji uporabljenih vzorcev. V času razvoja ogrodja sem pripravil devet učnih aplikacij. Te zajemajo predvsem seznam vseh obstoječih komponent s primeri uporabe in povezavami na dokumentacijo, uporabo najpomembnejših vsebnikov, kompleksnih komponent in različnih modelov, primere različnih implementacij poslušalcev itd.

Moduli 3iOS-ovega jedra, za katere sem pripravil namenske testne aplikacije, so bili predvsem 3iFS, application_state_manager, db_connection_pool in log_manager. Koda modulov je bila zahtevnejša od preostale kode zaradi konkurenčnih dostopov večjega števila niti in zahtev po visoki zmogljivosti modulov in prepustnosti podatkov. V ta namen sem za testiranje vsakega modula pripravil enostavno aplikacijo, ki je zgradila večje število niti in simulirala konkurenčno uporabo omenjenih modulov. Pri preverjanju pravilnosti delovanja modulov mi je bil v veliko pomoč tudi vtičnik razvojnega okolja FindBugs.

5.1 Zmogljivost ogrodja

Pri testiranju zmogljivosti ogrodja in aplikacij sem največ časa posvetil količini prenesenih podatkov in porabi pomnilnika na strežniku. **Količina prenesenih podakov** lahko niha od kilobajta na zahtevek za enostavne poglede do 100 kilobajtov na zahtevek za zelo obsežne poglede. Za vse „besedilne“ podatke sem predvidel stiskanje z metodo gzip, ki jo podpirajo brskalniki. Izkazalo se je, da je stiskanje HTML-ja zelo učinkovita rešitev. Določene poglede pa smo zaradi kompleksnosti razbili na več delov (na primer z uporabo zavihkov).

Največ časa pri raziskovanju mej zmogljivosti ogrodja sem posvetil merjenju **porabe pomnilnika**. Po tem, ko sem odgovornost za vzdrževanje stanja aplikacij prenesel na strežnik, se je poraba pomnilnika povečala. Modul application_state_manager je največji porabnik pomnilnika na strežniku. Za pomoč pri raziskovanju stanj aplikacij sem razvil sistemsko aplikacijo, ki omogoča pregled stanj in izračuna njihovo okvirno porabo pomnilnika. Porabo pomnilnika stanj sem ocenil na podlagi meritev. Postopek merjenja je bil sledeč:

- Počisti pomnilnik z nekaj zaporednimi klici smetarja (angl. garbage collector) in zaključevanja (angl. finalization) objektov. Kličeta se statični metodi `System.gc()` in `System.runFinalization()`.
- Izmeri trenutno količino porabljenega pomnilnika.
- Naloži stanje, za katerega želimo izmeriti porabo pomnilnika.
- Pomnoži stanje v pomnilniku, da dobiš 10 kopij stanja.
- Ponovi klice smetarja in zaključevanja objektov.
- Izmeri trenutno količino porabljenega pomnilnika.
- Velikost posameznega stanja izračunaj kot razliko med obema porabama pomnilnika deljeno z 10.
- Premakni (shrani) vsa stanja iz pomnilnika na disk.
- Primerjaj velikost stanja v pomnilniku z velikostjo datoteke stanja na disku.

Velikost stanja je seveda zelo odvisna od posamezne aplikacije in načina njene uporabe. Novejše aplikacije, ki jih razvijamo, imajo lastnost, da uporabniku omogočajo poljubno odpiranje novih pogledov (dokumentov) v zavihkih. Skrajni primer predstavlja aplikacija, ki lahko na enem samem zavihku vsebuje več kot 400 komponent (tako zahtevne poglede navadno dodatno razbijemo na posamezne (pod)zavihke). Velikost stanja v pomnilniku take aplikacije lahko niha od nekaj 10 kilobajtov do nekaj megabajtov. Pri testiranju različno velikih stanj sem z meritvami ugotovil okvirno razmerje med velikostjo stanja aplikacije v pomnilniku in velikostjo datoteke stanja na disku. Stanje v pomnilniku zasede približno 5-krat več pomnilnika v primerjavi z velikostjo na disku. Testiral sem tudi stiskanje (gzip) vsebine datotek s stanji in ugotovil, da je mogoče stanja precej stisniti. V večini primerov se je po stiskanju velikost datoteke zmanjšala pod četrtino prvotne velikosti. Ker pa so datoteke stanj v večini primerov zelo majhne, stiskanja stanj na disku v končni različici modula application_state_manager nisem uporabil. Razmerje stanja v pomnilniku in na disku je zelo uporabno za hitro ocenjevanje porabe pomnilnika. S hitrim pregledom mape shranjenih stanj lahko ocenimo, katere aplikacije imajo kritično porabo pomnilnika. Na podlagi takih ocen smo našli kar nekaj nedoslednosti oziroma pomnilniško potratne kode.

5.2 Diagnostika komponent in drugih objektov ogrodja

Pri razvoju aplikacij je potrebno upoštevati določena pravila. Pravila oziroma primerni vzorci kode velikokrat niso razvidni iz programskega vmesnika ogrodja, temveč jih je potrebno spoznati prek dokumentacije in učnih primerov kode. Kjub temu, da ogrodje ne more uveljaviti vseh pravil prek svojega programskega vmesnika, pa lahko uporabo nepravilnih vzorcev kode preverja v stvarnem času. Ogrodje opozarja na sumljive vzorce uporabe prek sistemskih dnevniških datotek. Primer je pravilo, ki določa, da lahko posamezno komponento dodamo v hierarhijo samo enkrat. V opisanem primeru ogrodje opozori, če je bila neka komponenta dodana v hierarhijo več kot enkrat, če je komponenta zamenjala svoj vsebnik, če se je prva inicializacija komponente izvedla več kot enkrat ipd. Točk, v katerih ogrodje preverja veljavnost uporabljenih vzorcev, je okoli 100, dodajam pa jih sproti glede na pojavljanje novih nepravilnih vzorcev uporabe oziroma napak, ki se pokažejo v praksi. Največ preverjanj opravi 3iGUI, nekaj pomembnejših pa opravijo tudi moduli 3iOS-ovega jedra. Slednji imajo zelo pomembno nalogo preverjanja pravilnega zapiranja referenc

na zunanje vire podatkov. 3iFS na primer preverja zapiranje vseh uporabljenih vhodnih in izhodnih podatkovnih tokov vsebovanih datotek. 3iOS pa preverja pravilno zapiranje povezav na podatkovne baze in zapiranje odprtih sej zalednega ogrodja za preslikovanje med poizvedbami SQL in objekti (iBatis). Ograjevanje vseh storitev, na katerem temelji 3iOS, se je v primeru testiranja in iskanja vzrokov napak izkazalo kot zelo učinkovito. Ker 3iOS ogradi vse dostope do zunanjih virov, je mogoče med aplikativno kodo in zaledna ogrodja vriniti kodo za razhroščevanje. Primer takega vrivanja je ograjevanje povezav na podatkovno bazo, ki omogoča sledenje vsem poizvedbam in vrstnemu redu klicev metod na določeni povezavi v stvarnem času. Tak način dela se je v določenih primerih izkazal za hitrejšega in učinkovitejšega v primerjavi z uporabo običajnega razhroščevalnika v Eclipsu.

3iOS se lahko izvaja v produkcijskem ali razvojnem načinu. Diagnostika se izvaja v obeh načinih, vendar z nekaj razlikami. V razvojnem načinu je diagnostika obsežnejša, poleg tega ogrodje shranjuje v stanja komponent dodatne metapodatke, ki so namenjeni razvijalcem pri morebitnem razhroščevanju. V produkciji se del diagnostike, ki bi bil preveč potraten z viri strežnika, izklopi. Kjub temu ogrodje vedno izvaja najpomembnejše točke diagnostike, saj lahko na nepravilno delovanje komponent vplivajo tudi produkcijski podatki, ki jih v razvojnem in testnem okolju ni. Zaradi diagnostike je stanje aplikacij, ki se izvajajo v razvojnem načinu, večje za približno 20 do 30 odstotkov glede na velikost stanja istih aplikacij v produkciji.

5.3 Produkcijsko okolje, kopija produkcije in stik aplikacij z okolico

Preden aplikacijo namestimo v produkcijo, opravimo več korakov:

1. **Razvoj aplikcije** in sprotno testiranje poteka na razvojnih računalnikih.

2. **Prva namestitev pred produkcijo** se izvede na testnem strežniku. Na tem strežniku se izvedejo dodatni testi, ki zajemajo predvsem ponovitev predvidenih scenarijev uporabe. Pri tem koraku navadno sodeluje več oseb.

3. **Druga namestitev pred produkcijo** se izvede na poseben strežnik, ki predstavlja kopijo produkcije. Ta korak služi kot priprava na končno namestitev in priprava rezervne kopije strežnika.

4. **Končna namestitev** je ponovitev prejšnjega koraka, vendar na produkcijskih strežnikih. Navadno sledi še nekaj osnovnih testov, s katerimi preverimo nadgradnjo sistema.

Za vzdrževanje strežnika in premikanje namestitev 3iOS-a med strežniki navadno skrbi oseba, ki ne pozna vseh podrobnosti aplikacij in ogrodja. 3iOS sem moral prilagoditi takemu načinu dela ter rešiti navezavo na zunanje storitve oziroma **stik z okolico**. Namestitev spletnega strežnika s 3iOS-om zato predvideva kopiranje vsebine ene same mape na strežniku in nastavitev parametrov v eni datoteki. Gre za nastavitveno datoteko, ki vsebuje parametre povezav na podatkovne baze, spletne storitve in osnovne parametre izvajanja 3iOS-a. Pomembnejši parameter je informacija, ali je strežnik dejansko nameščen v produkciji. Določene storitve sistema (na primer določena ciklična opravila) se zaženejo samo v testnem okolju ali končni produkciji, nikoli pa v primeru kopije produkcije. V vseh primerih, kjer je bilo potrebno podpreti obstoječo organizacijo dela v podjetju, sem to storil s prilagoditvami ogrodja, aplikacij pa zaradi takih sprememb ni bilo potrebno posebej prilagajati ali spreminjati.

Poglavje 6

Primerjava ogrodij

Primerjavo ogrodij GWT, Wicket in 3iGUI sem osredotočil na pregled razlik in podobnosti arhitekture ogrodij, pristope k razvoju aplikacij in tudi na možne povezave med njimi. V primerjavi sem upošteval predvsem značilnosti in zmožnosti osnovnih namestitev ogrodij brez dodatnih „zunanjih“ razširitev. Teh je za GWT in Wicket sicer veliko, kar seveda lahko štejemo v prid obema ogrodjema.

6.1 Razvojno okolje in priprava prve aplikacije

Google je za osnovno razvojno okolje izbral Eclipse. Čeprav **GWT** vsebuje orodja, ki jih lahko uporabljamo neposredno iz orodne vrstice, je uporaba Eclipsa enostavnejša. Uporaba GWT-jevih orodij oziroma vtičnikov je obvezna. GWT uporablja zanimiv pristop k razvoju, kjer lahko kodo sproti prevajamo z običajnim javanskim prevajalnikom. Poleg tega GWT-jev vtičnik skrbi za lastne posebnosti in pravila pri razvoju aplikacij ter nas dodatno opozarja na napake oziroma nedoslednosti. Primer takega opozorila je poskus uporabe razreda iz standardne javanske knjižnice, ki ga GWT-jeva emulacija knjižnice ne podpira. GWT podpira izvajanje aplikacij v dveh načinih, in sicer v razvojnem načinu (angl. development mode) in produkcijskem načinu (angl. production mode). Dostop do aplikacij v razvoju omogoča prek javanskega strežnika, ki je del GWT-jeve orodjarne. **Razvojni način** je primeren za hitro preverjanje rezultatov kodiranja in za razhroščevanje. Ko pripravljamo aplikacijo za produkcijo ali ko želimo preveriti delovanje dokončno prevedene aplikacije, uporabimo GWT-jev prevajalnik, ki javansko kodo prevede v JavaScript in pripravi vse ostale datoteke, ki so potrebne za izvajanje aplikacije. Tako pripravljeno aplikacijo izvajamo v **produkcijskem načinu**. V njem

lahko preverimo končno delovanje aplikacije (možna so predvsem odstopanja glede na razvojni način, ki se kažejo v razlikah hitrosti nalaganja in izvajanja aplikacije ter razlikah videza v različnih brskalnikih). **Wicket** in **3iGUI** sta ogrodji napisani v Javi in namenjeni izvajanju v Javi. Pri uporabi obeh ogrodij lahko zato uporabimo poljubno razvojno okolje za ta programski jezik. Tudi Wicket in 3iGUI uporabljata dva načina izvajanja, ki ločujeta med razvojem in produkcijo. V obeh primerih je razvojni način prilagojen razvijalcem in nudi možnost dodatne diagnostike in hitrega osveževanja sprememb vsebine besedilnih predlog. Produkcijski način tipično poskrbi za optimalno izvajanje aplikacij in zakrivanje informacij, ki niso namenjene končnim uporabnikom.

Za delo z vsemi tremi ogrodji je bilo potrebno pripraviti oziroma nastaviti primeren **strežnik**. Za potrebe dela z GWT-jem sem namestil privzet Googlov strežnik (Google App Engine), ki ga pri določanju novega projekta ponudi vtičnik za Eclipse. Za Wicket in 3iGUI sem nastavil ločeno instanco strežnika Jetty. Pri tem naj omenim, da gre tudi pri Googlovem strežniku zgolj za prirejeno različico strežnika Jetty. Ker so aplikacije razvite z GWT-jem v osnovi neodvisne od strežnika, je potrebno posegati v nastavitve le-tega samo v primeru uporabe storitev (RPC) na strežniku. Sicer razvojno okolje samo poskrbi za prenos odjemalčevih datotek v pravo mapo na strežniku. V primeru ogrodja Wicket je potrebno za vsako novo aplikacijo pripraviti oziroma spremeniti namestitev na strežniku. Pri tem je pomembna predvsem standardna datoteka `web.xml`. V njej povežemo izvajalni del ogrodja z lastno aplikacijo oziroma aplikacijami. Namestitev ogrodja 3iGUI zahteva predhodno namestitev 3iOS-a. Ta je standardna in vedno enaka, tako da sem uporabil že pripravljeno datoteko `web.xml` in korensko mapo za 3iFS. Namestitev aplikacije je potrebno opraviti nekje v mapi `/[PROGRAM_FILES]/`. Navadno gre za kopiranje neke že obstoječe mape in prirejanje ene ali dveh datotek.

Večina literature ogrodij posveti prvi primer kode aplikaciji „Pozdravljen svet“ (angl. Hello world). Taka aplikacija služi enostavnemu pregledu korakov, ki privedejo od nastavitve razvojnega okolja prek prvega primera izvorne kode do izvajanja aplikacije. Z vidika take „začetniške“ aplikacije omogoča **GWT** najhitrejši začetek dela, saj okolje ob začetku novega projekta pripravi delujočo aplikacijo skupaj s primerom implementacije enostavne strežniške storitve. Za začetnika je to zagotovo dobrodošlo, saj lahko prične s poskušanjem in spreminjanjem kode preden je predelal dokumentacijo. Google je za GWT pripravil pregledno in kvalitetno javno dostopno dokumentacijo in navodila za začetnike. Tudi sicer je GWT dobro zastopan na spletu (forumi in blogi) in v „klasični“

literaturi, saj je o njem napisanih kar nekaj knjig. Zaradi hitrega razvoja ogrodja je potrebno nekaj pazljivosti, saj je kar nekaj vsebine zastarele. Za pripravo začetne aplikacije z ogrodjem **Wicket** potrebujemo nekaj več dela in osnovno poznavanje vsebnika servetov in vsebine datoteke `web.xml`. Pomagamo si lahko z obsežno dokumentacijo, knjigami in ostalimi viri, kjer je tudi Wicket dobro opisan. Priprava prve aplikacije v **3iGUI** zahteva nekaj več dela od ostalih dveh ogrodij. Pomembna razlika med ogrodjem 3iGUI in ostalima ogrodjema je dejstvo, da je 3iGUI le del 3iOS-a in da z namestitvijo aplikacije le-ta dobi dostop do vseh storitev, ki jih nudi to ogrodje. To narekuje tudi organizacijo datotek, s katero se je potrebno vsaj delno seznaniti pred namestitvijo aplikacije. Literatura je seveda omejena na lastne vire. Razvijalci imajo v pomoč na voljo primere kode v obliki učnih aplikacij, predstavitve ogrodja z obširno dokumentacijo arhitekture in vzorci pravilne uporabe pomembnejših delov ogrodja.

Količina programske kode, ki je potrebna za vzpostavitev enostavne začetne aplikacije, je nasprotno od namestitve pri vseh ogrodjih približno enaka. V vseh primerih je potrebno pripraviti določen razred, ki predstavlja začetno točko izvajanja aplikacije. Navadno potrebujemo še navezavo na predlogo HTML. GWT potrebuje vsaj enostaven dokument HTML, v katerega lahko doda komponente, Wicket potrebuje predlogo HTML, ki je povezana z domačo stranjo aplikacije, 3iGUI potrebuje izvajalno formalno predlogo, v kateri je določen razred spletne aplikacije.

6.2 Izvajanje aplikacij in njihovo stanje

Spletne aplikacije, ki so razvite z **GWT-jem**, se izvajajo v brskalniku. Strežnik uporabljajo za oddaljene klice procedur, prek katerih navadno dostopajo do podatkov oz. le-te spreminjajo. GWT olajša klice storitev na (javanskem) strežniku z zakrivanjem podrobnosti pogovora odjemalec-strežnik za enostavnimi vmesniki in z lastno izvedbo serializacije objektov, ki se prenašajo med odjemalcem in strežnikom. Klici strežnika so vedno asinhroni, kar lahko zahteva nekaj privajanja, če s takimi klici še nismo imeli opravka. Stanje aplikacije je na odjemalcu in zanj ni potrebno posebej skrbeti. Zaradi omenjenih značilnosti so take aplikacije navadno zelo odzivne, poleg tega pa so manj obremenjujoče za strežnik in omrežje. Stanje na odjemalcu omogoča tudi uporabo strežniških storitev brez stanja in posledično skalabilnost celotnega sistema. Če razvijamo aplikacije za zelo široko občinstvo, se lahko uporaba GWT-ja

zelo hitro obrestuje.

Ogrodji **Wicket** in **3iGUI** sta si glede samega izvajanja aplikacij zelo podobni. Aplikacije se izvajajo na strežniku, obe ogrodji pa poskrbita za bolj ali manj „prikrito“ vzdrževanje stanja. Pri tem se osnovni implementaciji[5] ogrodij razlikujeta glede na način in mesto shranjevanja stanja. Wicket shranjuje stanje aplikacije v aktualni seji na strežniku. Prednost tega pristopa je v tem, da je vzdrževanje stanja v seji prepuščeno strežniku, ki upravlja z objekti v seji in lahko le-te shrani tudi na disk ali pa pošlje drugim strežnikom (vozliščem) v gruči. Slabost takega pristopa pa je povečana poraba pomnilnika, ki je neposredno odvisna od trajanja seje. Predolge seje bodo po nepotrebnem zasedale del pomnilnika, prekratke seje pa lahko povzročijo prezgodnjo izgubo stanj aplikacij. Wicket sicer omogoča tudi druge načine shranjevanja stanja, med katere spada tudi shranjevanje v začasno mapo na disku strežnika. 3iGUI vzdržuje stanje aplikacije s pomočjo 3iOS-a. Stanje se ne shranjuje v seji, temveč v ločenem vsebniku, ki stanje po potrebi premakne iz pomnilnika v mapo modula 3iFS. Stanja ločuje s pomočjo enolične oznake, ki je vezana na uporabniško ime in okno brskalnika. Prednost tega pristopa je razbremenitev seje, ki se izkaže predvsem v primerih, ko uporabnik dlje časa na isti seji poganja večje število aplikacij. Slabost takega pristopa je, da je potrebno dobro določiti čase trajanja stanja v pomnilniku, preden se to shrani v 3iFS. Prekratki časi lahko povzročijo večji pretok podatkov med diskom in pomnilnikom in s tem obremenitev strežnika, predolgi časi pa podobno kot v primeru uporabe seje povzročijo daljšo zasedenost pomnilnika.

V primeru GWT-jeve aplikacije je vsebina **stanja** kar množica vseh objektov, ki se vzdržujejo v pomnilniku v času izvajanja aplikacije. Gre predvsem za vsebnike, komponente, poslušalce, modele ipd. Za vse omenjene objekte velja, da so to „običajni“ objekti, ki ne zahtevajo sprememb ali prilagoditev zato, da so lahko del stanja. Aplikacija izgubi stanje, ko uporabnik zapusti spletno stran, v kateri „domuje“ aplikacija, oziroma ko zapre okno brskalnika.

Stanje aplikacije razvite z ogrodjem Wicket je množica objektov, ki implementirajo vmesnik `Serializable`. Wicket v stanju aplikacije shrani celotno hierarhijo spletnih strani, vsebnikov na teh straneh, komponent na vsebnikih in modelov komponent. Poleg tega Wicket shranjuje tudi zgodovino strani in prehodov med njimi. Stanje aplikacije se izgubi ob zaključku seje, določeni deli

[5]Gre za privzeto obnašanje ogrodij. Vsa tri omenjena ogrodja je sicer možno ponastaviti oziroma prirediti lastnim potrebam in s tem spremeniti njihovo privzeto obnašanje.

stanja pa se lahko razveljavijo tudi med izvajanjem aplikacije, če Wicket ugotovi, da je stanje strani zastarelo. Prednost take vsebine stanja je predvsem enostavnost razvoja, saj se vsi objekti in relacije med njimi vedno ohranijo v popolnosti. Slabost takega pristopa je lahko večje stanje aplikacije, če z dodatnimi ukrepi ne poskrbimo za obvladovanje velikosti stanja.

Stanje aplikacije razvite z ogrodjem 3iGUI vsebuje le del vsebine komponent in vsebnikov ter del vsebine modelov in drugih objektov, ki nastanejo med izvajanjem aplikacije in jih je potrebno vzdrževati v stanju. Za razliko ogrodja Wicket 3iGUI ne shranjuje celotne hierarhije komponent, temveč si v stanju zapomni samo njihove identifikacijske oznake, iz katerih zna ogrodje vsakokrat vzpostaviti hierarhijo objektov, ki je enaka hierarhiji ob zadnjem zahtevku HTTP. Objekti, ki so del stanja, morajo implementirati vmesnik `Externalizable`, kar ima podoben učinek kot v primeru ogrodja Wicket. Poleg tega lahko objekti aktivno sodelujejo v življenjskem ciklu aplikacije, če njihovi razredi implementirajo določen vmesnik. Na tak način lahko objekti sami določajo, kateri podatki bodo del stanja in kateri ne. Prednost tega načina so bolj očitni deli kode, ki vplivajo na velikost stanja in zato tudi lažje obvladovanje le-tega. Slabost tega pristopa pa je dodatno delo, ki ga mora v določenih primerih opraviti razvijalec.

Ena od pomembnih razlik med GWT-jem in ostalima dvema ogrodjema je, da GWT ni neposredno vezan na vzorec zahtevek-odgovor, saj se ta pojavi zgolj v primeru klicev strežnika. Ogrodji Wicket in 3iGUI sta vezani na vzorec zahtevek-odgovor, poskrbita pa, da je ta vzorec „neviden“ pri razvoju aplikacij.

6.3 Razvoj uporabniških vmesnikov

Ogrodje Wicket je bolj „spletno“ usmerjeno kot ostali dve ogrodji. Tukaj je mišljena predvsem zgradba tipične aplikacije, ki uporablja „spletne pojme“, kot so spletna stran, spletni obrazec in prehod med stranmi. Aplikacijo torej sestavlja množica spletnih strani, ki so predstavljene z javanskimi objekti. Gre za korenske vsebnike, v katere lahko dodamo ostale komponente. Wicket uporablja zanimivo prepletanje pristopov k razvoju, ki so značilni za splet in namizje. Za razliko od ogrodja Wicket, pa je razvoj aplikacij z ostalima ogrodjema še bolj podoben razvoju klasičnih namiznih aplikacij. GWT in 3iGUI temeljita na „klasičnih“ vsebnikih in komponentah. GWT dovolj dobro zakrije navezavo na brskalnik, ki je podlaga za izvajanje aplikacij, a se razvijalec

prej ali slej sreča tudi z deli kode, kjer je ta navezava očitna. Omogočena je na primer možnost neposrednega dela z DOM-om. Prav tako podrobnejši pogled implementacije komponent hitro razkrije, da so te v resnici zgolj „lupina“, ki ograjuje DOM. Implementacija komponent v ogrodjih Wicket in 3iGUI nasprotno pri podrobnen ogledu razkrije tipične vzorce, za katerimi se „skriva“ prilagajanje vzorcu zahtevek-odgovor in mehanizmom za vzdrževanje stanja.

Vzorci uporabe komponent so v primeru vseh treh ogrodij podobni. Wicket in 3iGUI temeljita na vzorcu MVC. Wicket je v tem pogledu vzoren primer uporabe tega vzorca. Implementacija modelov in njihova povezava s pogledi (komponentami) je enostavna. Pri prenosu podatka med modelom in pogledom Wicket samodejno poskrbi za primerno pretvorbo podatka in njegovo validacijo, ko podatek zapisuje nazaj v model. Pri tem omogoča tudi dodajanje lastnih validatorjev in pretvornikov podatkov. Wicket omogoča pisanje kratke in jedrnate kode, ki pa lahko zakriva tudi nekaj pasti. Tak primer je uporaba modelov, ki omogočajo samodejno preslikovanje podatkov med komponentami in domenskimi objekti, kjer ogrodje uporabi javansko introspekcijo (Java reflection). S tem omogoči preslikovanje celotne hierarhije komponent v hierhijo modelov v eni ali nekaj vrsticah kode. Prednost takega pristopa je velik učinek, ki ga dosežemo z malo kode. Slabost pa je v tem, da lahko taka koda ponudi premalo informacij za razumevanje „ožičenja“ komponent z modeli. Tako lahko na primer z enostavno spremembo imena metode domenskega objekta povzročimo eno ali več napak, ki jih prevajalnik ne zazna in se pokažejo šele med izvajanjem aplikacije, kar zahteva dodatno testiranje.

Pri pregledu in delu z ogrodjem Wicket sem opazil zanimivo razliko z ogrodjem 3iGUI. V ogrodju 3iGUI je ponekod potrebno nekaj več kode za to, da dosežemo željeno obnašanje komponente ali drugega objekta. V primeru dela z Wicketom pa se je dogajalo, da je ogrodje že privzeto omogočilo določeno obnašanje komponente; če takega obnašanja nisem želel, sem ga moral onemogočiti. Takih primerov je več, v nadaljevanju sem podal zgolj bolj očitne:

- Wicket omogoča obvladovanje velikosti stanja z uporabo posebnih modelov, ki so namenjeni razdruževanju (angl. detach) pomnilniško potratnih objektov od stanja aplikacije. Če takih modelov ne uporabljamo, bodo vsi podatki privzeto del stanja. 3iGUI predvideva, da predvsem kompleksni modeli sami določijo kateri podatki bodo del stanja (vsi ostali se privzeto zavržejo).

- Wicket izvaja samodejno validacijo vsebine komponent, razen če mu tega

posebej ne preprečimo. 3iGUI izvaja validacijo na zahtevo.

- Wicket izvaja samodejno pretvorbo objektov med pogledi in modeli. 3iGUI prepusti pretvorbo samim modelom.
- Wicket med drugim omogoča že omenjeno preslikovanje podatkov prek javanske introspekcije. 3iGUI uporablja javansko introspekcijo samo znotraj ogrodja, ne predvideva pa njene uporabe v aplikativni kodi. Tukaj naj še poudarim, da je pri obeh ogrodij možno uporabiti iste pristope in da gre pri opisanih razlikah bolj za razlike v „filozofiji“ uporabe ogrodij in ne za tehnične omejitve le-teh.

Ogrodje GWT temelji na vzorcu MVP. Ker ta vzorec predvideva predstavitelja, ki popolnoma razmeji model od pogleda, komponente uporabniškega vmesnika privzeto ne omogočajo neposredne navezave na modele. Uporaba komponent je v tem pogledu primerljiva z uporabo implicitnih modelov na komponentah ostalih dveh ogrodij in po tej plati spominja bolj na ogrodje AWT kot na Swing. Podatek neposredno dodamo v komponento ali ga iz nje pridobimo s klicem metode. Pri delu s podatki je potrebno vedno imeti v mislih dejstvo, da se aplikacija izvaja na odjemalcu, zato je potrebno predvideti tudi sodelovanje s strežnikom. Ločevanje kode na aplikacijo v brskalniku in storitve na strežniku je ena glavnih razlik pri razvoju, če ga primerjamo z ostalima ogrodjema. Privzeti nabor komponent uporabniškega vmesnika v GWT-ju vsebuje večino osnovnih in splošno uporabnih komponent, ki jih od takega ogrodja pričakujemo. Nabor ni tako obsežen kot pri drugih dveh ogrodjih, a je zanj na voljo veliko „zunanjih“ rešitev, ki razširijo nabor zmožnosti tega ogrodja. Zanimivo je, da med njimi najdemo tudi rešitve, kot so uporaba modela MVC (neposredna povezava komponent z modeli) in validacija vsebine komponent, saj GWT-jeve komponente v osnovi nimajo predvidenih validatorjev. Wicket ima prav tako veliko dodatnih rešitev, s katerimi lahko obogatimo osnovni nabor komponent. 3iGUI nasprotno temelji samo na lastnem (osnovnem) naboru komponent, ki se je s sprotnim razvojem aplikacij postopoma širil in se prilagajal izključno potrebam naših aplikacij. Po številu komponent je ta nabor obsežnejši, čeprav ne zajema popolnoma vseh komponent, ki jih privzeto ponujata ostali dve ogrodji. Predvsem so te komponente v določenih primerih bolj ozko specializirane, kar je seveda pogojeno z ozko usmerjenim „občinstvom“ ogrodja 3iGUI.

Še najbolj se ogrodja razlikujejo pri uporabi tehnike **AJAX**. Wicket nudi dobro podporo tej tehniki, tako da zakrije kar nekaj podrobnosti, predvsem

tistih, ki zadevajo kodo na odjemalcu. Namesto običajnih povezav, lahko na primer uporabimo posebne povezave AJAX, ki ob kliku sprožijo zahtevek AJAX in od strežnika dobijo le del pogleda, ki ga je potrebno ažurirati, ne pa celotne spletne strani. Na strežniku moramo pripraviti primernega poslušalca, ki bo obdelal zahtevek AJAX in kot odgovor na zahtevek, določil komponente, ki naj se ažurirajo na odjemalcu. Komponente, ki se lahko (delno) ažurirajo na odjemalcu, moramo tudi posebej nastaviti, tako da se njihova enolična oznaka uporabi tudi v brskalniku, saj jo Wicket potrebuje za iskanje in ažuriranje primernega vozlišča v DOM-u. Uporaba tehnike AJAX je v ogrodju 3iGUI še bolj zakrita. Ogrodje ne vsebuje posebnih nastavitev komponent ali poslušalcev, ki bi bili vezani na uporabo AJAX-a. Uporaba te tehnike je popolnoma odvisna od konteksta izrisa. Prednost takega pristopa je dejstvo, da se razvijalcem ni potrebno ukvarjati s posebnimi komponentami in dogodki in da na ažuriranje pogledov komponent lahko pozabijo. Slabost takega pristopa pa je, da je v določenih primerih izvajanje aplikacij zaradi tega manj optimalno, ko bi bilo, če bi kontrolo nad uporabo AJAX-a prepustili razvijalcem. Aplikacije razvite z ogrodjem GWT so privzeto aplikacije AJAX, kar pomeni, da se njihovi pogledi navadno osvežujejo po delih, za dostop do strežnika pa se uporabljajo asinhroni klici. Razvoj aplikacij bolj kot na „klasični“ AJAX spominja na „klasični“ razvoj odjemalec-strežnik. To pomeni, da se tipični zahtevek AJAX zakrije za običajnim postopkom: zahtevaj podatke, preberi podatke, nastavi novo vsebino komponent.

6.4 Povezovanje ogrodij

Iz osnovne primerjave ogrodij je razvidno, da se ta razlikujejo, čeprav je z njimi mogoče reševati podobne probleme. Vsako ogrodje se tipično izkaže na določeni množici aplikacij, zato me je bolj zanimalo ali lahko s povezovanjem ogrodij razširim to množico ter na koliko načinov in kako lahko povežem ogrodja, da bi lahko hkrati izkoristil kar največ njihovih prednosti. V nadaljevanju sem opisal nekaj možnosti povezav med posameznimi ogrodji.

Osnovno povezovanje GWT-ja s 3iOS-om je enostavno. Če želimo razvijati aplikacije v GWT-ju, lahko strežniški del kode brez težav povežemo s 3iOS-om. Pri tem na strani strežnika pridobimo predvsem storitve, ki jih nudi 3iOS, kot so možnost uporabe sistema uporabnikov z zaščito dostopov do aplikacij, dostop do podatkov prek modula 3iFS, uporabo obstoječih storitev pri povezovanju s podatkovnimi bazami in možnost uporabe predložnega stroja

za pripravo vstopnih strani za GWT. Vse statične datoteke, ki so rezultat razvoja z GWT-jem, lahko zaščitimo pred nepooblaščenimi dostopi, podobno velja tudi za storitve, ki se izvajajo na strežniku. Pri opisanem načinu povezovanja ni potrebno prilagajati kode nobenega od ogrodij, saj je vse možno izvesti samo s prilagoditvijo datoteke `web.xml` in z uporabo javanskih paketov, ki so potrebni za izvajanje GWT-jevih strežniških storitev. Enostavnost takega načina povezovanja je posledica arhitekture aplikacij razvitih z GWT-jem in dejstva, da 3iOS privzeto omogoča izvajanje in kontrolo nad izvajanjem javanskih servletov. Aktulna koda aplikacij v takem primeru ni enostavno prenosljiva iz ogrodja 3iGUI v GWT. Tak način povezovanja je zato smiseln za aplikacije, pri katerih bi prednosti GWT-ja upravičile dodatno uvajanje v to ogrodje ali celo (ponovni) razvoj lastnih namenskih ogrodij za GWT, ki so bila že razvita na podlagi ogrodja 3iGUI.

Drugi način povezovanja je uporaba GWT-ja v ogrodju 3iGUI. Gre za možnost priprave novega konteksta izrisa z uporabo GWT-jevega prevajalnika in vmesnika za DOM. V tem primeru se del konteksta izrisa, ki ga sedaj opravi strežniška koda ogrodja 3iGUI, prenese v brskalnik. Tega načina povezovanja ogrodij sicer nisem preizkusil, pri preverjanju koncepta pa sem se naslonil na eno od naših starejših ogrodij, ki za kontekst izrisa uporablja kar odjemalca napisanega v Javi (Java Appet), in ogrodje AWT. Ker gre za zelo podobno arhitekturo, sem ta način povezovanja uvrstil med najbolj zanimive kandidate. Tukaj naj še omenim, da podobno arhitekturo izkoriščajo tudi nekatera odprtokodna ogrodja, med katere spadata na primer Vaadin in Echo 3. Prednost takega povezovanja je večja odzivnost uporabniškega vmesnika, kjer se aplikacija še vedno izvaja na strežniku, kar navadno pomeni večjo varnost. Izvajanje na strežniku omogoča tudi uporabo celotnega nabora razredov, ki so del standardne javanske knjižnice. Velika prednost z vidika našega podjetja pa je, da v tem primeru ni potrebno prilagajati aplikacij, temveč le enega od slojev ogrodja. Slabost take povezave so predvsem dodatni stroški razvoja novega konteksta izrisa in večja kompleksnost ogrodja.

Pri povezovanju z ogrodjem Wicket sem poskušal povezati ogrodje samo z ogrodjem 3iOS kot tudi s predložnim strojem v 3iOS-u. Slednji način povezave je bil le delno uspešen, saj Wicket v določenih primerih uporablja objekte za zahtevek in odgovor tako, da povzročajo težave predloženemu stroju. Omenjene objekte sem moral dodatno ograditi z uporabo vzorca „dekorator“ in tako poskušati obdržati kontrolo nad njihovo vsebino, ki jo potrebuje predložni stroj. Z nekaj truda mi je uspelo „prepričati“ Wicket, da je prevzel vlogo ge-

neratorja vsebine v predloženem stroju, a sem za to moral delno prirediti tudi nastavitve aplikacije razvite z ogrodjem Wicket.

Drugi način povezovanja z Wicketom, torej neposredno s 3iOS-om brez povezave s predložnim strojem, je bil lažji. Čeprav 3iOS omogoča izvajanje servletov in je to zadostovalo za povezavo z GWT-jevo strežniško kodo, sem moral v primeru povezave z ogrodjem Wicket pripraviti prototip modula, ki je dodatno poskrbel za posebnosti tega ogrodja. Dela ni bilo veliko in za razliko od povezovanja s predložnim strojem, sem rezultat dosegel hitreje. Prednost povezave z Wicketom je predvsem možnost razvoja bolj „spletnih“ aplikacij, za katere 3iGUI ni primeren, kjer lahko uporabimo vse obstoječe storitve 3iOS-a. Slabost je predvsem povečana kompleksnost ogrodja 3iOS. Ta način povezave je sicer bolj zanimiv kot povezava z GWT-jem, saj bi v primeru povezave z ogrodjem Wicket dejansko razširil množico aplikacij, ki jih trenutno lahko razvijemo samo z ogrodjem 3iGUI. Nasprotno pa bi povezava z ogrodjem GWT vplivala predvsem na zmogljivosti (odzivnost) uporabniškega vmesnika, kar pa trenutno ne predstavlja take pridobitve, da bi se ta povezava izplačala.

Poglavje 7

Sklepne ugotovitve

Razvoj lastnega spletnega ogrodja, ki je zastavljeno tako široko, da zajema večino potreb razvoja spletnih aplikacij v podjetju, je bila gotovo tvegana odločitev. Mislim, da se je tveganje izplačalo in da je projekt uspešen. Seveda ostaja glavno vprašanje, kaj bi pomenilo, če bi se na začetku namesto lastnega razvoja takoj odločili za uporabo katerega od aktualnih (najverjetneje odprtokodnih) ogrodij, na katerega bi **neposredno vezali** aplikativno kodo. S tem bi prekinili podporo starejšim lastnim ogrodjem in že razvitim aplikacijam, ki bi jih sčasoma verjetno ponovno razvili na podlagi izbranega ogrodja. Na podlagi izkušenj dela v podjetju sklepam, da bi ogrodje, karšnokoli bi že bilo, zagotovo dodatno nadgradili in razvili specifične komponente, prilagodili videz ipd. Prav zato ocenjujem, da je lahko lasten razvoj ogrodja dolgoročno upravičen, predvsem ko govorimo o večjem številu sorodnih aplikacij, ki se med seboj povezujejo in ko ogrodje omogoča zaledno povezavo z drugimi (odprtokodnimi) rešitvami. Ocenjujem, da v takem primeru cena lastnega razvoja ni toliko večja od cene obvladovanja, nadgrajevanja, prilagajanja novim različicam, razhroščevanja ter podobnih aktivnosti, ki so posledica neposredne povezave aplikacij z „brezplačnim“ ogrodjem in da je mogoče razliko v ceni dolgoročno pokriti z učinki vseh prednosti, ki jih nudi lasten razvoj.

Med pisanjem diplomske naloge so me najbolj zanimale možnosti povezovanja ogrodij, ki sem jih opisal v prejšnjem poglavju. Znanje, ki sem ga pridobil s preizkušanjem povezav in testiranjem ogrodij, mi bo gotovo zelo koristilo. Za uporabo Wicket-3iOS bi se odločil, če bi narava aplikacije zahtevala bolj „spletni“ pristop, za GWT-3iOS, če bi želel doseči najvišjo stopnjo skalabilnosti, za GWT-3iGUI pa, če bi želel izboljšati odzivnost uporabniških vmesnikov obstoječih aplikacij brez poseganja v njihovo kodo. Moja končna

ocena primerjave ogrodij je, da povezovanje ogrodij zagotovo doda vrednost obstoječemu lastnemu ogrodju, a tudi dodatne stroške zaradi povečane kompleksnosti sistema, dodatnega vzdrževanja in morebitnega uvajanja razvijalcev.

7.1 Ogrodje v praksi

Na podlagi ogrodij 3iOS in 3iGUI smo v podjetju razvili večje število različnih aplikacij. Poleg tega smo 3iOS uspešno povezali z lastnim geografskim informacijskim sistemom, ki je pred tem temeljil na starejšem ogrodju. Na podlagi ogrodja 3iGUI je bilo razvitih tudi nekaj specifičnih višjenivojskih ogrodij, eno od teh je postalo podlaga za večino novejših aplikacij. Poleg aplikacij smo na podlagi 3iOS-a razvili tudi nekaj pomebnih spletnih storitev in v sklopu elektronskega poslovanja izvedli nekaj uspešnih povezav z drugimi sistemi. Pri uporabi ogrodja so se v praksi pokazale ali pa zgolj nakazale prednosti in slabosti, ki sem jih navedel v nadaljevanju.

7.1.1 Prednosti

- Popolno poznavanje delovanja ogrodja omogoča kvalitetno in hitro podporo razvijalcem med razvojem in razhroščevanjem.
- Ogrodje zelo dobro prilagajamo specifičnim zahtevam tako po videzu in občutku (angl. look and feel) aplikacij kot njihovem razvoju, nastavljanju in vzdrževanju v produkciji.
- Napake, nedoslednosti in morebitne potrebe po spremembah in nadgradnjah ogrodja rešujemo zelo hitro.
- Vse aplikacije, ki uporabljajo ogrodje, so znane in dostopne na enem mestu. To omogoča preverjanje pravilnosti vzorcev uporabe ogrodja, izboljšanje diagnostike pri nepravilni uporabi ogrodja, neposredno opozarjanje oziroma izobraževanje razvijalcev ipd. Pri morebitnih spremembah temeljnih vmesnikov ogrodja si lahko privoščimo pregled statistike napak, ki bi jih povzročila določena sprememba. Na njihovi podlagi se lahko odločimo za spremembo in sprotno prilagoditev ter testiranje aplikacij.
- Lasten razvoj zagotavlja stabilen in predvidljiv razvoj vmesnikov API. Razvijalci imajo zagotovljeno dolgoročno stabilnost vmesnikov in vzorcev uporabe ogrodja. Uvajanje večjih sprememb se izvaja po dogovoru z

razvijalci in po pregledu morebitnih vplivov na že razvite aplikacije. Prilagajanje morebitnim spremembam novejših različic zalednih ogrodij, ki jih ograjuje 3iOS, je stvar razvijalcev ogrodja in ne aplikacij.

- Dobre rešitve v aplikacijah se hitro preselijo v ogrodje. Tukaj velja omeniti predvsem ogrodje 3iGUI, ki je veliko svojih rešitev pridobilo na račun razvoja aplikacij, v katerih so si razvijalci z lastnimi rešitvami olajšali delo.

- Lastno ogrodje nudi možnost vzporednega dela pri vpeljevanju novih tehnologij, saj najprej določimo vmesnike za dostop do novih storitev, nato pa sledi vzporedni razvoj aplikacij ter implementacije novih modulov ogrodja, ki uporabljajo novo tehnologijo.

- Popolno obvladovanje ogrodja je povzročilo več dogovarjanja o razvoju uporabniških vmesnikov, standardizaciji le-teh in uvajanju ožje usmerjenih komponent.

- S časom smo ustvarili zajetno knjižnico ponovno uporabnih komponent in drugih gradnikov. Delo na projektih je vse bolj usmerjeno v zasnovo vsebinskih rešitev, manj pa v tehnično izvedbo le-teh. 3iOS in 3iGUI sta s časom postala samoumevna.

7.1.2 Slabosti

- Lasten razvoj ni poceni, predvsem ga je težko ovrednotiti ali primerjati z drugačnimi rešitvami. Dolgoročno sicer cena lastnega razvoja oziroma vzdrževanja ogrodja pada (posledica vse večjega števila standardnih gradnikov in njihove ponovne uporabe).

- Kot vsako ogrodje, ki se prilagaja aktualnim potrebam, je tudi lastno ogrodje podvrženo postopnemu širjenju funkcionalnosti, torej postaja s časom vse bolj zapleteno.

- Pri iskanju novih kadrov lahko zahtevamo samo poznavanje ogrodij s sorodno arhitekturo.

- Poenostavljanje uporabniških vmesnikov, ki smo ga na začetku želeli doseči tudi z argumentom o omejitvah brskalnikov in spleta, je bilo le deloma uspešno. S postopnim večanjem števila različnih komponent in

njihovih zmožnosti so se pričakovanja oziroma zahteve do spletnih aplikacij zopet povečale in hitro dosegle stopnjo, ki je primerljiva z namiznimi aplikacijami.

- Razvoj lastnega (večjega) ogrodja je za podjetje tvegan tudi zato, ker lahko pomeni večjo odvisnost od posameznika ali nekaj posameznikov, zato je na dolgi rok potrebno poskrbeti, da ogrodje obvlada več oseb.
- Število novosti, ki se lahko hitro uvedejo v ogrodje, je v primerjavi z dobro podprtimi odprtokodnimi ogrodji majhno, saj slednje tipično podpira (večja) skupnost. Zato je pri dodajanju novih komponent in drugih novosti potrebno dobro določiti prioritete razvoja.
- Popolna svoboda pri snovanju, razvoju in eksperimentiranju z novejšimi različicami ogrodja je lahko dvorezen meč, saj obstaja verjetnost, da projekt zaide v slepo ulico (izbira napačnega „zaledja“, zanašanje na razne „ad hoc“ rešitve, ki nimajo prihodnosti itd.).
- Verjetnost, da so razvijalci aplikacij neposredno soudeleženi tudi pri razvoju ogrodja, je v primeru lastnega razvoja velika. Čeprav je smiselno, da je razvijalec ogrodja tudi njegov uporabnik, pa lahko to med razvojem aplikacij povzroči uporabo „bližnjic“ v kodi ogrodja, ki niso dovolj premišljene in posplošene.

7.2 Možne izboljšave

Možnih izboljšav ogrodja je veliko. Najpomembnejše so tiste, ki omogočajo hitrejši razvoj in uporabniško bolj prijazne aplikacije. Med prve spada poenostavljanje zunanjih nastavitev komponent. V več primerih se je izkazalo, da bi lahko s primernim dogovorom **zmanjšali število nastavitev** (angl. convention over configuration). Tak pristop bi precej poenostavil delo razvijalcem in oblikovalcem pogledov uporabniškega vmesnika. Večjo **prijaznost** uporabniških vmesnikov pa bi bilo moč doseči z boljšo podporo uporabi tipkovnice, saj določene komponente podpirajo zgolj uporabo miške. Med ostale izboljšave spadajo še:

- **izboljšanje aktualne testne različice 3iOS-ovega jedra**, ki ni vezana na splet in omogoča testiranje kode brez uporabe spletnega strežnika,
- **prenova modula application_state_manager**, ki posameznih stanj ne bi več obravnaval monolitno, temveč bi jih razbil na posamezne dele

(še boljši izkoristek pomnilnika, saj bi v pomnilniku zadrževal le aktualni del stanja ne pa celega stanja aplikacije),

- **zamenjava implementacije modula service_manager** z implementacijo, ki bi namesto aktualnega razporejevalnika opravil uporabila naprednejšo odprtokodno rešitev, kot je na primer Quartz,
- **omogočiti kombiniranje različnih modulov** jedra z enostavnimi nastavitvami registra (nekatere kombinacije modulov je trenutno mogoče opraviti samo z izpeljavo in kasnejšo uporabo specifičnega jedra - torej s poseganjem v kodo),
- **omogočiti zamenjavo različice jedra** v stvarnem času ter dodati možnost izvajanja več različic hkrati (možnost nadgradnje ogrodja in aplikacij v stvarnem času, brez potrebe po ponovnem zagonu oz. motenj v produkciji),
- **izboljšanje odzivnosti** aplikacij s povezavo GWT-3iGUI,
- **razširitev množice podprtih aplikacij** s povezavo Wicket-3iOS.

Slike

2.1 Vloga skupnih in vročih točk v povezavi z obratom kontrole . . 9

3.1 Vzorec MVC 20
3.2 Vzorec MVP 27

4.1 Primer sistemske aplikacije v 3iOS-u 44
4.2 Shema izvajanja predložnega stroja 48
4.3 Primer kompleksne postavitve komponent 62
4.4 Primer kompleksne postavitve komponent, ki so bile izrisane v razhroščevalnem načinu izvajanja aplikacije 64

Tabele

2.1 Primerjava knjižnice z ogrodjem. 7

Literatura

[1] M. Baker. (2009, 2. okt.). What is a Software Framework? And why should you like 'em? *Cimetrix, Inc.* Dostopno na: http://info.cimetrix.com/blog/bid/22339/What-is-a-Software-Framework-And-why-should-you-like-em

[2] M. Hunter. (2006, jul.). Tidying the House: The MVPC Software Design Pattern. Dostopno na: http://www.martinhunter.co.nz/articles/MVPC.pdf

[3] R. E. Johnson, B. Foote. (1988, jun.-jul.). Designing Reusable Classes. *Journal of Object-Oriented Programming.* Št. 1, zv. 2, str. 22-35. Dostopno na: http://www.laputan.org/drc/drc.html

[4] M. E. Markiewicz, C. J. P. Lucena, "Object Oriented Framework Development," Crossroads, št. 7, zv. 4, str. 3-9, 2001.

[5] T. O'Reilly. (2005, 30. sep.). What Is Web 2.0. *O'Reilly Media, Inc.* Dostopno na: http://oreilly.com/pub/a/web2/archive/what-is-web-20.html?page=1

[6] Wikipedia contributors. (2010, 20. avg.). Container (data structure). *Wikimedia Foundation, Inc.* Dostopno na: http://en.wikipedia.org/wiki/Container_(data_structure)

[7] Wikipedia contributors. (2010, 29. avg.). Inversion of control. *Wikimedia Foundation, Inc.* Dostopno na: http://en.wikipedia.org/wiki/Inversion_of_control

[8] Wikipedia contributors. (2010, 23. jul.). Hollywood Principle. *Wikimedia Foundation, Inc.* Dostopna na: http://en.wikipedia.org/wiki/Hollywood_Principle

[9] Wikipedia contributors. (2010, 20. avg.). Dependency injection. *Wikimedia Foundation, Inc.* Dostopno na: http://en.wikipedia.org/wiki/Dependency_injection

[10] Wikipedia contributors. (2010, 29. avg.). Web application. *Wikimedia Foundation, Inc.* Dostopno na: http://en.wikipedia.org/wiki/Web_application

[11] Wikipedia contributors. (2010, 25. avg.). Web application framework. *Wikimedia Foundation, Inc.* Dostopno na: http://en.wikipedia.org/wiki/Web_application_framework

[12] Wikipedia contributors. (2010, 31. avg.). Model–view–controller. *Wikimedia Foundation, Inc.* Dostopno na: http://en.wikipedia.org/wiki/Model-view-controller

[13] Wikipedia contributors. (2010, 16. avg.). Facade Pattern. *Wikimedia Foundation, Inc.* Dostopno na: http://en.wikipedia.org/wiki/Facade_pattern

[14] Wikipedia contributors. (2010, 31. avg.). Decorator Pattern. *Wikimedia Foundation, Inc.* Dostopno na: http://en.wikipedia.org/wiki/Decorator_pattern

[15] (2010). Apache Wicket. *The Apache Software Foundation.* Dostopno na: http://wicket.apache.org/

[16] (2006, 13. dec.). Application Design Patterns: Producer/Consumer. *National Instruments Corporation.* Dostopno na: http://zone.ni.com/devzone/cda/tut/p/id/3023

[17] (1994). Building Object-Oriented Frameworks. *Taligent Inc.* Dostopno na: http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/buildingoo.pdf

[18] (2002). Designing Enterprise Applications with the J2EE Platform, Second Edition. *Sun Microsystems, Inc.* Dostopno na: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html

[19] (2002). Designing Enterprise Applications with the J2EE Platform, Second Edition. *Sun Microsystems, Inc.* Dostopno na: http://java.sun.com/blueprints/guidelines/ designing_enterprise_applications_2e/web-tier/web-tier5.html

[20] (2005, 2. feb.). Dissection of an Application Framework. *Apress Publishing.* Dostopno na: http://www.aspfree.com/c/a/.NET/Dissection-of-an-Application-Framework/

[21] (2010). Google Web Toolkit. *Google.* Dostopno na: http://code.google.com/intl/sl/webtoolkit/

[22] (2010). JDK 6 Swing (Java Foundation Classes). *Oracle Corporation.* Dostopno na: http://download.oracle.com/javase/6/docs/technotes/guides/swing/

[23] (2005, 20. avg.). What is Component? *TechTarget.* Dostopno na: http://whatis.techtarget.com/definition/0,,sid9_gci211826,00.html