UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Jure Cerjak

# Razvoj spletnih aplikacij s platformo Zope

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Marjan Krisper

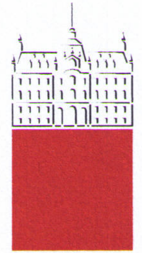
Ljubljana, 2010

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.

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: 01651/2010

Datum: 05.04.2010

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

Kandidat: **JURE CERJAK**

Naslov: **RAZVOJ SPLETNIH APLIKACIJ S PLATFORMO ZOPE**
**WEB APPLICATION DEVELOPMENT WITH ZOPE**

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V okviru diplomske naloge predstavite značilnosti spletnih aplikacijskih ogrodij in ogrodij in posebej platformo Zope, ki temelji na programskem jeziku Python in omogoča razvoj prilagodljivih in razširljivih spletnih sistemov v obliki rahlo povezanih komponent. Opišite njene posebnosti zlasti možnosti uporabe objektne baze podatkov. Prikažite praktični primer razvoja aplikacije z uporabo ogrodja Grok in možnosti uporabe sodobnih agilnih pristopov. V zaključkih predstavite prednosti in slabosti platforme Zope.

Mentor:

prof. dr. Marjan Krisper

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

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 Jure Cerjak,

z vpisno številko 63030142,

sem avtor diplomskega dela z naslovom:

Razvoj spletnih aplikacij s platformo Zope

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Marjana Krisperja
- 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 11.10.2010 Podpis avtorja:

## Zahvala

*Za potrpežljivost in strokovno pomoč pri izdelavi diplomske naloge se zahvaljujem profesorju dr. Marjanu Krisperju in dr. Damjanu Vavpotiču.*

*Posebna zahvala gre družini in prijateljem, ki so me vselej podpirali.*

# Kazalo

Povzetek 1
Abstract 2
1. Uvod 3
2. Predstavitev ogrodij 4
2.1 Kaj je programsko ogrodje? 4
2.2 Spletna aplikacijska ogrodja 5
3. Predstavitev platforme Zope 6
3.1 Kaj je Zope? 6
3.2 Kratek pregled zgodovine platforme Zope 7
3.3 Zbirka knjižnic ZTK 9
3.3.1 Komponentna arhitektura Zope (ZCA) 9
3.3.2 Objektna baza Zope (ZODB) 9
3.3.3 HTML/XML predloge 9
3.3.4 Avtomatsko generiranje in validiranje form 10
3.3.5 Internacionalizacija 10
3.3.6 Varnost 10
3.3.7 Indeksiranje in iskanje 10
3.3.8 Testiranje 11
3.4 Ogrodja, ki uporabljajo ZTK 12
4. Komponentna arhitektura Zope (ZCA) 14
4.1 Komponente ZCA 15
4.2 Vmesniki 15
4.2.1 Vmesniki v programskem jeziku Python 15
4.2.2 Vmesniki Zope 16
4.3 Pretvorniki 18
4.3.1 Pretvorniki kot načrtovalski vzorec 18
4.3.2 Pretvorniki ZCA 19
4.4 Storitve ZCA 25
4.4.1 Delitev storitev ZCA 25
4.4.2 Tovarne 26
4.5 Dogodki 26
5. Objektna baza Zope (ZODB) 28
5.1 Zgradba ZODB 28
5.2 Značilnosti ZODB 29

5.2.1 Transakcije imajo lastnosti ACID 29
5.2.2 Transparentnost 30
5.2.3 Razveljavitev transakcij 31
5.2.4 Izbira načina shranjevanja podatkov in skalabilnost 32
5.3 Prednosti in slabosti ZODB 34
5.3.1 Prednosti 34
5.3.2 Slabosti 35
5.3.3 Kdaj uporabiti ZODB? 36
6. Razvoj prototipa aplikacije 37
6.1 Uvod 37
6.2 Analiza in načrtovanje 37
6.2.1 Opis problemske domene 37
6.2.2 Diagrami primerov uporabe (PU) 38
6.2.3 Nefunkcionalne zahteve 64
6.2.4 Arhitektura 65
6.2.5 Diagrami zaporedja 68
6.2.6 Podatkovni model 72
6.2.7 Diagrami paketov 75
6.3 Razvoj aplikacije v praksi 76
6.3.1 Priprava razvojnega okolja 76
6.3.2 Izvedba primerov uporabe 80
7. Sklepne ugotovitve 96
Seznam uporabljenih virov 100

# Seznam uporabljenih kratic in simbolov

| | |
|---|---|
| **Zope**<br>Z Object Publishing Environment | Platforma za gradnjo spletnih aplikacij. |
| **ZTK**<br>Zope Toolkit Library | Zbirka knjižnic za gradnjo spletnih aplikacijskih ogrodij ali spletnih aplikacij. |
| **ZODB**<br>Zope object database | Objektna baza platforme Zope. |
| **ZCA**<br>Zope component Architecture | Komponentna arhitektura Zope. |
| **ZEO**<br>Zope Enterprize Objects | Način shranjevanja objektov v bazi ZODB, ki omogoča dostop in shranjevanje objektov na oddaljenem podatkovnem strežniku. |
| **PAU**<br>Pluggable Authentication Utility | Prilagodljiva storitev platforme Zope, namenjena avtentikaciji. |
| **API**<br>Application Program Interface | Aplikacijski programski vmesnik. |
| **HTML**<br>Hyper Text Markup Language | Oznacevalni jezik za določitev strukture dokumentov, ki se prenasajo po spletu in pregledujejo s spletnimi brskalniki. |
| **XML**<br>Extensible Markup Language | Razširljiv označevalni jezik. |
| **TAL**<br>Template Attribute Language | Atributni jezik za kreiranje dinamičnih predlog. |
| **TALES**<br>Template Attribute Language Expression Syntax | Izrazi, ki se lahko uporabljajo za pridobivanje podatkov okviru jezikov TAL in METAL. |
| **METAL**<br>Macro Expansion Template Attribute Language | Atributni jezik za strukturirano predprocesiranje makrov. |
| **ZPT**<br>Zope Page Templates | Sistem za generiranje HTML ter XML dokumentov. |
| **URL**<br>Uniform Resource Locator | Globalni naslov dokumentov in ostalih virov na spletu. |
| **ORM**<br>Object-relational mapping | Pretvorba podatkov med nekompatibilnimi tipnimi sistemi v objektno usmerjenih programskih jezikih. |
| **MVC**<br>Model-View-Controller | Tip arhitekture. |
| **HTTP**<br>HyperText Transfer Protocol | Glavna metoda za prenos informacij na spletu. |
| **TCP**<br>Transmission Control Protocol | Množica protokolov, ki izvaja protokolski sklad prek katerega teče internet. |
| **SMS**<br>Short Message Service | Storitev za pošiljanje kratkih tekstovnih sporočil. |
| **ACID**<br>*Atomicity, Consistency, Isolation, Durability* | Množica lastnosti, ki zagotavlja zanesljivost transakcij podatkovne baze. |
| **ZCML**<br>Zope Configuration Markup Language | Jezik za konfiguracijo komponent. |
| **CRUD**<br>Create, Read, Update, Delete | Osnovne operacije pri delu s trajnimi entitetami. |
| **PU** | Zaključen tok dogodkov, ki akterju vrne merljiv |

| Primeri uporabe | rezultat. |
|---|---|
| **SQL**<br>Structured Query Language | Strukturirani povpraševalni jezik za delo s podatkovnimi bazami. |
| **JSON**<br>JavaScript Object Notation | Preprosti tekstovni standard za izmenjavo podatkov v berljivem formatu. |

# Povzetek

Spletna aplikacijska ogrodja služijo podpori razvoju dinamičnih spletnih strani, aplikacij in storitev. Izbiramo lahko med številnimi že uveljavljenimi platformami, v zadnjih letih pa lahko opazimo porast števila ogrodij, ki temeljijo na dinamičnih programskih jezikih, kot sta Python in Ruby.

V svoji diplomski nalogi sem predstavil platformo Zope, ki je kljub zrelosti in nekaterim zanimivim lastnostim med razvijalci relativno neznana. Temelji na programskem jeziku Python in omogoča razvoj prilagodljivih in razširljivih spletnih sistemov v obliki rahlo povezanih komponent, posebnost platforme pa je tudi uporaba objektne baze ZODB. Prikazal sem praktični primer razvoja z ogrodjem Grok, ki uporablja večino komponent platforme Zope ob upoštevanju sodobnih agilnih pristopov.

V uvodnem delu bom predstavil značilnosti spletnih aplikacijskih ogrodij in ogrodij na splošno. Sledi pregled in krajši opis glavnih sestavnih delov platforme Zope. Drugi del diplomske naloge je namenjen podrobnejši predstavitvi komponentne arhitekture Zope in objektne baze ZODB. Opisu platforme sledi predstavitev razvoja z izbranim ogrodjem. Prikazan je razvoj prototipa aplikacije za pomoč pri organizaciji dogodkov rekreativnih športnih ekip.

V zaključku sem izpostavil prednosti in slabosti platforme Zope, ki sem jih spoznal med razvojem in podal predloge za izboljšavo izdelanega prototipa.

**Ključne besede:**
Zope, komponentna arhitektura, objektna baza, Python, vodenje športnih ekip

# Abstract

A web application framework is designed to support the development of dynamic websites, Web applications and Web services. There are many already established platforms to choose from. Recent years have witnessed an increasing number of web application frameworks based on dynamic programming languages such as Python and Ruby.

In my thesis I presented Zope, which is relatively unknown among developers despite the maturity of the platform and the fact that it offers some interesting features. It is based on Python and allows for the development of flexible and scalable web systems in the form of loosely coupled components. An interesting feature is also the use of object database ZODB. I will show a practical example of the development with the Grok framework, which uses most of the components of the Zope platform while taking into account modern agile approaches.

In the introduction I describe the characteristics of web application frameworks and frameworks in general. This is followed by an overview and a brief description of the main components of the Zope platform.The second section features a more detailed presentation of the Zope component architecture and object database ZODB. Description of the platform is followed by a presentation of the development cycle using the chosen framework. I present the development of a prototype application intended to assist in organizing events for recreational sports teams.

In the concluding section of my paper I emphasize the strenghts and weaknesses of the Zope platform, which came to my attention during the development, and make some suggestions for improving the application prototype.

**Keywords:**
Zope, component architecture, object database, Python, sports team management

# 1.Uvod

V zadnjih letih se je poleg že uveljavljenih tehnologij za gradnjo spletnih aplikacij, kot so Java, PHP ter ASP.NET, močno povečala priljubljenost spletnih aplikacijskih ogrodij, ki temeljijo na skriptnih programskih jezikih Python, Ruby in Perl. Ogrodja Ruby on Rails, Django, TurboGears, Grok, Pylons itd. so objektno usmerjena in omogočajo hiter ter učinkovit razvoj. Poudarjajo načela agilnosti, pomen ponovne uporabljivosti komponent ter konvencij namesto eksplicitne konfiguracije, velik poudarek pa je namenjen tudi testiranju. Programerju ponujajo vrsto uporabnih knjižnic za avtentikacijo in avtorizacijo, delo z vnosnimi formami, preprosto upravljanje s podatki s pomočjo sistema ORM (angl. *Object Relational Mapper*) ipd.

Eden izmed najstarejših tovrstnih sistemov je Zope. Začetek njegovega razvoja sega v leto 1998. Predstavlja prvo ogrodje, napisano v jeziku Python, njegova popularnost pa je prispevala tudi k širši uveljavitvi samega jezika [1]. Sprva je bil zasnovan kot ogromen monoliten sistem, vendar se je skozi čas preoblikoval v zbirko manjših ponovno uporabljivih komponent, ki so postale osnova za nova agilnejša ogrodja. Nekatere med njimi so dovolj splošne tudi za uporabo v ostalih projektih skupnosti Python.

Zope v osnovi ponuja podobno funkcionalnost kot zgoraj omenjena ogrodja, vendar se v nekaterih pogledih od njih tudi bistveno razlikuje. Je izrazito objektno naravnan, kar se odraža tako v načinu pretvarjanja naslova URL v ustrezne akcije kot v uporabi objektne baze ZODB (angl. *Zope object database*). Njegova posebnost je tudi uporaba komponentne arhitekture ZCA (angl. *Zope component Arhitecture*), ki omogoča gradnjo zelo prilagodljivih in razširljivih sistemov.

Cilj diplomske naloge je predstaviti platformo Zope, izpostaviti njene prednosti in slabosti ter predstaviti praktični primer razvoja z ogrodjem Grok, ki ponuja moč platforme Zope v uporabniku prijaznem paketu. Razvil bom prototip spletne aplikacije za podporo aktivnostim pri organiziranju rekreativnih športnih dogodkov. Glavni namen aplikacije je poenostaviti organizacijo in komunikacijo med člani ekipe. Podobne rešitve na spletu že obstajajo in služijo kot opora pri določanju funkcij, ki jih mora aplikacija podpirati. Nobena rešitev pa ni bila razvita z mislijo na internacionalizacijo, zato sem svojo aplikacijo razvil tako, da to omogoča.

V prvem delu diplomske naloge bom podal splošen opis ogrodij za izdelavo spletnih aplikacij, kratko zgodovino platforme Zope in njene glavne značilnosti. V nadaljevanju bom podrobneje opisal najzanimivejši značilnosti, komponetno arhitekturo Zope ter objektno bazo ZODB. Zadnji del pa je namenjen predstavitvi razvoja z izbranim ogrodjem na praktičnem primeru.

# 2. Predstavitev ogrodij

## 2.1 Kaj je programsko ogrodje?

Programsko ogrodje (angl. *Software framework*) je abstrakcija, pri kateri lahko uporabniška programska koda selektivno povozi ali specializira programsko kodo, ki ponuja splošno funkcionalnost. [2]

Ogrodja so poseben primer programskih knjižnic. Podobno kot knjižnice so tudi ogrodja ponovno uporabljive abstrakcije programske kode, ovite v dobro definiran vmesnik API (angl. *Application programming interface*), vendar se od knjižnic razlikujejo v naslednjih točkah [2]:

- *inverzija nadzora*:
  V nasprotju s programskimi knjižnicami ali navadnimi uporabniškimi aplikacijami, splošnega toka nadzora ne določa klicatelj, ampak ogrodje.

- *privzeto obnašanje*:
  Ogrodje ima privzeto obnašanje, ki mora biti uporabno in ne le serija nekoristnih ukazov (angl. *no-ops*).

- *razširljivost*:
  Uporabnik lahko razširi ali specializira ogrodje s selektivnim redefiniranjem programske kode oz. dodajanjem kode z določeno funkcionalnostjo.

- *nespremenljivost programske kode ogrodja*:
  Uporabniki lahko programsko kodo ogrodja razširijo, vendar je ne smejo spremeniti.

Inverzija nadzora je ključna razlika med ogrodjem in knjižnico. Pri uporabi knjižnice sami določimo, kdaj bomo izvedli določeno akcijo, pri ogrodju pa specifično funkcionalnost aplikacije vključimo v predhodno definirane vstopne točke, t.i. vroče točke (angl. *hot spots*) [2].

Nekateri načini za vključitev v vroče točke so [3]:

- uporaba načrtovalskega vzorca tovarna (angl. *factory)*
- uporaba načrtovalskega vzorca iskalnik storitev (angl. *service locator)*
- vstavljanje preko konstruktorja (angl. *constructor injection)*
- vstavljanje preko metode set (angl. *setter injection)*
- vstavljanje preko vmesnika (angl. *interface injection)*

Zadnje tri točke predstavljajo poseben primer inverzije nadzora, ki se imenuje vstavljanje odvisnosti (angl. *dependancy injection*). Če nek objekt potrebuje določeno storitev, lahko dobi dostop do nje z neposredno referenco na lokacijo storitve ali pa od iskalnika storitev zahteva dostop do implementacije zahtevane storitve. Pri vstavljanju odvisnosti pa je

referenca na implementacijo zahtevane zunanje storitve avtomatsko vstavljena v atribut ciljnega razreda ob kreiranju objekta. Prednost tega pristopa je v večji fleksibilnosti, saj je preprosti ustvariti alternativno implementacijo določenega tipa storitve in nato s pomočjo konfiguracijske datoteke določiti, katera implementacija naj se uporabi brez sprememb objektov, ki to storitev uporabljajo. Posledica pretirane uporaba vstavljanja odvisnosti pa je povečana kompleksnost aplikacije, saj mora razvijalec za razumevanje sistema preklapljati med domeno izvorne kode in konfiguracije [4]. To je bila tudi ena izmed glavnih kritik sistema Zope 3 in povod za razvoj ogrodja Grok, pri katerem vso potrebno konfiguracijo definiramo v sami izvorni kodi.

## 2.2 Spletna aplikacijska ogrodja

Spletno aplikacijsko ogrodje je programsko ogrodje, ki služi podpori razvoju dinamičnih spletnih strani, aplikacij in storitev. Njegov namen je poenostaviti ponavljajoče se aktivnosti pri spletnem razvoju. Večina ogrodij ponuja komponente za dostop do podatkovne baze, za izdelavo dinamičnih predlog in upravljanje seje. Z uporabo ogrodja razvijalec pri razvoju prihrani veliko časa, saj mu ni potrebno vsakič znova implementirati osnovnih sestavnih delov, ki se pojavljajo pri razvoju vsake spletne aplikacije.

Spletna aplikacijska ogrodja pogosto ponujajo naslednje funkcionalnosti [5]:

- *varnost*
  Nekatera ogrodja vsebujejo podporo za avtentikacijo in avtorizacijo, ki omogoča identifikacijo uporabnikov in omejevanje dostopa do funkcij aplikacije glede na definirano varnostno politiko.

- *upravljanje seje*
  Protokol HTTP je brez stanja. Odjemalec mora z vsako zahtevo HTTP GET ali HTTP POST s strežnikom vzpostaviti novo omrežno povezavo TCP. Strežnik se zato ne more zanašati na vzpostavljeno povezavo TCP dlje kot za čas trajanja HTTP GET ali HTTP POST operacije. Upravljanje seje je mehanizem, ki protokolu HTTP omogoča ohranjanje sejnega stanja.

- *trajnost*
  Veliko ogrodij ima enoten vmesnik API za dostop do podatkovne baze, kar spletnim aplikacijah omogoča kompatibilnost z vrsto podatkovnih baz brez sprememb v kodi. Poleg tega nekatera ogrodja vsebujejo tudi orodja ORM, orodja za migracijo sheme podatkovne baze itd.

- *pretvorba naslovov URL v ustrezne akcije*
  Ogrodja uporabljajo različne načine za interpretiranje naslovov URL. Pogost način je npr. uporaba regularnih izrazov. Pri platformi Zope pa naslov URL predstavlja drevo objektov: ogrodje potuje od objekta do objekta in za zadnji objekt v verigi izvede ustrezno akcijo. Omenjeni mehanizem pretvorbe se imenuje obhod (angl. *traversal*).

- *sistem za dinamične predloge*
  Dinamične spletne strani poleg statičnih vsebin vsebujejo tudi del, ki dinamično ustvari kodo HTML na podlagi stanja v podatkovni bazi in uporabniškega vnosa.

# 3. Predstavitev platforme Zope

## 3.1 Kaj je Zope?

Beseda Zope je kratica, ki pomeni »*Z Object Publishing Environment*«, torej sistem za objavljanje objektov (na spletu in drugih sistemih) [6].

Uradna definicija se glasi [7]:

Zope je:

- *zbirka brezplačne programske opreme,*
- *razvita s strani podjetja Zope v sodelovanju s skupnostjo programerjev,*
- *ki se lahko uporablja v celoti ali v delih*
- *za nadzorovanje kompleksnosti pri združevanju programskih komponent,*
- *varno objavo objektov na spletu ter drugih sistemih,*
- *omogoča pa tudi preprosto zagotavljanje kakovosti.*

Poglejmo si vsako izmed točk podrobneje:

- *zbirka brezplačne programske opreme:*
  Zope je velika zbirka komponent, namenjenih gradnji spletnih informacijskih sistemov, implementiranih v programskem jeziku Python. Temeljijo na odprtokodnem principu, torej so na voljo javnosti za prosto uporabo in spreminjanje.

- *razvita  s strani podjetja Zope v sodelovanju s skupnostjo programerjev:*
  Velik del platforme Zope 2 je bil razvit s pomočjo skupnosti, Zope 3 pa je skoraj v celoti produkt njihovega dela. Odprtokodni pristop k razvoju se je torej izkazal za zelo uspešnega.

- *ki se lahko uporablja kot celota ali v delih:*
  Zope je bil sprva zasnovan kot velik monoliten sistem, ki je poleg aplikacijskega strežnika ponujal tudi vrsto knjižnic za razvoj aplikacij. Knjižnice so bile med seboj močno soodvisne in jih ni bilo mogoče uporabljati izven domene sistema. S prihodom platforme Zope 3 pa so celoten sistem razdelili na množico manjših in med seboj rahlo povezanih komponent, ki se lahko uporabljajo tudi v drugih projektih, ki uporabljajo programski jezik Python.

- *za nadzorovanje kompleksnosti pri združevanju programskih komponent:*
  Tehnologija, ki omogoča to rahlo povezavo, je komponentna arhitektura Zope (v nadaljevanju ZCA - angl. *Zope Component Architecure*), ki je posebna izpeljanka principov komponentne arhitekture. Omogoča ločen razvoj posameznih komponent, torej je primerna za delo v velikih skupinah, močno pa olajša tudi vzdrževanje in prilagajanje programske opreme. ZCA predstavlja temelj sistema Zope, zato bo v nadaljevanju predstavljena podrobneje.

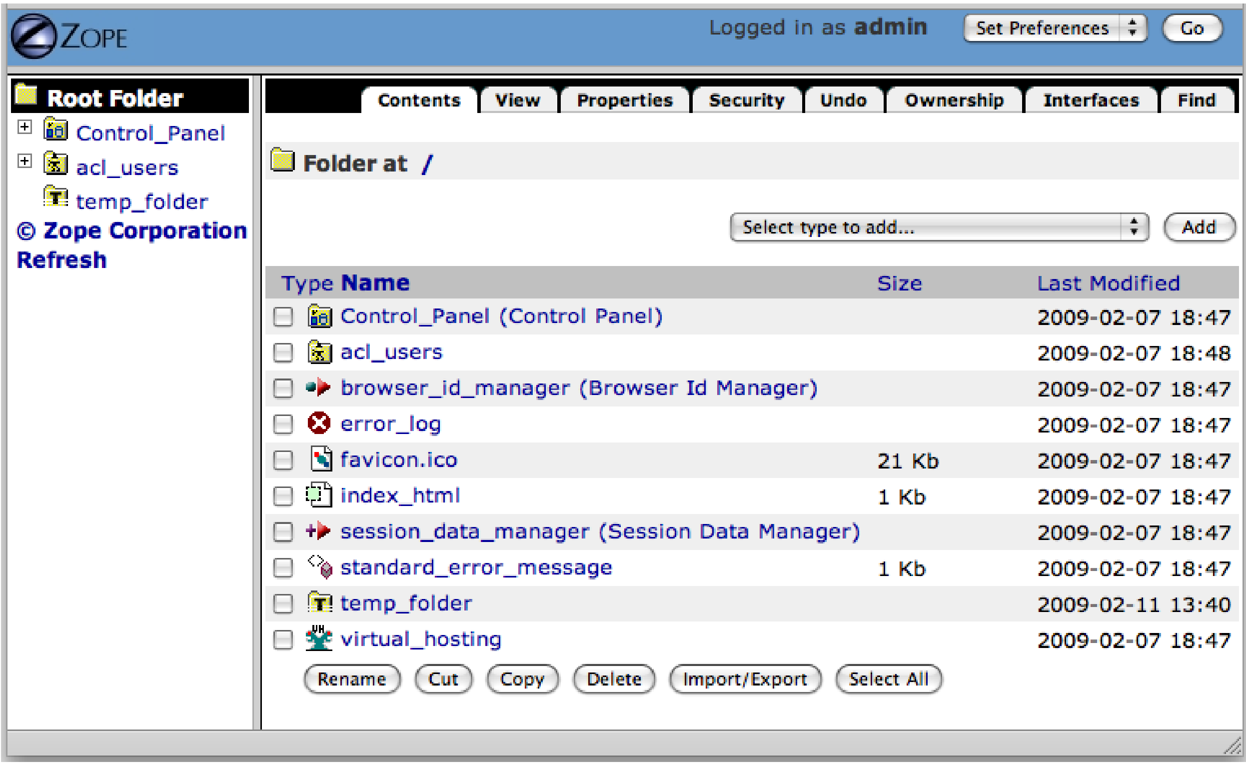
- *varno objavo objektov na spletu ter drugih sistemih:*
  Zope je močno objektno usmerjen sistem. Poleg aplikacijske logike in sistema za objavo sta tudi trajnostni in varnostni mehanizem objektno naravnana. Zope za shranjevanje podatkov namreč, v nasprotju z večino ostalih sistemov, uporablja objektno bazo ZODB (*Zope Object Database*), varnostni sistem pa sestavlja več manjših in prilagodljivih komponent, s katerimi lahko zgradimo kompleksen varnostni sistem.

  Kot potrditev dobre varnosti sistema Zope je vredno omeniti, da je Zope trenutno v procesu pridobitve varnostnega certifikata ISO-15408 (*Common Criteria Certificate*) [7]. Ta certifikat naj bi zagotavljal, da je bil proces specifikacije, implementacije in evaluacije varnostnega produkta izveden na rigorozen in standarden način [8]. Tudi priljubljen sistem za upravljanje z vsebinami Plone, ki temelji na tehnologiji Zope 2, velja za enega izmed najvarnejših med tovrstnimi odprtokodnimi sistemi [8, 9]**.**

- *omogoča pa tudi preprosto zagotavljanje kakovosti:*
  Razvoj sistema Zope poteka po načelu »netestirana koda je nedelujoča koda«. Nadzor kakovosti poteka s pomočjo več tisoč avtomatiziranih testov. Poleg teh mora vsaka nova funkcionalnost prestati tudi integracijske teste z že obstoječo kodo.

## 3.2 Kratek pregled zgodovine platforme Zope

Prva izdaja platforme Zope sega v leto 1998 in predstavlja naslednika sistemov Bobo in Principia, ki sta bila razvita za interne potrebe podjetja Digital Creations. Le-to se je kasneje preimenovalo v Zope Corporation. Naslednje leto je izšla druga verzija sistema, ki je aktivna še danes, predvsem zaradi dejstva, da na njem sloni Plone, sistem za upravljanje z vsebinami. Glavno prednost sistema Zope 2 je predstavljal administrativni vmesnik »*Zope Managemenent Interface*« (ZMI), s katerim lahko programiramo oz. povezujemo že obstoječe komponente preko spletnega vmesnika, napisana programska koda pa je shranjena v objektni bazi ZODB. To je sicer olajšalo delo začetnikom, imelo pa je tudi negativne učinke, kot so slabo načrtovani sistemi in težak prehod na programiranje na datotečnem sistemu.




*Slika 1:* Administrativni vmesnik ZMI (Zope Managemenent Interface) platforme Zope 2

Leta 2001 se je začel projekt Zope 3, katerega cilj je bil razdeliti preobsežne in medsebojno odvisne komponente na več manjših, predstavljal pa naj bi tudi odmik od programiranja preko spleta. Ker je Zope 2 postal preveč kompleksen za prilagajanje novi komponentni arhitekturi, so se razvijalci odločili za ponoven razvoj, v katerega so vključili posamezne uporabne dele in koncepte iz starega sistema. Tako sta se ohranila npr. objektna baza ZODB ter administrativni vmesnik ZMI, sicer v okrnjeni obliki, predvsem kot orodje za pregled objektne baze in povezav med komponentami, ne pa kot orodje za programiranje preko spleta. Od leta 2000 do 2003 je pri projektu Zope sodeloval tudi sam avtor programskega jezika Python, Guido van Rossum.

Komponentna arhitektura se je izkazala za pravilno odločitev, zato so želeli razvijalci nove pristope vključiti tudi v Zope 2. Pomembnejše knjižnice so prilagodili uporabi starejšega sistema v okviru projekta z imenom Five, poleg tega pa so se pojavila tudi nova ogrodja, v katere so začeli vključevati določene dele sistema Zope 3. Razvijalci so spoznali, da Zope 3 predstavlja tako aplikacijski strežnik kot tudi zbirko knjižnic, katero uporabljajo Zope 3 in ostala ogrodja, tako v celoti kot v delih. Zaradi tega so se osredotočili na podmnožico komponent projekta Zope 3, katere so znova uporabne v večini projektov. Poimenovali so jih *Zope Toolkit Library* (ZTK), celotno ogrodje Zope 3 pa so preimenovali v BlueBream.

## 3.3 Zbirka knjižnic ZTK

Zope toolkit library (ZTK) je zbirka knjižnic za pomoč pri gradnji spletnih aplikacij ali ogrodij. Predstavlja temelj ogrodij BlueBream, Grok, ter Zope 2, ki so del projekta Zope, nekatere knjižnice pa uporabljajo tudi druga ogrodja, npr. BFG. Glavna področja, ki jih ZTK pokriva, so naslednja [7]:

### 3.3.1 Komponentna arhitektura Zope (ZCA)

Komponentna arhitektura predstavlja načrtovalski vzorec, pri katerem se namesto navadnih objektov in večkratnega dedovanja**,** za nadzorovanje kompleksnosti uporabljajo komponente. Komponenta je objekt oz. povezana množica objektov z natančno definirano odgovornostjo, ki je opisana v obliki vmesnika. Vmesnik predstavlja seznam storitev, ki jih komponenta ponuja okolju**.** Objektu, ki to komponento uporablja, ni potrebno razumeti podrobnosti implementacije. Poleg tega lahko določeno komponento zamenjamo tudi z alternativno implementacijo, če le-ta vsebuje vsaj tiste metode, ki jih opisuje vmesnik [10]. Predstavlja temelj vseh ogrodij v okviru projekta Zope, zato bo podrobneje predstavljena v naslednjem poglavju.

### 3.3.2 Objektna baza Zope (ZODB)

Z objektno bazo ZODB (*Zope Object Database*) lahko programer objekte shranjuje skoraj popolnoma transparentno. Tudi realizacija hierarhičnih struktur objektov je enostavnejša kot pri relacijskih bazah, kar precej olajša gradnjo sistemov za urejanje vsebin. ZODB lahko uporabljamo v poljubnem okolju, ki temelji na programskem jeziku Python. Kljub temu, da je prevladujoči in priporočeni način realizacije trajnostnega mehanizma v ogrodjih Zope, pa velja omeniti, da je možna integracija tudi z ostalimi tipi podatkovnih baz, če to narekujejo zahteve. Značilnosti ter prednosti in slabosti baze ZODB bom opisal v posebnem poglavju.

### 3.3.3 HTML/XML predloge

Zope ima tudi lastno ogrodje za vnos dinamičnih vsebin v statične html strani, imenovano *Zope page Templates (ZPT).* Definira standarde TAL (angl. *Template Attribute Language*), TALES (angl. *TAL Expression Syntax*) in METAL (*Macro Expansion TAL*). Omogoča določanje spremenljivk, ponovitve stavkov in ostale preproste programske konstrukte, ki so v pomoč pri generiranju predstavitve, implementacijo kompleknejše poslovne logike v prikazni logiki pa namenoma otežuje, v nasprotju z npr. jezikom PHP. Predloga TAL je tudi veljaven dokument XHTML, saj so ukazi, ki so razloženi v posebnem imenskem prostoru, realizirani kot atributi na elementih HTML. To pomeni, da lahko izdelano predlogo grafični oblikovalec pregleduje v spletnem brskalniku ali urejevalniku WYSIWYG[1]. Če želimo, pa lahko v okviru ogrodij Zope uporabljamo tudi ostale sisteme za izdelavo dinamičnih predlog, kot so npr. Jinja, Genshi, Mako in drugi.

[1] What you see is what you get

### 3.3.4 Avtomatsko generiranje in validiranje form

Knjižnica *zope.formlib* omogoča avtomatsko generiranje ter validacijo vnosnih form na podlagi definirane sheme. Izdelava form z veliko vnosnimi polji je tako hitra in enostavna. Slabost knjižnice je pomanjkanje vizualno privlačnejših izbirnih in prikaznih elementov (angl. *widgets*). Pri izdelavi aplikacije sam tako moral sam izdelati element za izbiro datuma, ki uporablja knjižnico *jquery*.

### 3.3.5 Internacionalizacija

Splet je večjezično in multikulturno okolje, zato je pomembno, da uporabniku podamo informacije v njegovem jeziku in v lokalnem formatu (npr. oblika zapisa časa, datuma, števil*k*). Zope ponuja knjižnici za internacionalizacijo *i18n* ter lokalizacijo *l10n*, s katerima lahko aplikacijo na preprost način prilagajamo različnim okoljem. Za prevod v slovenščino sem pri izdelavi aplikacije knjižnico *i18n* uporabil tudi sam.

### 3.3.6 Varnost

Zope omogoča ločitev mehanizma za zagotavljanje varnosti in komponent, ki implementirajo določeno funkcionalnost sistema. Varnostni mehanizem je zelo prilagodljivo zasnovan. Sestavljen je iz številnih med sabo sodelujočih komponent, ki jih lahko prilagodimo glede na potrebe aplikacije in tako vplivamo na njegovo delovanje.

Avtentikacija uporabnikov je izvedena s pomočjo storitve *Pluggable Authentication Utility (PAU),* ki je razdeljena na dva osnovna dela:

- del za pridobitev prijavnih podatkov uporabnika
- del za avtentikacijo pridobljenih podatkov

Za vsak del lahko definiramo poljubno število vtičnikov (angl. *plugin)*, tako da je integracija različnih načinov avtentikacije zelo preprosta.

Avtorizacijo opravi ogrodje pri obhodu drevesa ogrodje za vsak objekt posebej, glede na definirano varnostno politiko.

Varnostno politiko sestavljajo[11]:

- uporabniki (angl. *principals)*, ki se lahko prijavijo v sistem
- skupine uporabnikov (angl. *groups)*, ki lahko vsebujejo poljubno število uporabnikov, uporabnik pa je lahko član poljubnega števila skupin
- pravice (angl. *permissions)*, ki uporabnikom omogočajo dostop do različnih delov aplikacije
- vloge (angl. *roles)*, ki predstavljajo množico pravic. Vloge priredimo uporabnikom ali skupinam.

### 3.3.7 Indeksiranje in iskanje

Objektna baza ZODB ne vsebuje indeksiranja in poizvedovalnih mehanizmov. Te naloge so prestavljene na aplikacijsko raven. V ta namen obstajata knjižnici *zope.index* ter *zope.catalog*. Imamo več tipov indeksov, med drugim tudi indeks, ki omogoča iskanje po

celotnem tekstu (angl. *full-text search*)[2]. Poleg tega je na voljo komponenta, imenovana katalog, s katero lahko izvajamo poizvedbe s pomočjo predhodno definiranih indeksov.

### 3.3.8 Testiranje

Zope ima, podobno kot ostali projekti v skupnosti Python, zelo razvito kulturo testiranja. Obstajata dve osnovni obliki testov: testi enot, ki jih poznajo tudi ostali programski jeziki, ter dokumentacijski testi (t.i. *Doctests*), ki so značilni samo za Python.

Dokumentacijski testi so tekstovne datoteke s kombinacijo dokumentacije in ukazov, ki so videti kot interaktivna seja v lupini Python. Programer tako testira funkcionalnost komponente in hkrati dokumentira njeno uporabo. Po mojem mnenju je ta pridobitev dvorezen meč, saj nekaterim programerjem zmanjka moči za izdelavo obsežnejše in uporabniku prijaznejše dokumentacije.

S pomočjo testov enot preverjamo pravilnost delovanja posamezne programske komponente neodvisno od ostalih enot sistema. Dokumentacijski testi so praviloma namenjeni funkcionalnem testiranju, ki se osredotoča na uporabniški vidik, torej ali programska oprema podpira uporabniške zahteve ali ne.

Knjižnica *zope.testing* vsebuje podporo za kreiranje in zagon množice dokumentacijskih testov ter testov enot, poleg tega pa tudi nekatere dodatne komponente za pripravo testnega okolja.

Obstaja tudi knjižnica *zope.testbrowser*, ki sicer ni del zbirke ZTK, je pa zelo uporabna za funkcijsko in sistemsko testiranje, ki vključuje interakcijo uporabnika s sistemom. Vsebuje programske komponente, s katerimi lahko simuliramo delovanje spletnega brskalnika. Tako se lahko s pisanjem ukazov »sprehajamo« po spletni strani, preverjamo vsebino strani, vnašamo podatke v vnosne forme itd., torej simuliramo obnašanje uporabnika brez napornega ročnega preverjanja pravilnosti delovanja. Pisanje tovrstnih testov lahko poenostavimo s knjižnico *zope.testrecorder*, s katero posnamemo določeno interakcijo s sistemom preko spletnega brskalnika. Nato posnetek v obliki dokumentacijskega testa ali testa Selenium izvozimo, s čimer je zagotovljena ponovljivost testiranja. *Selenium* je ogrodje za funkcionalno testiranje, ki v primerjavi s knjižnico *zope.testbrowser* ne simulira brskalnika, ampak izvaja testiranje v pravem brskalniku, kar nam omogoča testiranje tudi kode javascript.

---

2 Podobno funkcionalnost ponuja recimo tudi knjižnica *Apache Lucene*.

```
Getting HTML
============

Having defined form fields, we can use them to generate HTML
forms. Typically, this is done at run time by form class
instances. Let's look at an example that displays some input widgets:

    >>> class MyForm:
    ...     form_fields = form.Fields(IOrder, omit_readonly=True)
    ...
    ...     def __init__(self, context, request):
    ...         self.context, self.request = context, request
    ...
    ...     def __call__(self, ignore_request=False):
    ...         widgets = form.setUpWidgets(
    ...             self.form_fields, 'form', self.context, self.request,
    ...             ignore_request=ignore_request)
    ...         return '\n'.join([w() for w in widgets])

Here we used ``form.setUpWidgets`` to create widget instances from our
form-field specifications.  The second argument to ``setUpWidgets`` is a
form prefix.  All of the widgets on this form are given the same
prefix.  This allows multiple forms to be used within a single form
tag, assuming that each form uses a different form prefix.

Now, we can display the form:

    >>> from zope.publisher.browser import TestRequest
    >>> request = TestRequest()
    >>> print MyForm(None, request)() # doctest: +NORMALIZE_WHITESPACE
    <input class="textType" id="form.name" name="form.name" size="20"
           type="text" value=""  />
    <input class="textType" id="form.min_size" name="form.min_size" size="10"
           type="text" value=""  />
    <input class="textType" id="form.max_size" name="form.max_size" size="10"
           type="text" value=""  />
    <input class="textType" id="form.color" name="form.color" size="20"
           type="text" value=""  />
```

*Slika 2: Primer dokumentacijskega testa iz knjižnice zope.formlib*

## 3.4 Ogrodja, ki uporabljajo ZTK

Ogrodja, ki uporabljajo ZTK in ZCA, so naslednja [12]:

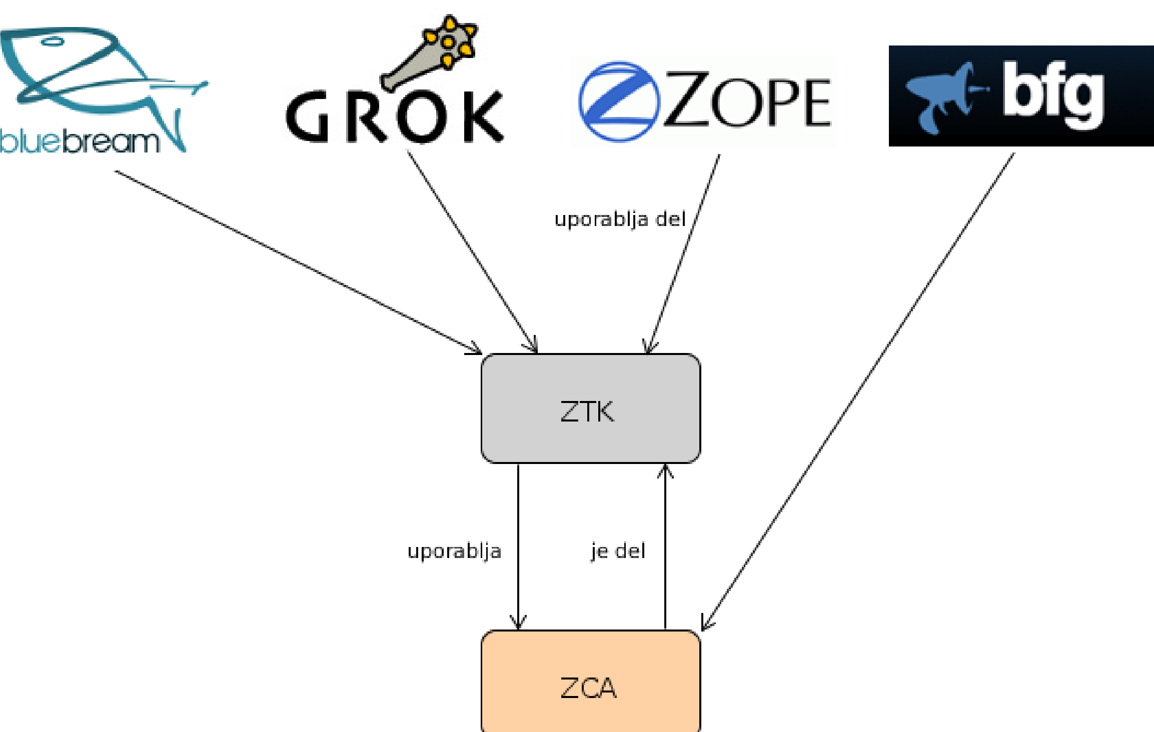
- *BlueBream (nekoč poznan pod imenom Zope 3):*
  BlueBream je ogrodje, ki temelji na celotni zbirki ZTK, vključno z aplikacijskim strežnikom, torej vsebuje vse potrebno za razvoj aplikacij. Uporablja ZCML, ki je na XML-ju temelječ jezik za konfiguracijo komponent.

- *Grok:*
  Grok se je pojavil kot odgovor na kritike prevelike kompleksnosti sistema Zope 3. Poudarja načelo »konvencije naj imajo prednost pred konfiguracijo«, ki zmanjšuje količino konfiguracijske kode, katero je potrebno napisati. Za

konfiguracijo komponent ne uporablja jezika ZCML, ampak posebne ukaze (angl. *grok directives*) in dekoratorje v sami kodi. Rezultat je agilno ogrodje, ki omogoča hiter razvoj aplikacij, s prilagodljivosto komponentne arhitekture. Grok je v večini primerov tudi združljiv z navadnimi Zope 3 komponentami.

- *Zope2*
  Zope 2 je starejša verzija sistema Zope, ki je še vedno aktivno v rabi predvsem zato, ker na njem sloni priljubljen sistem za upravljanje z vsebinami Plone. V zadnjih letih je prevzel tudi veliko konceptov in knjižnic novejše verzije, tako je med njima čedalje manjša razlika.

- *Ogrodje BFG*
  BFG je minimalistično ogrodje za bolj izkušene razvijalce. Uporabniku namreč ponuja samo osnovni nabor komponent, ostale mora izbrati sam. Temelji na tehnologiji ZCA.




*Slika 3: Ogrodja, ki uporabljajo Zope Toolkit Library (ZTK)*

# 4. Komponentna arhitektura Zope (ZCA)

Komponenta v širokem pomenu besede predstavlja modul, knjižnico ali spletno storitev, ki vsebuje množico funkcij ali podatkov. Komponente komunicirajo med sabo preko vmesnikov, ki opisujejo njene storitve, katere ponuja zunanjemu svetu. Vmesnik predstavlja pogodbo med komponento in uporabnikom, ki se lahko poslužuje z njenimi storitvami brez poznavanja njenega notranjega delovanja [13].

Komponentna arhitektura Zope je ogrodje za podporo komponentnemu razvoju v programskem jeziku Python. Sestavljajo ga naslednje knjižnice [10]:

- *zope.component*: za kreiranje, registracijo in iskanje komponent
- *zope.interface*: za definiranje vmesnikov komponent
- *zope.event*: preprost sistem za upravljanje z dogodki

Omogoča izdelavo prilagodljivih in razširljivih sistemov z uporabo komponent, ki so osnovni gradniki sistema. Za upravljanje komponent uporablja poseben *register komponent*, ki omogoča spremembe v implementaciji s pomočjo konfiguracije. To prinaša naslednji prednosti:

- možnost preklapljanja med implementacijami brez posegov v izvorno kodo;
- dodajanje nove funkcionalnosti brez posegov v izvorno kodo.

Posledica je arhitektura, ki je prilagodljiva ter skalabilna, izboljšana je tudi ponovna uporaba komponent.

*Slika 4: ZCA kot register za iskanje storitev*

## 4.1 Komponente ZCA

Pri ZCA so komponente objekti z definiranimi vmesniki, ki imajo introspekcijske značilnosti. Glavni ideji sta upoštevanje načela »programiraj z vmesniki, ne z implementacijo« ter nadzorovanje kompleksnosti s pomočjo komponent in delegacije namesto večkratnega dedovanja [14].

Poznamo dva osnovna tipa komponent:

- *pretvornik (*angl. *adapter):*
  Pretvorniki prilagodijo med seboj nekompatibilne vmesnike ali razširjajo funkcionalnost določene komponente, npr. komponento modela nadgradijo s predstavitveno funkcijo ali procesiranjem podatkov.

- *storitev (*angl. *utility):*
  Storitve so komponente, ki v nasprotju s pretvorniki ne delujejo v kontekstu neke druge komponente, ampak ponujajo splošne storitve, kot so recimo indeksiranje in iskanje objektov, avtentikacijo itd.

## 4.2 Vmesniki

### 4.2.1 Vmesniki v programskem jeziku Python

Nekateri programski jeziki, kot sta Java ali C#, ponujajo formalno definiranje vmesnikov, pri jeziku Python pa so definirani implicitno s pomočjo dokumentacije in protokola uporabe. Python je dinamično tipiziran jezik. Če želimo uporabiti nek objekt, ni potrebno določiti njegovega tipa, ampak ga samo uporabimo na željeni način v danem kontekstu. Naj kot primer navedem razred StringIO, ki realizira določeno podmnožico vmesnika razreda File, vendar ne uporablja datoteke, temveč del glavnega pomnilnika. Vsaki metodi, ki pričakuje odprto datoteko, lahko namesto primerka razreda File podamo StringIO. Le-ta ne implementira istega vmesnika kot razred File, ampak samo metode, ki so potrebne v danem kontekstu [15].

Omenjeni princip se imenuje »račje tipiziranje« (angl. *duck-typing*), torej: »Če izgleda kot raca in se oglaša kot raca, potem je ta ptič raca.« [16] V jeziku objektov to pomeni: Če ima objekt tiste metode in atribute, ki jih v določenem primeru potrebujemo, potem ga lahko uporabimo, čeprav mogoče v resnici sploh ni »pravega« tipa. Ta lastnost nam torej omogoča polimorfizem brez dedovanja. Njena slabost je možnost napak ob klicu metod z argumenti napačnega tipa ali sprememb v definiciji razredov. V praksi se zaradi poudarka na testiranju to redko dogaja, na voljo pa imamo tudi metode za preverjanje, ali objekt podpira določen protokol ali ne [17]. Najpogostejši način preverjanja je uporaba metod:

- *isinstance (object, class_or_type)*, s katero lahko ugotovimo, če je objekt ustreznega tipa oz. primerek določenega razreda, ter

- *hasattr (object, name),* s katero lahko preverimo, če objekt vsebuje določen atribut oz. metodo.

Vendar omenjeni metodi nista stoodstotno zanesljivi, zato so se pojavile pobude, da bi način preverjanja standardizirali. Organizacija Zope je že nekajkrat predlagala uporabo vmesnikov, a njihovi predlogi niso bili upoštevani. Veliko programerjev je mnenja, da formalno definiranje vmesnikov ni potrebno, saj Python podpira večkratno dedovanje, poleg tega pa strog formalizem ne ustreza dinamični in odprti naravi jezika [18].

### 4.2.2Vmesniki Zope

Po neuspešnem poskusu standardizacije vmesnikov so morali v skupnosti Zope poiskati alternativno rešitev. Rezultat njihovega dela je knjižnica *zope.interface*, ki omogoča definiranje vmesnikov, ponuja pa tudi vrsto uporabnih metod. Potrebno je poudariti, da primarni namen uporabe vmesnikov v ZCA ni preverjanje, če objekti določen protokol dejansko podpirajo, kljub temu da imamo na voljo metode, s katerimi lahko to dosežemo.

Njihova glavna prednost je ta, da nam omogočajo razvoj prilagodljivih sistemov, kjer sodelujoče komponente niso odvisne od specifične implementacije. Komponente so zbrane v posebnem registru, vmesnike pa uporabljamo za registracijo in iskanje storitev, ki jih v danem trenutku potrebujemo. To nam omogoča, da na preprost način razširimo ali zamenjamo določeno funkcionalnost aplikacije [19].

Vmesniki predstavljajo za programerje tudi priročno dokumentacijo, saj imamo na enem mestu[3] zbran opis storitev celotnega sistema, brez motečih implementacijskih podrobnosti.

V nadaljevanju sledi primer uporabe vmesnikov s pomočjo knjižnice zope.interface.

#### 4.2.2.1 Definiranje in uporaba vmesnikov Zope

```
from zope.interface import Interface,Attribute,invariant,Invalid


class IUporabnik(Interface):
    """Predstavlja registriranega uporabnika sistema."""

    ime = Attribute("Ime")
    priimek = Attribute("Priimek")
    geslo = Attribute("Geslo")

    def nastaviGeslo(geslo,tip_kodiranja):
        """Shrani kodirano uporabnikovo geslo.

           Parametri:

           geslo - uporabnikovo geslo
           tip_kodiranja - niz, ki označuje tip
                   kodiranja(npr. 'SHA1','MD5',..)
```

3 Ponavadi so vmesniki zbrani v datoteki interfaces.py oz. – v primeru večjega števila – v posameznih modulih v mapi interfaces.

```
    """

  def preveriGeslo(geslo):
    """Preveri uporabnikovo geslo.

      Parametri:

      geslo - uporabnikovo geslo

      Vrni 'True', če je geslo pravilno,
      v nasprotnem primeru 'False'.
    """

  @invariant
    def gesloUstrezno(uporabnik):
      if len(uporabnik.geslo)<6:
        raise Invalid("Geslo mora biti dolgo vsaj 6 znakov!")
```

*Primer 1: Definiranje vmesnika s pomočjo knjižnice* zope.interface

Vmesnike po konvenciji pišemo z veliko črko »I« na začetku. Ker niso sestavni del jezika, je za njihovo uporabo potrebno razširjati razred *zope.interface.Interface*. Tudi njih lahko, podobno kot navadne razrede, uredimo v hierarhije. Atribute definiramo s pomočjo razreda *zope.interface.Attribute*, pri metodah pa izpustimo parameter *self*, saj nikoli ne ustvarimo primerka razreda.

Atribute lahko definiramo tudi malo drugače. Z uporabo knjižnice *zope.schema* namreč podamo dodatne informacije, na podlagi katerih lahko s knjižnicami za delo z obrazci[4] avtomatsko generiramo in preverjamo pravilnost vnosnih obrazcev. Z oznako @*invariant* pa lahko izrazimo kompleksnejše omejitve, ki morajo veljati za objekte, ki realizirajo vmesnik.

```
ime = schema.TextLine(title=u"Ime")
priimek = schema.TextLine(title=u"Priimek",required=False)
geslo = schema.Password(title=u"Geslo")
```

*Primer 2: Definicija atributov v vmesniku z uporabo knjižnice zope.schema*

Razred, katerega primerki realizirajo definirani vmesnik, mora vsebovati klic metode implements(). Implementiranih vmesnikov je lahko več.

```
from zope.interface import implements

class Uporabnik(object):
  implements(IUporabnik)
```

4 zope.formlib, z3c.forms, zeam.form

```
#...sledi implementacija vmesnika
```

*Primer 3: Implementacija vmesnika*

Z metodama implementedBy ter providedBy lahko preverimo, če razred oz. primerek razreda realizira definirani vmesnik:

```
>>>IUporabnik.implementedBy(Uporabnik)
True

>>>uporabnik = Uporabnik('jim','fulton','geslo')
>>>IUporabnik.providedBy(uporabnik)
True
```

*Primer 4: Prikaz uporabe metod implementedBy ter providedBy*

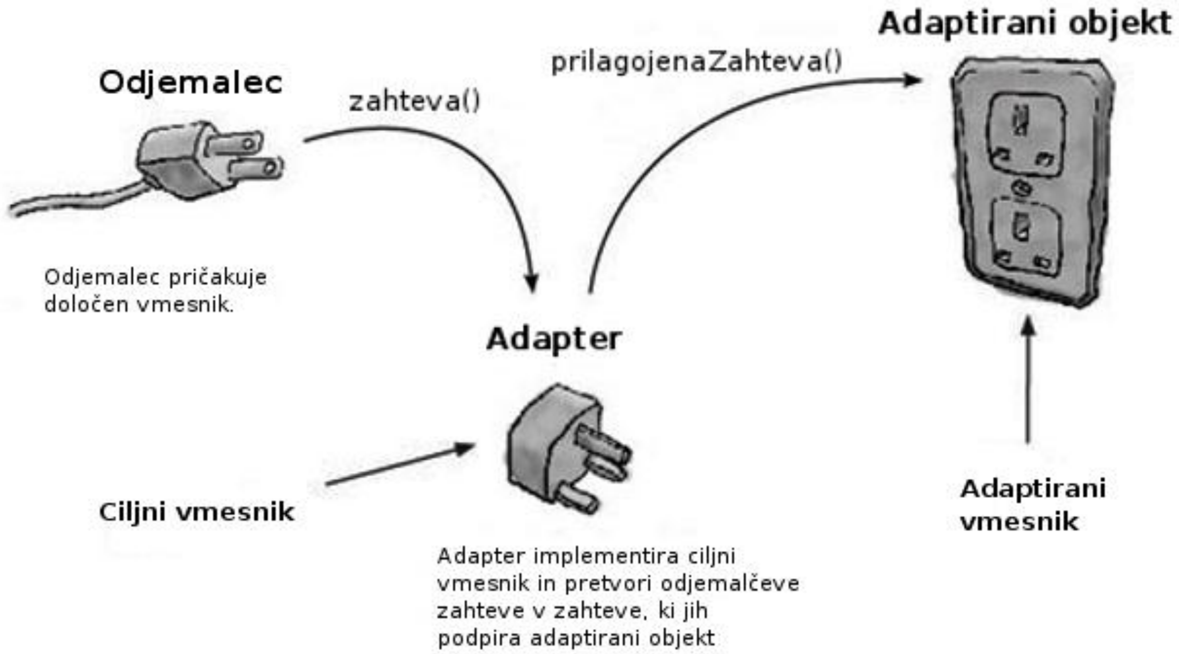
Omenjeni metodi uporabljamo za selektivno izvajanje akcij glede na tip objekta. Pozitiven rezultat metod ne zagotavlja, da razred oz. objekt spoštuje pogodbo, definirano v vmesniku, ampak samo, da je razred oz. objekt deklariral implementacijo vmesnika. Če želimo dobiti zagotovilo, da razred oz. objekt vmesnik tudi v resnici realizira, lahko uporabimo metodi verifyClass ter verifyObject. To običajno storimo v aktivnosti testiranja.

Če želimo, lahko že ustvarjenim objektom z ukazom alsoProvides() določimo dodatne vmesnike, ki jih ti realizirajo. To je koristno predvsem pri uporabi vmesnikov, imenovanih označevalni vmesniki (*angl. marker interfaces*). Ti vmesniki nimajo definiranih nobenih atributov ali metod, ampak se uporabljajo samo za označevanje tipa objektov, kar nam omogoča selektivno obravnavo objektov. Tako bi lahko definirali vmesnik INaprodaj, s katerim bi označili objekte, za katere želimo omogočiti nakup, in nato za vse takšne objekte uporabniku prikazali neke dodatne možnosti.

## 4.3 Pretvorniki

### 4.3.1 Pretvorniki kot načrtovalski vzorec

Pretvornik omogoča sodelovanje med komponentami, ki so v osnovi nekompatibilne. Pretvornik implementira vmesnik, ki ga odjemalec potrebuje. Vse njegove zahteve ustrezno pretvori v klice metod prilagojenega objekta [20]. Tako lahko s pomočjo pretvornikov ponovno uporabimo komponente brez sprememb izvorne kode.




*Slika 5: Prikaz načrtovalskega vzorca »Pretvornik«*

Poznamo dva načina realizacije pretvorbe [21]:

- *razredna pretvorba (angl. class adaptation):*
  Ta način pretvorbe dosežemo s pomočjo večkratnega dedovanja: pretvornik deduje od odjemalca in od prilagojenega razreda.

- *objektna pretvorba (angl. object adaptation):*
  V tem primeru pretvorbo implementiramo s pomočjo delegacije. Pretvornik vsebuje primerek prilagojenega razreda in mu posreduje odjemalčeve zahteve.

Vsak način ima določene prednosti in slabosti. Python sicer podpira večkratno dedovanje, vendar so pretvorniki pri komponentni arhitekturi Zope realizirani s pomočjo delegacije zaradi upoštevanja načela »Kompozicija objektov naj ima prednost pred dedovanjem razredov.« [10, 22]

### 4.3.2 Pretvorniki ZCA

Pretvornik kot načrtovalski vzorec se v splošnem uporablja predvsem za povezovanje nekompatibilnih vmesnikov, pretvorniki ZCA pa imajo širši namen - uporabljajo se tudi za razširjanje funkcionalnosti objektov[5]. Pri platformi Zope 2 je bila pogosta uporaba razredov za dodajanje funkcionalnosti glavnim razredom (*angl. Mix-in classes*). Posledice nekontrolirane uporabe večkratnega dedovanja so bile kompleksne in neobvladljive hierarhije razredov. Z uporabo delegacije lahko osnovne razrede ohranimo majhne z natančno definirano funkcionalnostjo, kar močno prispeva k prilagodljivosti sistema [22].

---

5 Prvotno so bili pretvorniki celo poimenovani z besedo "Feature"

Glavna prednost pretvornikov ZCA je register pretvornikov (*angl. adapter registry),* ki za podani vmesnik in kontekst pretvorbe poišče najustreznejši pretvornik. To pomeni, da lahko definiramo neko splošno privzeto obnašanje sistema, posamezni deli aplikacije pa imajo možnost to obnašanje prilagoditi.

#### 4.3.2.1 Uporaba pretvornikov ZCA

Za uporabo pretvornikov ZCA so potrebni naslednji koraki:

- *definiranje vmesnika,*
- *implementacija,*
- *registracija* in
- *iskanje ustreznega pretvornika.*

Oglejmo si jih podrobneje.

##### 4.3.2.1.1 Definiranje vmesnika

Pri komponentni arhitekturi so pomembne natančno definirane komponente. To dosežemo z uporabo vmesnikov, ki predstavljajo pogodbo za zagotavljanje opisane funkcionalnosti. Vmesniki so tudi enolični identifikator komponente v komponentnem registru in podlaga za iskanje ustrezne implementacije.

##### 4.3.2.1.2 Implementacija

Kot smo že omenili, Zope uporablja objektno pretvorbo. To pomeni, da mora naša implementacija pretvornika vsebovati primerek prilagojenega objekta, ki ga podamo kot argument konstruktorju. Poleg tega moramo določiti:

- kateri vmesnik implementira pretvornik:

  To storimo s pomočjo metode ***implements*** iz knjižnice zope.interface. V primeru, da pretvornik realizira več vmesnikov, je potrebno pri njegovi registraciji z ukazom ***provides*** določiti, za kateri od vmesnikov ga želimo registrirati.

- kontekst pretvorbe:

  Tipe objektov, ki jih pretvornik prilagodi, določimo s pomočjo metode ***adapts*** iz knjižnice *zope.component*. Kot parameter metodi ponavadi podamo vmesnik prilagojenega objekta, lahko pa tudi konkretni razred. Prilagojenih objektov je lahko več, v tem primeru pretvornik imenujemo večkratni pretvornik (angl. *multi-adapter*). V ogrodjih Zope 2, BlueBream ter Grok je to zelo pogosta oblika, saj je pogled (oz. krmilnik v klasičnem pojmovanju vzorca model-pogled-krmilnik[6]) implementiran kot pretvornik zahteve in modela.

---

6 V skupnosti Zope ter Django imajo malo drugačno interpretacijo vzorca model-pogled-krmilnik, več o tem v poglavju 6.2.4.2

#### 4.3.2.1.2.1 Primer implementacije

Za primer implementacije sem izbral pretvornik, ki se uporablja v ogrodjih BlueBream ter Grok. Poznamo dva osnovna tipa podatkovnih entitet: entitete, ki lahko vsebujejo druge objekte (imenujemo jih vsebniki), ter navadne entitete. Vsebniki so pravzaprav nekoliko naprednejši slovarji, zato mora imeti vsak vsebovani objekt unikatno ime oz. ključ, s katerim ga lahko poiščemo. Pri izbiri imena si pomagamo s pretvornikom, ki realizira vmesnik INameChooser. Lahko se odločimo, da bomo naše uporabnike shranjevali pod zaporedno številko, s kombinacijo imena in priimka, itd. Upoštevati je potrebno, da je ta ključ v primeru objave objekta tudi del naslova url, zato je zaželeno, da izbrano ime vsebuje samo osnovne znake.

```
from zope.component import adapts
from zope.interface import implements
from zope.container.interfaces import IContainer
from zope.container.interfaces import INameChooser

class NameChooser(object):
    """ Določi unikatno ime za objekt v
      danem vsebniku."""
    adapts(IContainer)
    implements(INameChooser)

    def __init__(self, context):
        self.context = context

    def chooseName(self, name, object):
        # sledi implementacija metode

    def checkName(self, name, object):
        # sledi implementacija metode
```

*Primer 5: Realizacija pretvornika za izbiro unikatnega imena v vsebniku*

### 4.3.2.1.3 Registracija

Definirani pretvornik je potrebno registrirati v registru komponent. Pri testiranju ali uporabi ZCA v samostojnih projektih, izven platforme Zope, uporabljamo metodo provideAdapter iz knjižnice zope.component:

- *provideAdapter* (factory, adapts=None, provides=None, name="")

Parameter factory predstavlja objekt, ki ustvari primerek pretvornika. Ponavadi je to razred, lahko pa je tudi metoda. V našem primeru bi pretvornik registrirali z ukazom provideAdapter(NameChooser), parametra adapts ter provides pa ni potrebno vnesti, saj se potrebne definicije nahajajo že v samem razredu. Če bi naš pretvornik implementiral več vmesnikov, bi bilo potrebno s parametrom provides določiti, za katerega ga želimo registrirati.

V primeru, ko potrebujemo več enakovrednih pretvornikov, tj. pretvornikov, ki

implementirajo isti vmesnik ter prilagodijo iste tipe objektov, jih moramo registrirati z edinstvenim imenom. Tako vrsto pretvornikov imenujemo »poimenovani pretvorniki« (angl. *named adapters*).

#### *4.3.2.1.3.1 Registracija z uporabo jezika ZCML*

Če uporabljamo ZCA v okviru platforme Zope, komponente registriramo z uporabo posebnega jezika ZCML (angl. *Zope Configuration Markup Language*), ki temelji na jeziku XML. Registracija našega pretvornika v datoteki *configure.zcml* bi izgledala tako:

```
<configure xmlns="http://namespaces.zope.org/zope">
  <adapter factory=".modul.NameChooser" />
</configure>
```

Tudi ostale parametre lahko namesto v razredu definiramo v konfiguracijski datoteki:

```
<adapter
   factory=".modul.NameChooser"
   for="zope.container.interfaces.IContainer"
   provides="zope.container.interfaces.INameChooser"
   />
```

*Primer 6: Primer registracije pretvornika z uporabo jezika ZCML*

Z ustrezno konfiguracijo lahko brez spreminjanja izvorne kode zamenjamo implementacijo komponente ali pa jo popolnoma onemogočimo. To je zelo močno orodje, ki nam omogoča veliko prilagodljivost in ponovno uporabo komponent. Pri realizaciji aplikacije sem za vključitev komentarjev uporabil že obstoječo komponento, ki pa je vsebovala neprimerne poglede. Potrebno je bilo zgolj ustvariti in registrirati nov pogled za pravi vmesnik, ogrodje pa je poskrbelo za ustrezen prikaz komentarjev z alternativno realizacijo.

Možnost zamenjave določene implementacije je zelo koristna tudi pri testiranju. Kot primer naj navedem testiranje pošiljanja elektronske pošte. Namesto privzete storitve za pošiljanje lahko registriramo alternativno realizacijo, ki implementira isti vmesnik IMailDelivery. Tako smo lahko poslano sporočilo namesto dejanskega pošiljanja samo izpisovali na zaslon oz. programsko preverjali, če se vsebina ujema s pričakovano.

#### *4.3.2.1.3.2 Registracija z ukazi grok (angl. grok directives)*

Posledica te prilagodljivosti je tudi povečana kompleksnost, saj je potrebno preklapljati med dvema domenama – med programsko kode Python ter med jezikom ZCML.

Cilj ogrodja Grok je olajšati uporabo komponentne arhitekture z definiranjem uporabnega privzetega obnašanja, kar zmanjša obseg potrebne konfiguracije. Le-to pa lahko namesto v jeziku zcml definiramo v implementaciji komponente s posebnimi ukazi. Ogrodje ob zagonu strežnika pregleda našo izvorno kodo in poskrbi za ustrezno registracijo.

```
class NameChooser(grok.Adapter):
   grok.provides(INameChooser)
   grok.context(IContainer)

   def chooseName(self, name, object):
      # sledi implementacija metode

   def checkName(self, name, object):
      # sledi implementacija metode
```

*Primer 7: Implementacija in registracija pretvornika z uporabo ogrodja Grok*

Komponento *martian*, ki pregleda kodo in poskrbi za registracijo, lahko uporabljamo neodvisno od ostalih komponent ogrodja Grok, npr. pri razvoju ogrodja BFG ali celo na platformi Zope 2.
Pri tem nismo omejeni samo na standardne ukaze za registracijo komponent, ampak lahko definiramo tudi lastne. To je zelo koristno pri izdelavi ponovno uporabljivih komponent, saj s tem močno olajšamo uporabo. Potrebno je napisati komponento, ki definira ukaz in izvede potrebne konfiguracijske korake. Nato lahko celotno konfiguracijo opravimo s klicem tega ukaza v naši aplikaciji.

Kot primer naj navedem uporabo komponente *megrok.login* za avtentikacijo in registracijo. V definiciji aplikacijskega razreda z ukazom megrok.login.enable() omogočimo sejno avtentikacijo namesto privzete osnovne avtentikacije (angl. *Basic Authentication*). Ukaz poskrbi za namestitev in konfiguracijo komponente, registrira pa tudi preprost prijavni obrazec 'login' s polji za vnos uporabniškega imena in gesla. Na voljo imamo tudi ukaza *megrok.login.viewname,* s katerim določimo, kateri pogled naj se uporabi za prikaz prijavnega obrazca, ter ukaz *megrok.login.autoregister,* s katerim omogočimo samostojno registracijo uporabnikov.

#### 4.3.2.1.4 Iskanje ustreznega pretvornika

Da bi lahko pretvornik uporabili, ga je potrebno najprej poiskati v registru komponent. V primeru registracije pretvornika brez imena in samo za en tip objektov, lahko to storimo preprosto s klicem vmesnika. Ta kot parameter sprejme objekt, katerega želimo prilagoditi.

```
>>>INameChooser(container).chooseName(name,object)
```

Primer: Klic neimenovanega pretvornika

V nasprotnem primeru moramo uporabiti metode iz knjižnice *zope.component*:

- *getAdapter* ali *querryAdapter*[7] za iskanje poimenovanega pretvornika;
- *getMultiAdapter* za iskanje pretvornika, ki prilagodi več objektov.

Iskanje ustreznega pretvornika poteka tako, da ZCA preveri, katere vmesnike realizira prilagojeni objekt in poskuša najti ustrezen pretvornik z želenim vmesnikom. Pogosto prilagojeni objekt realizira več vmesnikov: poleg tistih, ki jih je definiral njegov razred,

7 Metodi sta podobni, razlika je le v tem, da getAdapter v primeru neuspešne prilagoditve sproži izjemo, querryAdapter pa vrne None.

podeduje tudi vmesnike nadrazredov. Kot smo videli, pa lahko tudi že ustvarjenemu objektu priredimo izbran vmesnik (npr. označevalni vmesnik). Zato je potreben nek mehanizem, ki med vsemi realizacijami vmesnikov izbere najbolj specifičnega. Ta mehanizem se pri ZCA imenuje IRO (angl. *interface resolution order).* Določanje prioritete je podobno kot pri polimorfizmu oz. pri izbiri ustrezne metode pri hierarhiji objektov. Pravila IRO so naslednja[23]:

- pretvornik, registriran za razred, ima prednost pred pretvornikom, registriranim za vmesnik;
- vmesnik, ki ga realizira objekt neposredno (npr. označevalni vmesnik), ima prednost pred vmesnikom, ki ga realizira njegov razred;
- vmesnik, ki je v ukazu *implements()* naveden prej, ima prednost pred vmesniki, ki sledijo;
- vmesnik, ki ga realizira razred, ima prednost pred vmesnikom, ki ga realizira osnovni razred (razred od katerega deduje);
- vmesnik ima prednost pred osnovnim vmesnikom (vmesnik, od katerega deduje);
- itd.

Posledica opisanih pravil je, da lahko splošen pretvornik registriramo za splošni vmesnik, nato pa definiramo več specifičnih pretvornikov in bomo vedno dobili ustreznega v danem kontekstu. V primeru zgoraj sem definiral pretvornik NameChooser za splošen vmesnik IContainer. Lahko bi definiral še dodaten pretvornik, npr. za izbiro imena novic, ki bi naslovu novice dodal trenutni datum, registrirali pa bi ga za vmesnik INewsContainer.

```
class INewsContainer(IContainer):
   """Vsebnik za novice"""

class NewsNameChooser(grok.Adapter):
   """Določ unikatno ime novice"""
   grok.provides(INameChooser)
   grok.context(INewsContainer)

   # sledi implementacija...
```

*Primer 8: Definicija in registracija pretvornika za izbiro unikatnega imena novice*

Ob klicu INameChooser(news_container) bi komponentna arhitektura vrnila primerek našega novega pretvornika, saj je vmesnik INewsContainer bolj specifičen od vmesnika IContainer.

#### 4.3.2.2 Prednosti uporabe pretvornikov ZCA [24-26]:

- Osnovni objekti so majhni in preprosti. Dodajanje nove funkcionalnosti pomeni samo kreiranje novih pretvornikov. Python sicer podpira večkratno dedovanje, tako da lahko podoben rezultat dosežemo tudi z razširjanjem dodatnega osnovnega razreda, vendar je vzdrževanje tega načina težje, saj so hierarhije objektov kompleksnejše.

- Z uporabo registra pretvornikov dosežemo visoko stopnjo ponovne uporabljivosti in prilagodljivosti. Po navadi imamo nek splošen pretvornik, ki definira neko privzeto

obnašanje, registriran pa je za zelo splošen tip objektov. Uporabniška programska koda nato definira bolj specifičnega glede na potrebe aplikacije in tako privzeto obnašanje ustrezno prilagodi.

## 4.4 Storitve ZCA

V registru komponent prevladujejo pretvorniki, ki razširjajo funkcionalnost prilagojenih objektov, vendar je včasih koristno definirati tudi komponente, ki ponujajo splošne storitve, neodvisno od konteksta. Definiranje in registracija te vrste komponent sta podobna kot pri pretvornikih, le da v tem primeru nimamo prilagojenega objekta. Majhna razlika je tudi v načinu iskanja storitve v registru, saj nimamo na voljo priročne sintakse kot v primeru neimenovanih pretvornikov, ampak moramo vedno uporabiti metodo *getUtility*. Storitve so podobno kot pretvorniki registrirani za določen vmesnik, kar nam omogoča preprosto zamenjavo implementacije.

### 4.4.1 Delitev storitev ZCA

Poznamo dva osnovna tipa storitev [27]:

- *singleton* :
  V tem primeru imamo en primerek storitve za celotno aplikacijo. To so storitve za povezavo s podatkovno bazo, indeksiranje, iskanje, pošiljanje e-pošte, enkripcijo itd..

- *poimenovana storitev*:
  Lahko imamo več primerkov storitve iste vrste, ki so registrirani pod različnim imenom. Tipični primeri uporabe so pravice in vloge(angl. roles and permissions), preobleke(angl. skins), prevajalske domene itd. .

Pri uporabi ZCA v okviru platforme Zope poznamo še eno delitev storitev:

- *globalna storitev:*
  Globalne storitve so ustvarjene in registrirane v globalnem registru komponent ob zagonu aplikacije, na voljo pa so celotni aplikaciji.

- *lokalna storitev:*
  V sistemu Zope poznamo koncept *strani (*angl. *sites)*, ki je s svojim lokalnim registrom komponent ločen del aplikacije, dostop pa ima tudi do globalnega registra, pri čemer imajo lokalne komponente prednost pred globalnimi. Storitve te vrste so registrirane v lokalnem registru, od globalnih pa se razlikujejo tudi po tem, da je njihovo stanje shranjeno v podatkovni bazi. Primer tovrstne storitve je npr. storitev za poizvedbe zope.catalog.

### 4.4.2 Tovarne

Posebna vrsta storitev so tovarne (angl. *factories*). Kot pri ostalih vrstah komponent imamo tudi pri razredih podatkovne plasti možnost zamenjave implementacije. Brez uporabe tovarn mora programska koda, ki ustvari nov podatkovni objekt določenega tipa, uvoziti ustrezni razred, kar onemogoči preprosto zamenjavo implementacije. Problem lahko rešimo z uporabo tovarn, ki so posrednik, kateri poskrbi za ustvarjanje objektov. Tako lahko določeno podatkovno komponento zamenjamo, priskrbeti moramo le nadomestno tovarno, ki uporabi našo implementacijo [27].

Oglejmo si primer uporabe tovarne, ki ustvari novega uporabnika:

- *definiranje tovarne:*
  Konstruktorju razreda Factory moramo podati naš implementacijski razred, lahko pa definiramo še parametra *title* ter *description* za opis komponente.

```
from zope.component.factory import Factory

tovarnaUporabnik = Factory(
    Uporabnik,
    title=u"Tovarna za uporabnika"
   )
```

- *registracija tovarne:*
  Tovarna je poimenova storitev, zato moramo poleg modula, v katerem se tovarna nahaja, navesti še njeno ime, s katerim jo bomo poiskali v registru komponent.

```
<utility
    component=".uporabnik.tovarnaUporabnik"
    name="diploma.Uporabnik"
    />
```

- *uporaba tovarne:*
  Tovarno poiščemo in pokličemo z ukazom createObject, kot parameter pa podamo ime tovarne.

```
from zope.component import createObject
uporabnik = createObject(u"diploma.Uporabnik")
```

*Primer 9: Primer uporabe tovarne za kreiranje novega uporabnika*

## 4.5 Dogodki

Pomemben del komponentne arhitekture Zope so tudi *dogodki* (angl. *events*), s katerimi lahko komponenta obvesti zainteresirane komponente oz. *naročnike* (angl. *event subscribers*) o spremembi stanja. Naročniki ob sprožitvi dogodka izvedejo ustrezno akcijo, zato jih imenujemo tudi *upravitelji dogodkov* (angl. *event handlers*).

ZTK definira vrsto koristnih dogodkov, ki so podani v spodnji tabeli, ustvarimo pa lahko tudi lastne.

Tabela 1: Seznam pomembnejših dogodkov

| **Dogodek** | **Opis** |
|---|---|
| IObjectModifiedEvent | Objekt je bil spremenjen. To vključuje dodajanje, premikanje (iz enege vsebnika v drugega), kopiranje ali brisanje objektov. |
| IContainerModifiedEvent | Vsebnik je bil spremenjen. Tu se spremembe nanašajo na dodajanje, odstranjevanje ali razvrščanje objektov, ki jih vsebnik vsebuje. |
| IObjectMovedEvent | Objekt je bil premaknjen v drug vsebnik ali preimenovan. |
| IObjectAddedEvent | Objekt je bil dodan v vsebnik. |
| IObjectCopiedEvent | Objekt je bil kopiran. |
| IObjectCreatedEvent | Objekt je bil ustvarjen. Ta dogodek se sproži, preden je objekt dodan v podatkovno bazo. |
| IObjectRemovedEvent | Objekt je bil odstranjen iz vsebnika. |
| IBeforeTraverseEvent | Ta dogodek se sproži, preden publikacijski mehanizem prispe do objekta. |

Oglejmo si primer uporabe dogodkov:

- *definiranje novega dogodka*

  Dogodek vsebuje podatke, ki upravitelju dogodkov omogočajo izvedbo ustreznih akcij. Ponavadi ob dogodku pošljemo tudi objekt, ki je povzročil spremembo stranja sistema, podamo pa lahko tudi dodatne koristne informacije.

  ```
  class RegistracijaPotrjena(object):
    implements(IRegistracijaPotrjena)
    def __init__(self, registracija):
      self.registracija = registracija
  ```

- *definiranje upravitelja dogodka*

  Upravitelj dogodka je naročen na določen dogodek in na tip objekta, ki je dogodek povzročil. Če želimo dogodek ujeti za vse tipe objektov, podamo ukazu @adapter kot prvi parameter splošen vmesnik Interface.

  ```
  @ zope.component.adapter(IUporabnik, IRegistracijaPotrjena)
  def dodajUporabnika(uporabnik,event):
    print "Registriral se je nov uporabnik: %s." % uporabnik.ime
    # dodaj uporabnika v podatkovno bazo in pošlji sporočilo
  ```

- *registracija upravitelja dogodka*

  Za registracijo dogodka je v potrebno v konfiguracijsko datoteko dodati vrstico:

  ```
  <subscriber handler=".registracija.dodajUporabnika" />
  ```

*Primer 10: Primer uporabe dogodkov*

# 5. Objektna baza Zope (ZODB)

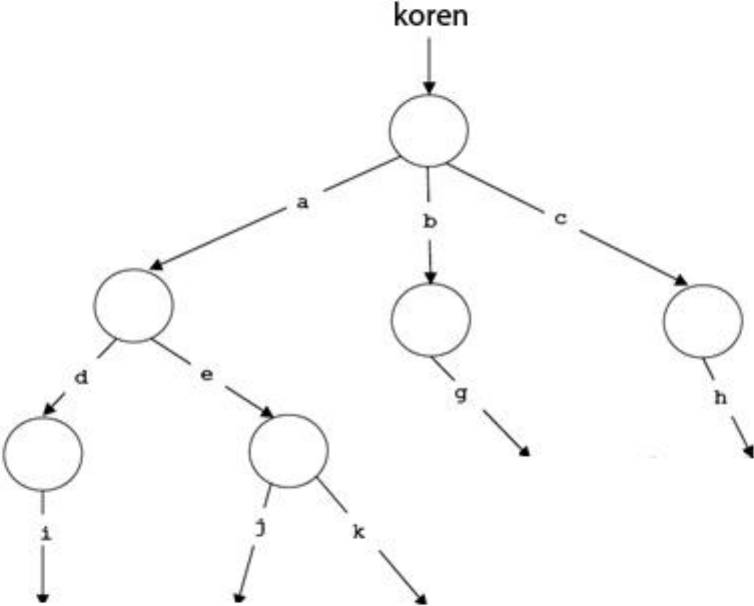
V zadnjem času je opaziti porast popularnosti alternativnih [1] rešitev za shranjevanje podatkov, ki ne temeljijo na relacijskem pristopu. Podatkovne baze, kot so BigTable, Dynamo, Cassandra, MongoDb in druge, prištevamo k t.i. "NoSql" gibanju. Ime je morda malo nerodno izbrano, saj gibanje ne nasprotuje samemu poizvedovalnemu jeziku SQL, ampak želi poudariti, da obstajajo različni pristopi za zagotavljanje trajnosti in da relacijske podatkovne baze niso primerne za vsako situacijo. Glavni problem relacijskih baz je namreč zagotavljanje skalabilnosti pri porazdeljenih sistemih. [28]

Nekateri k omenjenemu gibanju prištevajo tudi objektne baze. Kljub temu, da imajo nekatere skupne lastnosti, pa se je razvoj objektnih baz začel veliko prej in z drugačnimi cilji. Uporaba objektnih baz v kombinaciji z objektno usmerjenim razvojem omogoča številne prednosti, saj ni neskladja med objektno in relacijsko domeno (angl. *object-relational impendance mismatch)*. Neskladje med objektno in relacijsko domeno je množica konceptualnih in tehničnih problemov, na katere naletimo pri uporabi relacijskih baz za shranjevanje podatkov aplikacije, ki je bila zasnova na objektnih principih. [29] Relacijske baze ne podpirajo ključnih konceptov objektno usmerjenega programiranja, dedovanja in polimorfizma. Tako se npr. pojavijo problemi pri pretvorbi kompleksnih hierarhij objektov v tabele - za dostop do podatkov je potrebno veliko stikov tabel, kar je pri globokih hierarhijah zelo potratno, alternativna rešitev pa je uporaba denormalizacije. [30]
Ena izmed prednosti objektnih baz je tudi v tem, da ni potrebno vzdževati in usklajevati dveh različnih podatkovnih shem.

## 5.1 Zgradba ZODB

ZODB ima hierarhično strukturo, ki jo lahko predstavimo z drevesom. Korensko vozlišče je vsebnik (angl. *container)* za vse ostale objekte v podatkovni bazi. Posamezni objekti tudi lahko delujejo kot vsebniki za ostale objekte.

Iskanje določenega objekta ZODB vedno začne v korenskem vozlišču in rekurzivno preišče vsebnike, dokler ga ne najde. Ta mehanizem se imenuje obhod (angl. *traversal)*.




*Slika 6: Struktura podatkovne baze ZODB*

## 5.2 Značilnosti ZODB

### 5.2.1 Transakcije imajo lastnosti ACID

Upoštevanje lastnosti ACID prinaša zanesljivost transakcij in preprečuje nekonsistentnost podatkov [31]:

- *Atomarnost (angl. atomicity)*
  Atomarnost trasakcij zagotavlja, da se izvede celotna transakcija, torej vsi njeni sestavni deli, v nasprotnem primeru mora biti transakcija prekinjena in stanje podatkovne baze nespremenjeno. Spremembe v okviru transakcije se odrazijo v podatkovni bazi s klicem stavka commit(). Spremembe lahko razveljavimo s klicem stavka abort(). Če uporabljamo ZODB v okviru platforme Zope, se transakcija avtomatsko potrdi po vsaki zahtevi (*request*) in razveljavi v primeru napake.

- *Konsistentnost (*angl. *consistency)*
  Konsistentnost transakcij zahteva, da vsaka transakcija pusti podatkovno bazo v stabilnem in konsistentnem stanju.

- *Izolacija (*angl. *isolation)*
  Izolacija transakcij zahteva, da določena operacija ne sme dostopati do podatkov, ki so bili spremenjeni v okviru druge transakcije in ki še ni bila zaključena. Za dosego te lastnosti ZODB uporablja mehanizem MVCC (*Multi-version concurrency control*). Ta mehanizem zagotavlja, da transakcija vedno operira z verzijami objektov, ko je bilo stanje baze še konsistentno.

- *Vzdržljivost (*angl. *durability)*
  Vzdržljivost zagotavlja, da so po potrditvi transakcije njene spremembe shranjene in da se tudi v primeru sistemske napake te spremembe ne izgubijo.

### 5.2.2 Transparentnost

Trajnostni mehanizem je za programerja skoraj popolnoma transparenten. Za shranjevanje ter pridobivanje objektov iz baze ni potrebno uporabljati preslikav med tabelami in objekti ali kakšnih drugih pretvorb.

Pogoji trajnosti (angl. *rules of persistence*):

- Razred mora razširjati razred persistent.Persistent ali katerega od njegovih naslednikov
- Primerki razredov morajo biti povezani v hierarhično drevesno strukturo. Korenski objekt vsebuje ostale objekte, podobno velja tudi za njegove naslednike
- Ob spremembah spremenljivih (angl. *mutable*) objektov (npr. seznam, slovar), ki sami niso trajnostni, je potrebno obvestiti trajnostni mehanizem

Za zagotavljanje transparentnosti trajnostni mehanizem zazna spremembo stanja objekta od znotraj in ob ponastavitvi atributov in klicev metod od zunaj.

Kot smo omenili, je potrebna pazljivost v primeru, ko ima razred atribute tipa seznam, slovar ali podobne spremenljive strukture. Za rešitev problema imamo na voljo dve možnosti:

**1.)** Manj elegantna rešitev je, da po vsaki spremembi takega atributa trajnostnemu mehanizmu s pomočjo zastavice **_p_changed** signaliziramo, da je prišlo do spremembe stanja objekta.

```
uporabnik = Uporabnik()
uporabnik.prijatelji.append("Mojca")
uporabnik._p_changed
```

*Primer 11: Sporočanje trajnostnemu mehanizmu, da je prišlo do spremembe*

**2.)** Bolj transparentna rešitev pa je uporaba trajnostnih implementacij omenjenih podatkovnih struktur. ZODB ponuja razreda PersistentList ter PersistentDist, ki ga je pri velikem številu objektov bolje zamenjati s strukturo BTree. Btree je implementacija B+ dreves, ki omogoča učinkovito dodajanje, iskanje elementov in operaciji unijo ter presek.

Seveda pa ni nujno, da so vsi objekti trajnostni. To velja tudi za atribute objekta. Če se ime atributa začne z »_v_«, trajnostni mehanizem ne beleži sprememb in ga ne shrani v bazo kot del objekta. To je zelo koristno za predpomnenje rezultatov računsko zahtevnih operacij.

#### 5.2.2.1 Prikaz transparentnosti ZODB

Za prikaz transparentnosti in preprostosti upravljanja s podatki podajam še kratek primer.

Definicija trajnostnega razreda:

```
from persistent import Persistent

class Uporabnik(Persistent):

    def __init__(self, ime, priimek):
        self.ime = ime
        self.priimek = priimek
```

*Primer 12: Definicija trajnostnega razreda*

Prikaz osnovnih operacij CRUD (angl. *create, read, update, delete*), simuliranih v konzoli Python:

- **kreiranje:**

```
>>> from uporabnik import Uporabnik
>>> from zope.site.folder import Folder
>>> uporabniki = Folder()
>>> uporabnik = Uporabnik('jim','fulton')
>>> uporabniki['jim-fulton'] =  uporabnik
```

- **branje:**

```
>>> uporabniki['jim-fulton']
<uporabnik.Uporabnik object at XXX>
```

- **posodabljanje:**

```
>>> uporabnik = uporabniki['jimm-fulton']
>>> uporabnik.ime = 'jim'
```

- **brisanje:**

```
>>> del uporabniki['jim-fulton']
```

*Primer 13: Prikaz operacij CRUD pri bazi ZODB*

### 5.2.3 Razveljavitev transakcij

Nekatere implementacije ZODB omogočajo razveljavitev že potrjenih transakcij, tako da lahko obnovimo stanje objekta za mnogo transakcij nazaj. Na to funkcionalnost pa se ne smemo zanašati brezpogojno, saj v določenih primerih razveljavitev ni možna, ker bi privedla do nekonsistentnosti podatkov.

## 5.2.4Izbira načina shranjevanja podatkov in skalabilnost

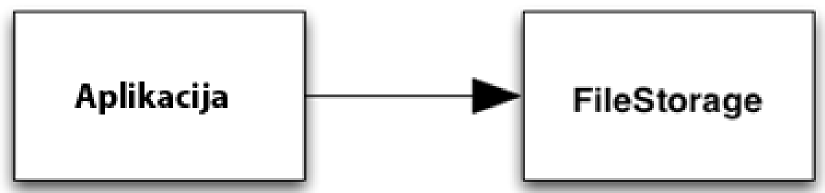
ZODB ponuja več načinov fizičnega shranjevanja objektov: Lahko jih shranjujemo v datoteko, relacijsko bazo ali kateri drug medij, na voljo pa imamo tudi različne možnosti za povečanje zanesljivosti in hitrosti shranjevanja podatkov.

### 5.2.4.1 Shranjevanje na datotečnem sistemu(FileStorage)

*FileStorage* je privzeti način shranjevanja objektov. Vsi podatki so shranjeni v datoteki *Data.fs*, ki vsebuje podatke o vseh transakcijah, torej celotno zgodovino vseh verzij objektov. V delovnem spominu se nahaja indeks za hitro iskanje po datoteki. Njegova velikost je premo sorazmerna z velikostjo datoteke, v povprečju se giblje med 2% in 10%.

Ker so v datoteki shranjene vse pretekle verzije objektov, je pri pogostih spremembah objektov potrebno redno vzdrževanje (t.i. *packing)*. S tem postopkom odstranimo starejše verzije  in nepovezane objekte ter tako zmanjšamo velikost datoteke na normalno raven. Običajno ta postopek avtomatiziramo, paziti pa je potrebno, da je na disku dovolj prostora, saj se ustvari kopija datoteke.

Kreiranje varnostne kopije podatkov je preprosto: Najlažji način je ustavitev strežnika in kreiranje kopije datoteke *Data.fs*. Ustavimo ga zato, da preprečimo nekonsistentnost podatkov, ki se pojavi v primeru, ko v času kreiranja varnostne kopije poteka operacija razveljavitve ali operacija brisanja starejših verzij objektov. V produkcijskem okolju se zato raje izbere orodje *repozo*, ki poskrbi za varnostno kopiranje tudi v času, ko je strežnik aktiven. Imamo tudi možnost kreiranja inkrementalne varnostne kopije.




Slika 7: Shranjevanje podatkov z uporabo datotečnega sistema FileStorage

### 5.2.4.2 Shranjevanje podatkov pri večjem številu zahtev

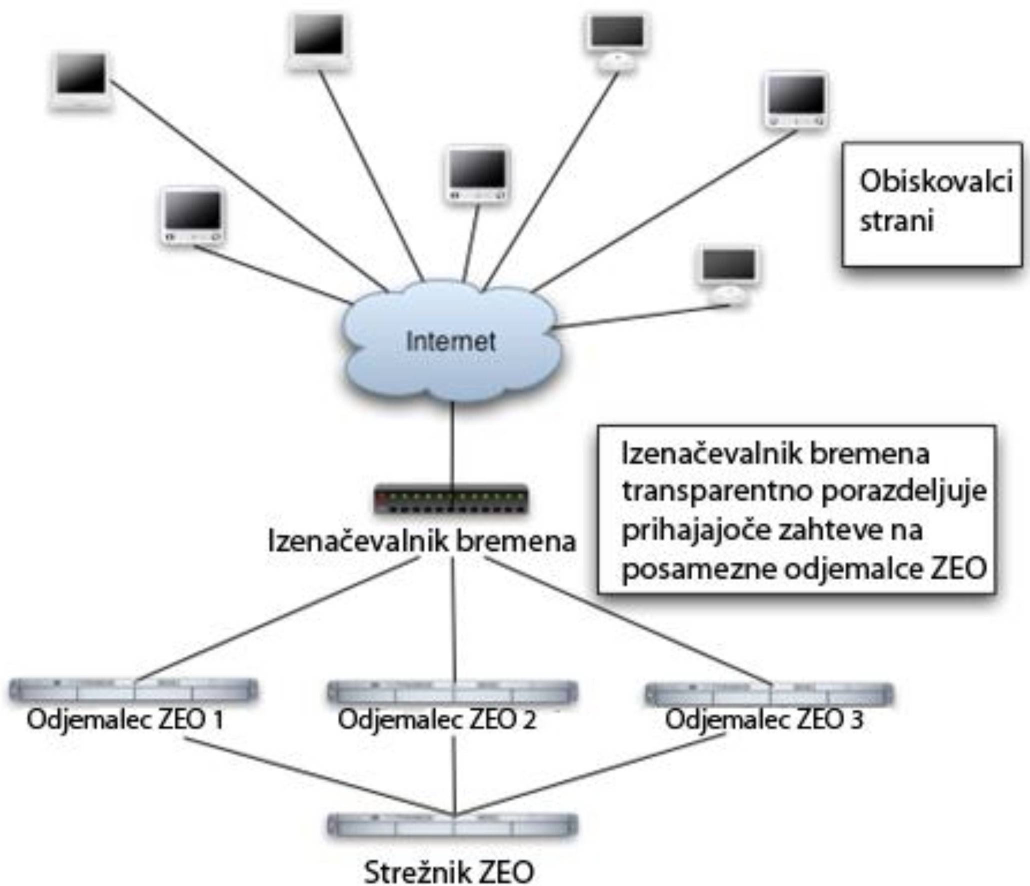
V večini primerov je za strežbo uporabniških zahtev dovolj en sam strežnik. V okoljih z velikim številom uporabnikov pa je za normalno delovanje potrebno zagotoviti več procesorske moči, kar dosežemo z dodajanjem fizičnih strežnikov ali procesorskih enot (procesorjev/procesorskih jeder). Na vsaki strežniški enoti teče svoj proces, vsi procesi pa morajo imeti dostop do skupne podatkovne baze. V tem primeru ne moremo uporabiti načina *FileStorage*, saj posamezen proces zaklene datoteko *Data.fs,* tako je ostalim dostop onemogočen.

Za usklajevanje dostopov do podatkovne baze je potreben mehanizem. Možnih rešitev je več:

#### 5.2.4.2.1 Uporaba strežnika ZEO (ClientStorage)

Strežnik ZEO je način za usklajevanje dostopov do baze ZODB preko internetnega omrežja. Temelji na arhitekturi odjemalec – strežnik: več odjemalcev, ki so lahko na istem fizičnem strežniku ali na različnih koncih sveta, dostopa do centralnega strežnika ZEO, ki usklajuje dostope do podatkovne baze. To poteka tako, da za čas potrjevanja transakcije strežnik ZEO datoteko *Data.fs* zaklene.

Porazdeljevanje zahtev se lahko realizira na več načinov. Realizacija je odvisna tudi od razloga za uporabo strežnika ZEO. Če želimo uporabnikom na različnih koncih sveta omogočiti hitrejši dostop do sistema, jim ponudimo alternativni strežnik blizu njihove lokacije ali jih avtomatsko preusmerimo nanj. V primeru, da želimo povečati zanesljivost ali zmogljivost sistema, pa uporabimo sistem za porazdelitev bremena (angl. *load balancer*), ki enakomerno usmerja uporabniške zahteve na posamezne odjemalce ZEO in tako skrbi za stabilno obremenitev sistema.



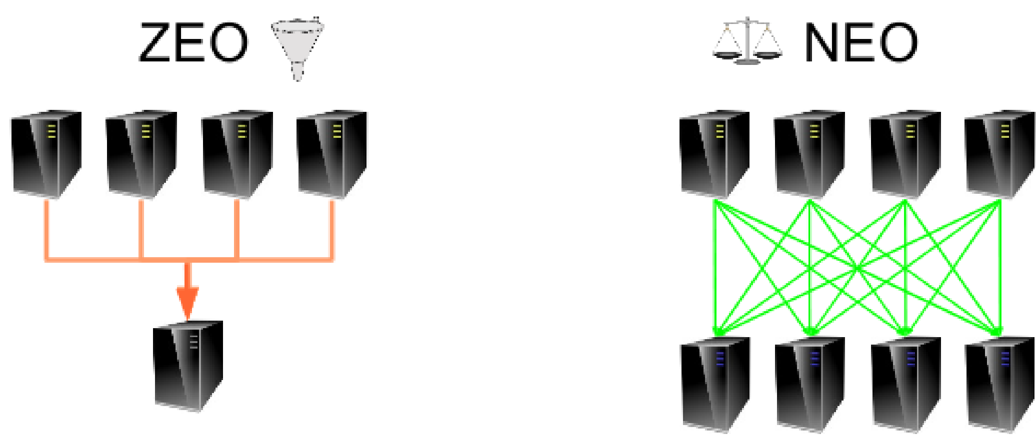

*Slika 8: Shranjevanje podatkov s pomočjo strežnika ZEO in izenačevanja bremena*

Kot je razvidno iz omrežne topologije na zgornji sliki, je strežnik ZEO kritična točka odpovedi (angl. *single point of failure*). Problem običajno rešujemo tako, da za strežnik uporabimo najboljše komponente, z reduntantinmi napajalniki ipd., medtem ko so odjemalci ZEO lahko cenejši, saj tudi v primeru enega izpada ostali še delujejo.

#### 5.2.4.2.2 Uporaba tehnologije NEO

NEO je trenutno še v fazi razvoja in ni primeren za produkcijsko okolje. Njegov cilj je odpraviti omejitve strežnika ZEO ter ponuditi hiter, skalabilen in zanesljiv način shranjevanja podatkov za objektno bazo ZODB [32]. Problem centralnega ZEO strežnika je poleg možnosti za izpad sistema tudi omejena skalabilnost pri velikem številu zahtev. Namesto centralizirane rešitve zato NEO ponuja porazdeljen mehanizem.

Objekti v podatkovni bazi so enakomerno porazdeljeni po vozliščih, vsako vozlišče pa je odgovorno za določeno podmnožico objektov. Te so vse podvojene, tako da lahko v primeru odpovedi katerega od vozlišč odgovornost za njegovo podmnožico objektov prevzame drugo vozlišče. Poleg tega je pri potrjevanju transakcij omejen dostop samo do tistega objekta, na katerega se transakcija nanaša. Pri strežniku ZEO se namreč pri potrjevanju ene transakcije zaklene celotna baza, ostale transakcije pa morajo počakati.




*Slika 9: Primerjava omrežne topologije med sistemoma ZEO ter NEO*

## 5.3 Prednosti in slabosti ZODB

### 5.3.1 Prednosti

Uporaba podatkovne baze ZODB prinaša številne prednosti[33-35]:

- *za aplikacijo je shranjevanje podatkov transparentno*
  Trajnost podatkov je za programerja zelo transparentna, njegova naloga je osredotočiti se na dani problem.

- *ni objektno-relacijskega neskladja (angl. object-relational mismatch)*

- *možnost razveljavljanja sprememb (angl. undo)*
  Nekatere implementacije baze ZODB omogočajo razveljavitev sprememb za teoretično neomejeno število transakcij.

- *iskanje po celotnem besedilu*
  ZODB ima vgrajeno možnost indeksiranja in poizvedovanja po celotnem besedilu.

- *hierarhične strukture so trivialne*
  Pogosto je struktura podatkov takšna, da je hierarhična predstavitev najbolj naravna. Pri ZODB je implementacija takšnih struktur trivialna, pri relacijskih bazah to ni tako preprosto.

- *skalabilnost je transparentna*
  Strežnik ZEO je zelo preprosto vzpostaviti in spremembe v sami aplikaciji niso potrebne.

- *hitrost*
  Hitrost branja in pisanja je pri ZODB zelo spodobna in primerljiva z relacijskimi bazami ter hitrejša kot pri uporabi sistemov ORM [34]. Velika prednost objektnih baz pred relacijskimi se pokaže pri poizvedbah, ki vključujejo veliko povezanih objektov, saj je sledenje kazalcem veliko hitrejše kot operacija stika tabel. Sicer v praksi takšne poizvedbe niso tako pogoste, vendar določenim specializiranim aplikacijam to lahko zelo koristi.

- *stabilnost*

### 5.3.2 Slabosti

Slabosti podatkovne baze ZODB so [34, 35]:

- *prevelika povezanost med aplikacijo in podatkovno bazo*
  Ena izmed večjih omejitev za uporabo ZODB kot centralnega podatkovnega vira v organizaciji je dejstvo, da lahko do podatkovne baze dostopamo samo s pomočjo programskega jezika Python. Poleg tega se lahko zgodi, da ne bomo mogli dostopati do določenih vozlišč v podatkovni bazi, če nimamo nameščene vse potrebne programske opreme, od katere je odvisna naša aplikacija.

- *pomanjkanje orodij za administracijo podatkovne baze*
  Relacijske baze imajo na voljo številna orodja za administracijo in optimizacijo, generiranje poročil itd. Nabor orodij za ZODB je zelo omejen in večinoma brez grafičnega uporabniškega vmesnika. Obstajajo sicer določena orodja za navigacijo po vsebini podatkovne baze, vendar so daleč od orodij, ki so na voljo za relacijske baze.

- *poljubne relacije med objekti niso tako preproste kot pri relacijskih bazah*
  Zasnova baze ZODB je kot nalašč ustvarjena za hierarhično urejene podatke. Sicer podpira tudi poljubne relacije med objekti, vendar v določenih primerih navadne

relacije med objekti ne zadoščajo in moramo uporabiti knjižnice za upravljanje relacij. To je potrebno, če hočemo realizirati relacije tipa »mnogo-mnogo« ali izvajati hitre poizvedbe, da ugotovimo s katerimi objekti je povezan določen objekt, brez potratnega pregledovanja celotnega drevesa objektov.

- *nestandardni način izvajanja poizvedb*
  Med proizvajalci objektnih baz so nekaj časa potekala usklajevanja in poskus vzpostavitve enotnega poizvedovalnega jezika za objektne baze, ki naj bi bil precej podoben jeziku SQL [33]. Do uveljavitve standarda ni nikoli prišlo, nekatere objektne baze za dostop in poizvedovanje sicer ponujajo tudi možnost uporabe jezika SQL, vendar ZODB tega ne podpira. Poizvedbe so možne samo s programskim jezikom Python, na voljo pa so tudi knjižnice za preprosto iskanje po indeksiranih atributih.

- *indeksiranje ni transparentno*
  Indeksiranje je realizirano na aplikativni ravni – željene indekse definiramo programsko, v določenih primerih pa mora programer tudi sam poskrbeti za njihovo posodobitev in ponovno indeksiranje objektov.

- *težavne spremembe podatkovne sheme*
  Če spremenimo podatkovno shemo določenega razreda, npr. dodamo ali spremenimo atribut, je potrebno napisati programsko skripto za migracijo, ki ustrezno popravi že shranjene objekte. To je sicer potrebno samo v primeru, ko je atribut inicializiran v konstruktorju in ne na ravni razreda. Nekatere objektne baze tovrstne spremembe opravijo avtomatsko. Tudi pri relacijskih bazah je dodajanje atributa preprosto, vendar so v določenih primerih spremembe sheme zahtevnejše kot pri ZODB.

- *slaba hitrost ob istočasnem dostopu do istega objekta*

- *ni v široki uporabi*

### 5.3.3Kdaj uporabiti ZODB?

Kot vidimo, ima ZODB nekaj zanimivih lastnosti, vendar tudi določene slabosti. Odločitev o izbiri vrste podatkovne baze je tako odvisna od tipa projekta in potreb organizacije.

Če moramo zagotoviti dostop do podatkovne baze različnim aplikacijam, napisanim v različnih programskih jezikih, izvajati veliko kompleksnih poizvedb ter generirati razna poročila, potem je primernejša uporaba relacijske baze.

ZODB na drugi strani je kot nalašč ustvarjena za splet in sisteme za upravljanje z vsebinami, kjer imamo opravka z napol strukturiranimi in hierarhično urejenimi podatki, ter v specializiranih znanstvenih aplikacijah, kjer se srečujemo s kompleksnimi grafi objektov [33].

V primeru, ko je glavna skrb zagotavljanje razpoložljivosti v porazdeljenih okoljih, pa je vredno razmislilti o uporabi ene izmed modernejšh podatkovnih baz NoSQL.

# 6. Razvoj prototipa aplikacije

## 6.1 Uvod

V tem poglavju bom predstavil praktičen primer razvoja z ogrodjem Grok. Poskušal bom prikazati, kako lastnosti platforme vplivajo na posamezne korake razvoja in kako razvijalcu olajšajo določena opravila.

Razvil bom preprost prototipni sistem za upravljanje aktivnosti, povezanih z ekipnimi rekreativnimi športi. Namen aplikacije je poenostaviti organizacijo dogodkov in komunikacijo med člani ekipe, omogočiti vodenje evidence dogodkov ter upravljanje s plačili (nakup novih dresov, plačilo prijavnine za tekmovanja,..).

Ideja za realizacijo tovrstne rešitve se je pojavila kot posledica slabih izkušenj pri organizaciji treningov in tekem naše rekreativne nogometne ekipe. Medsebojno usklajevanje glede lokacij, terminov ipd. je pri velikem številu igralcev precej naporna naloga, zato si jo želimo s pomočjo aplikacije malo olajšati.

## 6.2 Analiza in načrtovanje

### 6.2.1 Opis problemske domene

Izdelati želimo spletni informacijski sistem za pomoč pri vodenju aktivnosti, povezanih s skupinskimi športi (nogomet, košarka, rokomet itd.). Anonimni **obiskovalec** ima možnost registracije. Po vnosu svojega elektronskega naslova in želenega gesla, mu sistem pošlje elektronsko pošto s potrditveno povezavo. Po kliku na to povezavo se lahko potem prijavi v sistem. Kot prijavljen **uporabnik** sistema ima možnost:

- urejanja svojega profila, kjer lahko poleg podatkov, ki jih je vnesel ob registraciji, ureja še svoje ime, priimek, naslov in telefonsko številko;
- ustvarjanja ene ali več ekip (ime ekipe, opis, vrsta športa, fotografija).

Ko **uporabnik** ustvari novo ekipo, v njej prevzame vlogo **administratorja ekipe**. Dostop ima do naslednjih funkcionalnosti:

- urejanje podatkov o ekipi;
- dodajanje in urejanje članov ekipe (ime, priimek, elektronska pošta, vloga v ekipi – »vodja ekipe«, »igralec« itd., status igralca – »pripravljen na igro«, »poškodovan«, »začasno odsoten«) in urejanje njihovih pravic – dodeli jim lahko vlogo navadnega člana ekipe, administratorja dogodkov ali administratorja ekipe;
- vodenje evidence plačil, povezanih z delovanjem ekipe, za nakup nove žoge, prijavnine za ligo itd. (namen plačila, opis, znesek, rok plačila).

**Uporabnik**, ki ima v določeni ekipi vlogo **administratorja dogodkov**, lahko dodaja in ureja dogodke(ime, datum in ura začetka, datum in ura konca, tip dogodka, lokacija, opis) ter vnaša

in ureja rezultate dogodkov.

**Uporabnik**, ki ima v določeni ekipi vlogo navadnega **člana ekipe**, pa ima naslednje možnosti:

- pregledovanje podatkov o ekipi in njenih članih;
- pregledovanje dogodkov, potrjevanje udeležbe za določen dogodek in vnos komentarjev pri dogodku;
- pregledovanje rezultatov preteklih dogodkov.

**Administrator ekipe** ima poleg svojih pravic tudi vse pravice **administratorja dogodkov** in **člana ekipe**.

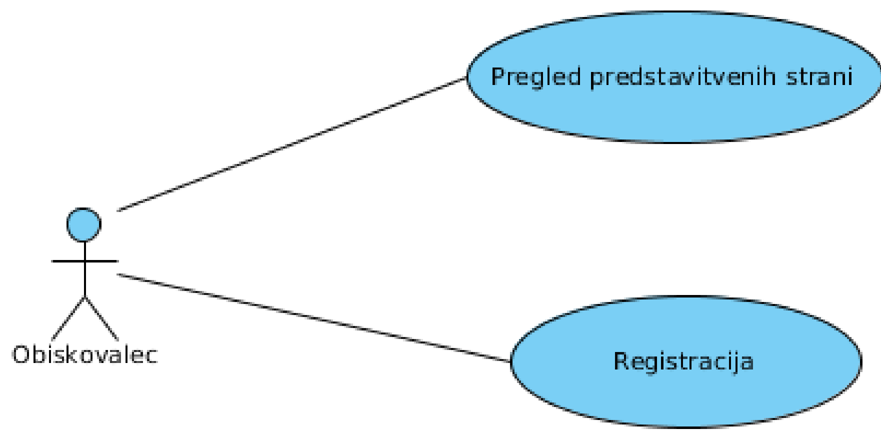
Ob vseh pomembnejših dogodkih (kreiranje ekipe, dodajanje igralca, dogodka itd.) naj sistem pošlje sporočilo vpletenim akterjem.

Sistem bo robustno zasnovan in pripravljen na razširitve, omogočal pa bo tudi prevod v tuje jezike.

## 6.2.2 Diagrami primerov uporabe (PU)

Primer uporabe je zaključen tok dogodkov, ki imajo določen namen. Prikazuje pomembnejši način uporabe sistema za enega ali več akterjev, ki vplivajo na ta primer uporabe. V celotu naj bi vsi primeri uporabe vključevali vse možne načine uporabe sistema.

### 6.2.2.1 Diagram primerov uporabe za akterja "Obiskovalec"




*Slika 10: Diagram primerov uporabe za akterja »Obiskovalec*

##### 6.2.2.1.1 Pregled predstavitvenih strani

***Kratek opis***

Ta primer uporabe omogoča obiskovalcu pregled strani, ki mu predstavijo funkcije in delovanje sistema.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko obiskovalec klikne na določeno povezavo na predstavitveno stran.

1. Sistem obiskovalcu prikaže zahtevano stran.

***Alternativni tokovi***

Brez.

***Posebne zahteve***

Brez.

***Predpogoji***

Brez.

***Popogoji***

Brez.

***Razširitvene točke***

Brez.

##### 6.2.2.1.2 Registracija

***Kratek opis***

Ta primer uporabe omogoča obiskovalcu registracijo v sistem in s tem dostop do dodatnih storitev sistema.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko obiskovalec klikne na gumb »Registriraj se«.

1. Sistem obiskovalcu prikaže formo z naslednjimi vnosnimi polji:
    - e-naslov *
    - geslo *
    - potrditev gesla *

2. Obiskovalec izpolne zahtevana polja in klikne na gumb »Registracija«

3. Sistem preveri, če je uporabnik s tem elektronskim naslovom morda že registriran, ali je naslov veljaven ter ali se vnešeni gesli ujemata.

4. V podatkovni bazi se ustvari nova registracija s podatki o uporabniku.

5. Sistem obiskovalcu po elektronski pošti pošlje povezavo za potrditev registracije, na zaslon pa izpiše obvestilo o registraciji.

6. Ko obiskovalec klikne na poslano povezavo, sistem uporabnika doda v podatkovno bazo z vlogo "Uporabnik", nato prikaže formo za prijavo v sistem.

### *Alternativni tokovi*

#### *Obiskovalec vnese napačne podatke*

Če se obiskovalec zmoti pri vnosu zahtevanih podatkov oz. je vnešen e-naslov že v uporabi, sistem izpiše opozorilo ob poljih z napako, obiskovalec pa ima možnost popraviti zahtevane spremembe in nato ponovno klikniti gumb za registracijo.

#### *Obiskovalec ne potrdi registracije*

Če obiskovalec ne potrdi registracije v roku enega tedna, sistem uporabnika izbriše.

### *Posebne zahteve*

Brez.

### *Predpogoji*

Brez.

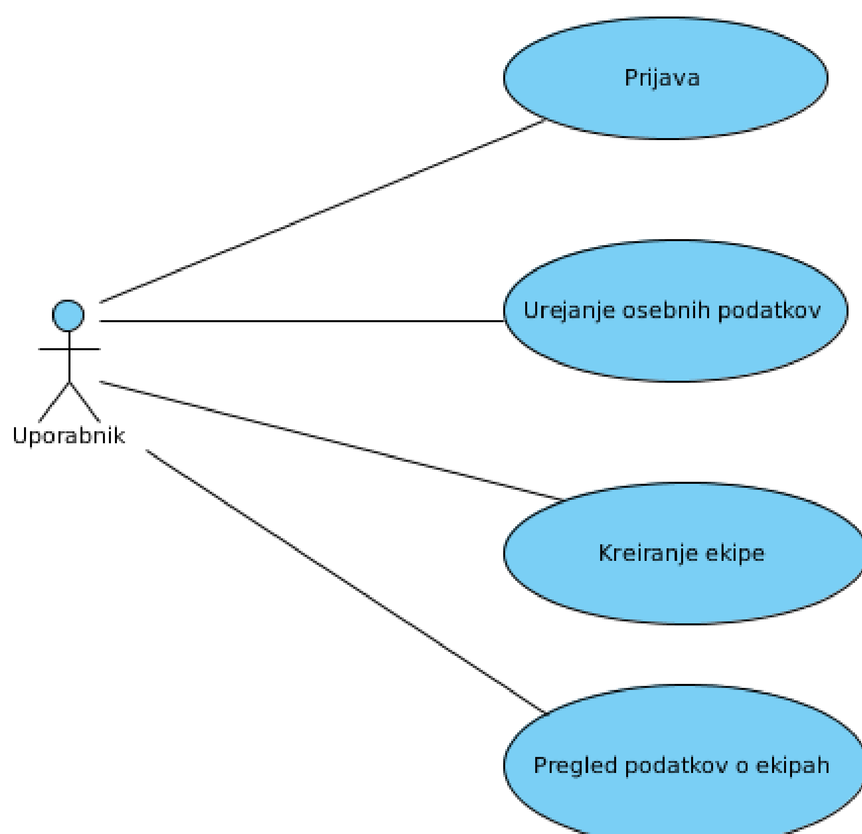
### *Popogoji*

Če je primer uporabe uspešno zaključen, je v podatkovni bazi dodan nov uporabnik. Če uporabnik registracije ne potrdi, je v podatkovni bazi registracija aktivna še en teden, nato se izbriše. V nasprotnem primeru se stanje sistema ne spremeni.

### *Razširitvene točke*

Brez.

### 6.2.2.2 Diagram primerov uporabe za akterja "Uporabnik"




*Slika 11: Diagram primerov uporabe za akterja »Uporabnik«*

#### 6.2.2.2.1 Prijava

***Kratek opis***

Ta primer uporabe omogoča uporabniku prijavo v sistem.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko uporabnik klikne na gumb »Prijava«.

1. Sistem uporabniku prikaže formo s polji za vnos uporabniškega imena in gesla.

2. Uporabnik vnese zahtevane podatke in klikne na gumb »Prijava«.

3. Sistem preveri, če v podatkovni bazi obstaja uporabnik s tem uporabniškim imenom in geslom.

4. Sistem uporabnika prijavi, v sejo shrani uporabnikovo privzeto ekipo in ga preusmeri na njegovo domačo stran.

##### *Alternativni tokovi*

###### *Uporabnik vnese napačno uporabniško ime ali geslo*

Sistem uporabnika opozori na napačno vnešene podatke. Uporabnik lahko ponovno vnese uporabniško ime in geslo in klikne gumb »Prijava«.

#### *Posebne zahteve*

Brez.

#### *Predpogoji*

Brez.

#### *Popogoji*

Če je primer uporabe uspešno zaključen, je uporabnik prijavljen v sistem. V nasprotnem primeru se stanje sistema ne spremeni.

#### *Razširitvene točke*

Brez.

### 6.2.2.2.2 Kreiranje ekipe

#### *Kratek opis*

Ta primer uporabe omogoča uporabniku kreiranje nove športne ekipe.

#### *Tok dogodkov*

##### *Glavni tok*

Primer uporabe se začne, ko uporabnik klikne na gumb »Ustvari ekipo«.

1. Sistem uporabniku prikaže formo s polji za vnos nove ekipe:
   - fotografija ekipe
   - ime ekipe *
   - šport *
   - opis ekipe

2. Uporabnik vnese zahtevane podatke in klikne na gumb »Ustvari ekipo«.

3. Sistem v podatkovni bazi ustvari novo ekipo, uporabnik pa postane član ekipe z vlogo "Administrator ekipe".

4. Sistem uporabnika preusmeri na domačo stran ustvarjene ekipe.

##### *Alternativni tokovi*

###### *Uporabnik ne vnese vseh zahtevanih podatkov*

Če uporabnik ne vnese vseh zahtevanih podatkov, sistem izpiše opozorilo ob poljih z napako, uporabnik pa ima možnost opraviti zahtevane spremembe in nato ponovno klikniti gumb za kreiranje ekipe.

***Posebne zahteve***

Brez.

***Predpogoji***

Uporabnik mora biti prijavljen v sistem.

***Popogoji***

Če je primer uporabe uspešno zaključen, sta v podatkovni bazi ustvarjena nova ekipa ter član ekipe z vlogo Administrator ekipe. V nasprotnem primeru se stanje sistema ne spremeni.

***Razširitvene točke***

Brez.

#### 6.2.2.2.3 Urejanje osebnih podatkov

***Kratek opis***

Ta primer uporabe Uporabniku omogoča urejanje osebnih nastavitev, kot so ime, priimek, naslov itd. ter spreminjanje gesla za dostop do sistema.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko uporabnik klikne na gumb »Urejanje profila«.

1. Sistem prikaže stran z naslednjimi vnosnimi polji:
   - ime
   - priimek
   - naslov
   - telefonska številka
   - geslo
   - potrditev novega gesla * (obvezno, če je polje geslo ni prazno)

2. Uporabnik vnese željene podatke in klikne gumb »Shrani«.

3. Sistem shrani spremembe in uporabniku prikaže sporočilo.

***Alternativni tokovi***

***Uporabnik vnese gesli, ki se ne ujemata***

Če uporabnik želi spremeniti geslo in ne vnese dveh enakih gesel, sistem uporabnika opozori na napako. Uporabnik ima nato možnost opraviti zahtevane spremembe in nato ponovno klikniti gumb za shranjevanje.

***Posebne zahteve***

Brez.

***Predpogoji***

Uporabnik mora biti prijavljen v sistem.

***Popogoji***

Če je primer uporabe uspešno izveden, so v podatkovni bazi posodobljeni podatki o uporabniku. V nasprotnem primeru se stanje sistema ne spremeni.

***Razširitvene točke***

Brez.

#### 6.2.2.2.4 Pregled podatkov o ekipah

***Kratek opis***

- Ta primer uporabe omogoča uporabniku pregled novic in dogodkov, povezanih z njegovimi ekipami.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko se uporabnik prijavi v sistem ali ko klikne na gumb »Domov«.

1. Sistem uporabniku prikaže stran z naslednjimi razdelki:

    - "Moje ekipe":
      Prikazan je seznam vseh ekip, katerih član je, s povezavami na domače strani ekip.

    - "Prihajajoči dogodki":
      Prikazan je seznam bližajočih se dogodkov, ki vsebuje naslednja polja:
        - datum
        - lokacija
        - ekipa
        - nasprotnik
        - tip dogodka
        - podrobnosti(povezava na stran dogodka)

    - "Zadnje novice":
      Tu so prikazani razni dogodki, povezani z delovanjem ekip, kot so sporočila o zadnjih odigranih tekmah, dodanih igralcih, zadnji komentarji igralcev ipd. Vsak vnos vsebuje tudi povezavo.

***Posebne zahteve***

Brez.

***Predpogoji***

Uporabnik mora biti prijavljen v sistem.

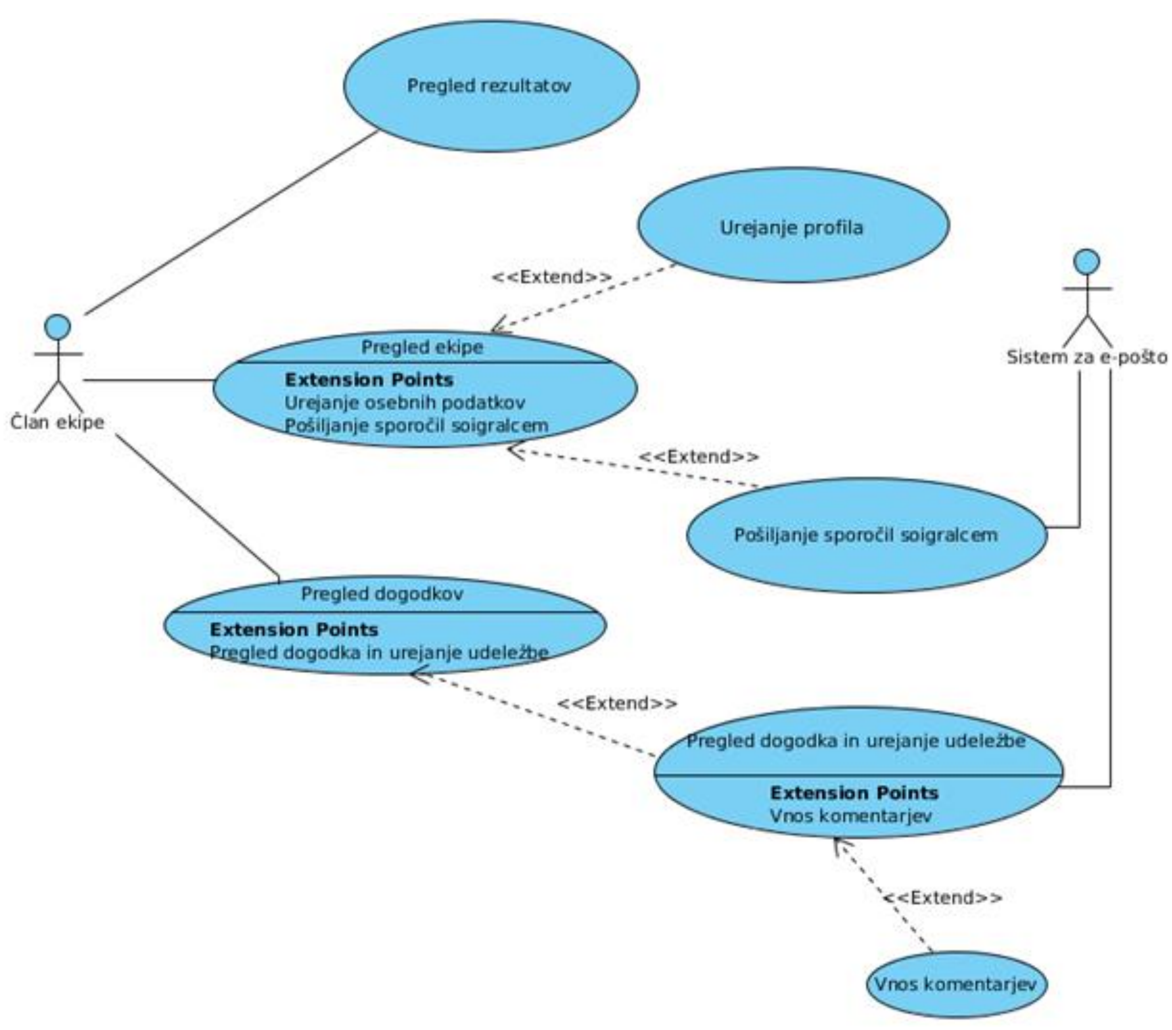
***Popogoji***

Brez.

***Razširitvene točke***

Brez.

### 6.2.2.3 Diagram primerov uporabe za akterja "Član ekipe"




*Slika 12: Diagram primerov uporabe za člana ekipe*

#### 6.2.2.3.1 Pregled rezultatov

***Kratek opis***

Ta primer uporabe omogoča članu ekipe pregled rezultatov in raznih statističnih podatkov preteklih dogodkov.

### Tok dogodkov

#### Glavni tok

Primer uporabe se začne, ko član ekipe klikne na gumb »Rezultati«.

1. Sistem prikaže rezultate dogodkov v tekoči sezoni: v prvi vrstici so prikazani seštevki rezultatov v tej sezoni:
    - sezona
    - število zmag
    - število porazov
    - število neodločenih izidov
    - najboljši igralec sezone

Nato sledi seznam z rezultati za vsak dogodek posebej z naslednjimi stolpci:
- datum
- lokacija
- tip dogodka
- nasprotnik
- rezultat
- gumb »Podrobno«

2. Uporabnik izbere sezono, za katero želi pregledati rezultate.

3. Sistem članu ekipe prikaže rezultate izbrane sezone.

V primeru, da član izbere možnost "Podrobno" zraven izbranega dogodka, se izvede podtok "Pregled rezultatov tekme".

#### Podtok pregled rezultatov tekme

Član ekipe si lahko ogleda tudi podrobnejše podatke o določenem dogodku s klikom na gumb »Podrobno« zraven izbranega dogodka.

1. Sistem članu ekipe prikaže podrobne podatke o izbrani tekmi.

### Posebne zahteve

Brez.

### Predpogoji

Član ekipe mora biti prijavljen v sistem.

### Popogoji

Brez.

### Razširitvene točke

Brez.

#### 6.2.2.3.2 Pregled ekipe

***Kratek opis***

Ta primer uporabe članu ekipe omogoča pregled podatkov o njegovi ekipi in igralcih, ki jo sestavljajo. Poleg tega omogoča tudi urejanje njegovega profila v okviru te ekipe ter pošiljanje sporočil celotni ekipi oz. izbranim igralcem.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko član ekipe klikne na gumb »Ekipa«.

1. Sistem članu ekipe prikaže splošne podatke o ekipi:
    - slika ekipa
    - ime ekipe
    - število članov(vseh, aktivnih, poškodovanih)
    - leto ustanovitve itd. ...

Pod podatki o ekipi je prikaz seznam z Igralci oblike:

- osebna slika
- vzdevek
- ime
- priimek
- položaj v ekipi("branilec","napadalec", "trener",..)
- vloga("Administrator ekipe, Administrator dogodkov", "Član ekipe")
- status(»pripravljen na igro«, »poškodovan«, »začasno odsoten«)

***Alternativni tokovi***

***Član ekipe želi poslati sporočilo ostalim igralcem***

V primeru, da želi član ekipe poslati sporočilo celotni ekipi oz. izbranim igralcem, se izvede primer uporabe **Pošiljanje sporočil soigralcem.**

***Posebne zahteve***

Brez.

***Predpogoji***

Član ekipe mora biti prijavljen v sistem.

***Popogoji***

Brez.

***Razširitvene točke***

- Urejanje profila
- Pošiljanje sporočil soigralcem

#### 6.2.2.3.3 Urejanje profila

***Kratek opis***

Poleg osebnega profila ima uporabnik kot član določene ekipe tudi možnost urejanja podatkov v kontekstu te ekipe. Na strežnik lahko naloži svojo fotografijo, vnese kratko biografijo, določi ali je trenutno pripravljen na igro itd. .

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko član ekipe klikne na gumb »Urejanje profila« na strani izbrane ekipe.

1. Sistem prikaže stran z naslednjimi vnosnimi polji:
   - vzdevek
   - osebna fotografija
   - biografija
   - status (»pripravljen na igro«, »poškodovan«, »začasno odsoten«)

2. Uporabnik vnese željene podatke in klikne gumb »Shrani«.

3. Sistem shrani spremembe in uporabniku prikaže sporočilo.

***Alternativni tokovi***

Brez.

***Posebne zahteve***

Brez.

***Predpogoji***

Član ekipe mora biti prijavljen v sistem.

***Popogoji***

Če je primer uporabe uspešno izveden, so v podatkovni bazi posodobljeni podatki za člana ekipe. V nasprotnem primeru se stanje sistema ne spremeni.

***Razširitvene točke***

Brez.

#### 6.2.2.3.4 Pošiljanje sporočil soigralcem

***Kratek opis***

Ta primer uporabe članu ekipe omogoča pošiljanje sporočil celi ekipi ali izbranim soigralcem.

### *Tok dogodkov*

#### *Glavni tok*

Primer uporabe se začne, ko **član ekipe** klikne na gumb »Pošlji sporočilo«.

1. Sistem prikaže vnosno formo oblike:
    - zadeva *
    - sporočilo *

Spodaj sistem prikaže še formo za izbiro prejemnikov sporočila:
- pošlji vsem
- pošlji samo tistim, ki imajo določen status(»pripravljen na igro«, »poškodovan«, »začasno odsoten«)
- pošlji izbranemu članu ekipe

2. Član ekipe vnese zahtevane podatke in klikne gumb »Pošlji«.

3. Sistem pošlje e-pošto izbranim soigralcem.

#### *Alternativni tokovi*

##### *Član ekipe ne vnese vseh zahtevanih podatkov*

Če član ekipe ne vnese vseh zahtevanih podatkov, sistem izpiše obvestilo o napaki. Član ekipe lahko nato opravi zahtevane spremembe in ponovno pošlje sporočilo ali prekine pošiljanje s klikom na »Prekliči«,  pri čemer se primer uporabe zaključi.

##### *Član ekipe prekliče pošiljanje*

Član ekipe lahko prekliče pošiljanje s klikom na »Prekliči«, pri čemer se primer uporabe zaključi.

### *Posebne zahteve*

Brez.

### *Predpogoji*

Član ekipe mora biti prijavljen v sistem.

### *Popogoji*

Brez.

### *Razširitvene točke*

Brez.

## 6.2.2.3.5 Pregled dogodkov

### *Kratek opis*

Ta primer uporabe članu ekipe  omogoča pregled planiranih dogodkov ter podrobnosti o izbranem izbranem dogodku.

### Tok dogodkov

#### Glavni tok

Primer uporabe se začne, ko član ekipe klikne na gumb »Dogodki«.

1. Sistem prikaže seznam planiranih dogodkov oblike:
   - datum
   - tip dogodka
   - nasprotnik(če gre za tekmo, v nasprotnem primeru je polje prazno)
   - lokacija
   - podrobnosti

2. Član ekipe lahko klikne na gumb »Podrobnosti« pri izbranem dogodku, pri čemer se izvede primer uporabe "Pregled dogodka in urejanje udeležbe"

### Posebne zahteve

Brez.

### Predpogoji

Član ekipe mora bit prijavljen v sistem.

### Popogoji

Brez.

### Razširitvene točke

Pregled dogodka in urejanje udeležbe

## 6.2.2.3.6 Pregled dogodka in urejanje udeležbe

### Kratek opis

Ta primer uporabe omogoča članu ekipe pregled podatkov o dogodku, urejanje statusa udeležbe, pregled nad statusom udeležbe ostalih igralcev ter vnos komentarjev.

### Tok dogodkov

#### Glavni tok

1. Sistem prikaže podatke o dogodku, prikazani so naslednji podatki:
   - datum
   - tip dogodka
   - nasprotnik(če gre za tekmo, v nasprotnem primeru je polje prazno)
   - lokacija

Pod formo s podatki o dogodku je prikazan povzetek statusa ostalih igralcev in forma za spremembo statusa udeležbe za Člana ekipe z naslednjimi polji:
   - izbira statusa (»potrjujem udeležbo«,»ne utegnem«,»morda«)
   - število dodatnih igralcev
   - opombe

2. Član ekipe vnese oz. spremeni status udeležbe in klikne na gumb »Shrani« za s

hranitev sprememb.

3. Sistem vnešene spremembe shrani v bazo.

### *Alternativni tokovi*

#### *Član ekipe ne shrani nastavitev*

V primeru, da član ekipe ne klikne gumba »Shrani«, se spremembe razveljavijo.

### *Posebne zahteve*

Brez.

### *Predpogoji*

Član ekipe mora biti prijavljen v sistem.

### *Popogoji*

V primeru uspešne izvedbe primera uporabe je v bazi shranjen novi status udeležbe Člana ekipe za dani dogodek, v nasprotnem primeru se stanje sistema ne spremeni.

### *Razširitvene točke*

13. Vnos komentarjev

## 6.2.2.3.7 Vnos komentarjev

### *Kratek opis*

Ta primer uporabe omogoča članu ekipe vnos komentarjev.

### *Tok dogodkov*

#### *Glavni tok*

1. Sistem prikaže seznam komentarjev ter formo za vnos novega komentarja.

2. Član ekipe vnese besedilo in klikne »Potrdi«.

3. Sistem shrani njegov komentar v podatkovno bazo.

### *Posebne zahteve*

Brez.

### *Predpogoji*

Član ekipe mora biti prijavljen v sistem.

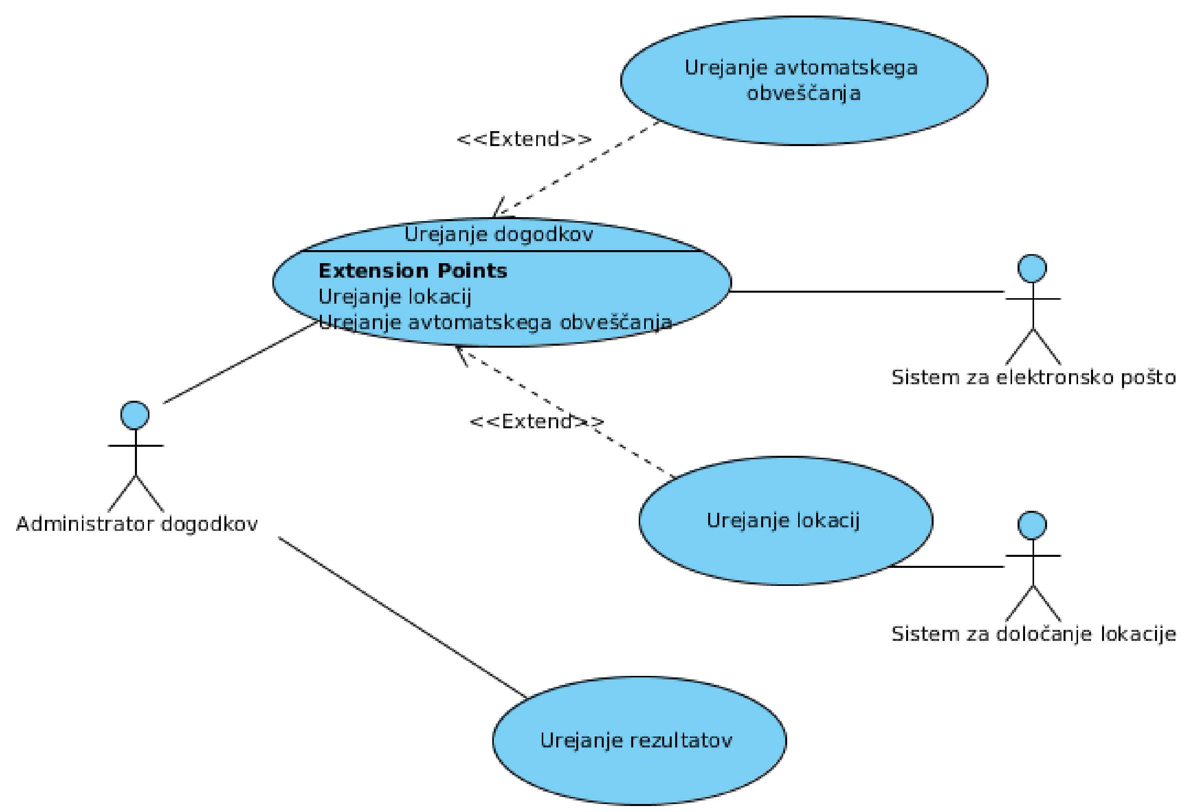
### *Popogoji*

V primeru uspešne izvedbe primera uporabe, se v podatkovno bazo shrani komentar Člana ekipe, v nasprotnem primeru se stanje sistema ne spremeni.

***Razširitvene točke***

Brez.

### 6.2.2.4 Diagram primerov uporabe za akterja "Administrator dogodkov"




*Slika 13: Diagram primerov uporabe za administratorja dogodkov*

#### 6.2.2.4.1 Urejanje dogodkov

***Kratek opis***

Ta primer uporabe Administratorju dogodkov omogoča dodajanje, urejanje in brisanje dogodkov.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko Administrator dogodkov klikne na gumb »Dogodki«.

1. Sistem pokliče primer uporabe **Pregled dogodkov,** poleg tega pa **Administratorju dogodkov** ponudi nekatere dodatne možnosti:

- dodajanje dogodka --> podtok »Dodajanje dogodka«

- urejanje dogodka --> podtok »Urejanje dogodka«
- brisanje dogodka --> podtok »Brisanje dogodka«
- urejanje nastavitev obveščanja --> izvede se primer uporabe **Urejanje avtomatskega obveščanja**
- **urejanje lokacij --> izvede se primer uporabe Urejanje lokacij**

2. Glede na izbiro **Administratorja dogodkov** se izvede ustrezen podtok.

### *Podtok »Dodajanje dogodka«*

1. Sistem prikaže formo z naslednjimi vnosnimi polji:
   - ime dogodka
   - tip dogodka *
   - datum začetka *
   - ura začetka *
   - datum zaključka
   - ura zaključka
   - nasprotnik
   - lokacija

2. **Administrator dogodkov** vnese zahtevane podatke in klikne na gumb »Dodaj dogodek«

3. Sistem v bazi ustvari novi dogodek, nato se ponovno izvede glavni tok.

### *Podtok »Urejanje dogodka«*

1. Sistem prikaže formo z istimi polji kot pri podtoku »Dodajanje dogodka«, vendar z vnešenimi podatki o dogodku.

2. Administrator dogodkov opravi željene spremembe in klikne na gumb »Shrani«.

3. Sistem posodobi podatke za dogodek, nato se ponovno izvede glavni tok.

### *Podtok »Brisanje dogodka«*

1. Administrator dogodkov klikne na gumb »Izbriši« zraven izbranega dogodka.

2. Sistem prikaže potrditveno okno.

3. Administrator dogodkov potrdi brisanje s klikom na »Potrdi«.

4. Sistem izbriše dogodek iz baze.

## *Alternativni tokovi*

### *Administrator dogodkov vnese napačne podatke*

Če pride do napake pri vnosu zahtevanih podatkov pri katerem od podtokov, sistem

izpiše opozorilo ob poljih z napako, Administrator dogodkov ima možnost popraviti zahtevane spremembe in ponovno potrditi akcijo.

***Posebne zahteve***

Brez.

***Predpogoji***

Administrator dogodkov mora biti prijavljen v sistem.

***Popogoji***

V primeru uspešne izvedbe katerega od podtokov so v bazi dodani, posodobljeni ali izbrisani podatki o dogodku, v nasprotnem primeru se stanje sistema ne spremeni.

***Razširitvene točke***

- Urejanje lokacij
- Urejanje avtomatskega obveščanja

##### 6.2.2.4.2 Urejanje avtomatskega obveščanja

***Kratek opis***

Ta primer uporabe **Administratorju dogodkov** omogoča nastavitve avtomatskega obveščanja pred dogodkom.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko Administrator dogodkov klikne na gumb »Nastavitve obveščanja«.

1. Sistem prikaže stran z naslednjimi vnosnimi polji:
   - dovoli obveščanje
   - koliko dni pred dogodkom pošlji obvestilo?
   - pošlj samo igralcem s statusom ("pripravljen na igro", "odsoten",...)

2. Uporabnik vnese željene podatke in klikne gumb »Shrani«.

3. Sistem shrani spremembe in uporabniku prikaže sporočilo.

***Posebne zahteve***

Brez.

***Predpogoji***

Administrator dogodkov mora biti prijavljen v sistem.

***Popogoji***

Če je primer uporabe uspešno izveden, so v podatkovni bazi posodobljene nastavitve obveščanja. V nasprotnem primeru se stanje sistema ne spremeni.

***Razširitvene točke***

Brez.

#### 6.2.2.4.3 Urejanje lokacij

***Kratek opis***

Ta primer uporabe Administratorju dogodkov omogoča dodajanje, urejanje ter brisanje lokacij, kjer se odvijajo dogodki.

***Tok dogodkov***

***Glavni tok***

1. Sistem prikaže seznam lokacij oblike:
   - fotografija
   - naslov
   - mesto
   - država
   - podrobno(povezava)
   - uredi(povezava)
   - izbriši(povezava)

2. Administrator dogodkov ima na voljo naslednje možnosti:
   - dodajanje nove lokacije – podtok »Dodajanje lokacije«
   - urejanje lokacije – podtok »Urejanje lokacije«
   - brisanje lokacije – podtok »Brisanje lokacije«
   - ogled lokacije na zemljevidu – podtok »Podroben ogled lokacije«

3. Glede na izbiro se izvede ustrezni podtok.

***Podtok »Dodajanje lokacije«***

1. Sistem prikaže formo z naslednjimi vnosnimi polji:

   - ime lokacije
   - naslov *
   - poštna številka *
   - država *
   - tip podlage

2. Administrator dogodkov vnese zahtevane podatke in klikne na gumb »Dodaj novo lokacijo«

3. Sistem v bazi ustvari novo lokacijo, ponovno se izvede glavni tok.

***Podtok »Urejanje lokacije«***

1. Sistem prikaže formo z istimi polji kot v podtoku »Dodajanje lokacije« in

shranjenimi podatki lokacije.

2. Administrator dogodkov opravi željene spremembe in klikne na gumb »Shrani«.

3. Sistem posodobi podatke za lokacijo v podatkovni bazi, ponovno se izvede glavni tok.

#### Podtok »Brisanje lokacije«

1. Administrator ekipe klikne na gumb »Izbriši« zraven izbrane lokacije.

2. Sistem prikaže potrditveno okno.

3. Administrator dogodkov potrdi brisanje s klikom na »Potrdi«.

4. Sistem izbriše igralca iz baze, ponovno se izvede glavni tok.

#### Podtok »Podroben ogled lokacije«

1. Sistem pokliče funkcijo sistema za določanje lokacije.

2. Sistem prikaže podatke o lokaciji ter položaj lokacije na zemljevidu.

3. Administrator dogodkov po ogledu lokacije klikne gumb »Nazaj«, ponovno se izvede glavni tok.

### Alternativni tokovi

#### Administrator dogodkov vnese napačne podatke

Če pride do napake pri vnosu zahtevanih podatkov pri katerem od podtokov,  sistem izpiše opozorilo ob poljih z napako, Administrator dogodkov pa ima možnost popraviti zahtevane spremembe in nato ponovno potrditi akcijo.

#### Administrator dogodkov prekliče operacijo

Administrator dogodkov lahko s klikom na gumb »Prekliči« prekliče dodajanje, urejanje ali brisanje lokacije.

### Posebne zahteve

Brez.

### Predpogoji

Adminstrator dogodkov mora biti prijavljen v sistem.

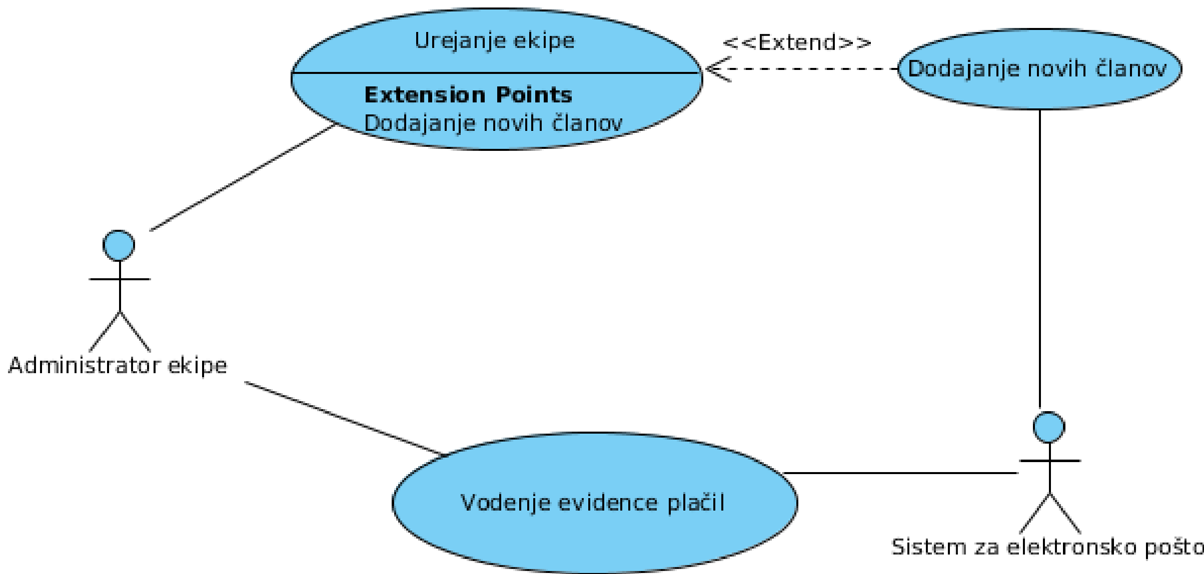
### Popogoji

V primeru uspešno izvedenega podtoka »Dodajanje lokacije«, »Urejanje lokacije« ali »Brisanje lokacije«, so v bazi podatki dodani, posodobljeni oz. izbrisani, v nasporotnem primeru se stanje sistema ne spremeni.

### Razširitvene točke

Brez.

### 6.2.2.5 Diagram primerov uporabe za akterja "Administrator ekipe"




*Slika 14: Diagram primerov uporabe za administratorja ekipe*

#### 6.2.2.5.1 Urejanje podatkov o ekipi

***Kratek opis***

Ta primer uporabe administratorju ekipe omogoča urejati osnovne podatke o ekipi.

***Tok dogodkov***

***Glavni tok***

Primer uporabe se začne, ko Administator ekipe klikne na gumb »Ekipa«. Poleg podatkov, ki se prikažejo navadnemu članu ekipe v okviru PU Pregled ekipe, ima Administrator na voljo dodatne možnosti:

- urejanje igralca – podtok »Urejanje igralca«
- brisanje igralca – podtok »Brisanje igralca«
- urejanje podatkov o ekipi – podtok »Urejanje podatkov o ekipi«

2. Glede na izbiro **Administratorja ekipe** se izvede ustrezen podtok.

***Podtok »Urejanje igralca«***

1. Sistem prikaže formo z istimi polji kot v podtoku »Dodajanje igralca«, le da polja vsebujejo shranjene podatke o igralcu.

2. Administrator ekipe opravi željene spremembe in klikne na gumb »Shrani«.

3. Sistem posodobi podatke za Člana ekipe v podatkovni bazi.

### *Podtok »Brisanje igralca«*

1. Administrator ekipe klikne na gumb »Izbriši« zraven izbranega igralca.

2. Sistem prikaže potrditveno okno.

3. Administrator ekipe potrdi brisanje s klikom na »Potrdi«.

4. Sistem izbriše igralca iz baze.

### *Podtok »Urejanje nastavitev ekipe«*

1. Sistem prikaže vnosno formo za urejanje ekipe:
   - fotografija
   - ime ekipe *
   - šport *
   - biografija

2. Administrator ekipe spremeni željene nastavitve in potrdi spremembe s klikom na »Shrani«

3. Sistem posodobi nastavitve ekipe v podatkovni bazi.

## *Alternativni tokovi*

### *Administrator ekipe vnese napačne podatke*

Če pride do napake pri vnosu zahtevanih podatkov pri katerem od podtokov, sistem izpiše opozorilo ob poljih z napako, administrator ekipe pa ima možnost popraviti zahtevane spremembe in nato ponovno potrditi akcijo.

### *Administrator ekipe prekliče brisanje igralca*

Administrator ekipe lahko prekliče brisanje igralca s klikom na »Prekliči«.

## *Posebne zahteve*

Brez.

## *Predpogoji*

Administrator ekipe mora biti prijavljen v sistem.

## *Popogoji*

Če je uspešno izveden kateri izmed podtokov, so v podatkovni bazi podatki dodani, spremenjeni oz. izbrisani, v nasprotnem primeru se stanje sistema ne spremeni.

## *Razširitvene točke*

- Dodajanje novih članov

#### 6.2.2.5.2 Dodajanje novih članov

***Kratek opis***

Ta primer uporabe omogoča administratorju ekipe vnos novih članov ekipe.

***Tok dogodkov***

***Glavni tok***

***Dodajanje člana ekipe***

1. Sistem prikaže formo z naslednjimi vnosnimi polji:
   - e-pošta *
   - ime
   - priimek
   - vloga("Član ekipe","Administrator ekipe", "Administator dogodkov")
   - sporočilo

2. Administrator ekipe vnese zahtevane podatke in klikne na gumb »Pošlji povabilo«

3. Sistem preveri, ali član ekipe že obstaja oz. ali je že registriran porabnik sistema z vnešenim e-naslovom.

Glede na to sta možna dva uspešna scenarija:

***Člana ekipe in uporabnika sistema z vnešenim e-naslovom še ni v sistemu***

1. Sistem v bazi ustvari novega uporabnika z naključno generiranim geslom ter člana ekipe, oba s statusom "nepotrjen". Na vnešen e-naslov pa pošlje povezavo za potrditev registracije in članstva v ekipi.

2. Ko uporabnik obišče poslano povezavo, sistem prikaže formo za vnos novega gesla:
   - geslo *
   - potrdi geslo *

3. Po kliku na "Potrdi" se status uporabnika in člana ekipe spremeni na "potrjen", sistem pa uporabniku prikaže obrazec, s katerim se lahko prijavi v sistem.

***V podatkovni bazi že obstaja uporabnik z vnešenim e-naslovom***

1. Sistem kreira novega člana ekipe s statusom "nepotrjen". Na vnešen e-naslov pa pošlje povezavo za potrditev članstva.

2. Ko uporabnik obišče poslano povezavo, sistem prikaže prijavni obrazec.

3. Po prijavi sistem uporabniku prikaže obrazec za potrditev članstva.

4. Po kliku na "Potrdi" se status člana ekipe spremeni na "potrjen", sistem pa uporabnika preusmeri na njegovo domačo stran.

### *Alternativni tokovi*

#### *Administrator ekipe ne vnese vseh zahtevanih podatkov*

Če administrator ekipe ne vnese vseh zahtevanih podatkov, sistem izpiše obvestilo o napaki. Administrator ekipe lahko nato opravi zahtevane spremembe in ponovno pošlje vabilo.

#### *V ekipi že obstaja član z vnešenim e-naslovom*

V tem primeru sistem prikaže obvestilo o napaki, administator ekipe pa ima možnost vnesti novega člana ekipe z drugim e-naslovom.

#### *Uporabnik ne potrdi registracije in članstva v ekipi*

Če uporabnik ne potrdi registracije in članstva v ekipi v roku enega tedna, sistem izbriše uporabnika in člana ekipe.

#### *Uporabnik ne potrdi članstva v ekipi*

Če uporabnik ne potrdi članstva v ekipi v roku enega tedna, sistem člana ekipe zbriše.

### *Posebne zahteve*

Brez.

### *Predpogoji*

Administrator ekipe mora biti prijavljen v sistem.

### *Popogoji*

Če je uspešno izveden prvi podtok glavnega toka, sta v podatkovni bazi ustvarjena nov uporabnik ter član ekipe, v primeru uspešne izvedbe drugega podtoka pa samo član ekipe. V primeru, da uporabnik ne potrdi registracije oz. članstva v ekipi, se v podatkovni bazi nahajata član ekipe in/ali uporabnik s statusom "nepotrjen", dokler sistem po enem tednu ne poskrbi za izbris.

V ostalih primerih se stanje sistema ne spremeni.

### *Razširitvene točke*

Brez.

## 6.2.2.5.3 Urejanje plačil

### *Kratek opis*

Ta primer uporabe Administratorju ekipe omogoča pregled nad plačili npr. za nakup sezonske karte za igrišče, nove žoge, plačilo prijavnine za tekmovanja, lige, itd. .

### Tok dogodkov

#### Glavni tok

Primer uporabe se začne, ko Administrator ekipe klikne na gumb »Plačila«.

1. Sistem prikaže seznam vseh plačil v tekočem letu, kjer so vnosi oblike:
   - datum začetka pobiranja vplačil
   - datum, do katerega morajo biti vsa vplačila izvedena
   - namen plačila
   - celoten znesek plačila
   - dosedaj zbrani znesek

2. Uporabnik se lahko odloči tudi za pregledovanje plačil v preteklih letih, dodajanje,urejanje ali brisanje plačila, pri čemer se izvede ustrezni podtok:
   - »Pregledovanje plačil
   - »»Dodajanje plačila«
   - »Urejanje plačila«
   - »Brisanje plačila«.

#### Podtok »Pregledovanje plačil«

1. Administrator ekipe izbere leto, za katero želi pregledati plačila(v izbirnem okencu so na voljo letnice od takrat, ko je bila ustvarjena ekipa, do tekočega leta)
2. Sistem prikaže seznam vseh plačil v izbranem letu, oblika vnosov je enaka kot pri 1. točki glavnega toka.

#### Podtok »Dodajanje plačila«

1. Sistem prikaže formo za vnos novega plačila z naslednjimi polji:
   - namen plačila *
   - opis
   - znesek *
   - rok plačila

   Pod to formo se nahaja forma za pregled in urejanje vplačil posameznih igralcev oblike:
   - id igralca
   - ime in priimek igralca
   - datum plačila
   - znesek plačila – če igralec še ni plačal, ima polje privzeto vrednost znesek/št. igralcev, v nasprotnem primeru pa znesek, ki ga je na datum plačila poravnal
   - v dobro (v primeru, da se spreminja število igralcev v ekipi, se znesek, potreben za plačilo zmanjuše ali povečuje)
   - plačal/ni plačal

2. Administrator ekipe vnese zahtevane podatke in potrdi s klikom na »Potrdi«

3. Sistem shrani vnešene podatke o plačilu.

#### *Podtok »Urejanje plačila«*

1. Sistem prikaže enako formo kot v podtoku »Dodajanje plačila«, vnosna polja pa vsebujejo shranjene vrednosti.

2. Administrator ekipe opravi željene spremembe ali potrdi plačilo izbranega Člana ekipe, nato klikne gumb »Potrdi«.

3. Sistem posodobi podatke v podatkovni bazi.

#### *Podtok »Brisanje plačila«*

1. Administrator ekipe klikne na gumb »Izbriši« zraven plačila, ki ga želi izbrisati.

2. Sistem prikaže potrditveno okno.

3. Po potrditvi sistem izbriše podatke o izbranem plačilu.

### *Alternativni tokovi*

#### *Administrator ekipe vnese napačne podatke*

Če Administrator ekipe pri podtoku **Dodajanje plačila** ali podtoku **Urejanje plačila** ne vnese zahtevanih podatkov oz. so vnešeni podatki napačni, sistem izpiše obvestilo o napaki. Administrator ekipe nato lahko opravi zahtevane spremembe in jih nato potrdi ali pa prekliče urejanje plačil, nakar se ponovno izvede **Glavni tok**.

#### *Administrato ekipe prekine izvajanje podtoka*

Če administrator ekipe pri podtoku **Dodajanje plačila** ali podtoku **Urejanje plačila** klikne na gumb »Prekliči«, se do tedaj opravljene spremembe razveljavijo in ponovno se izvede **Glavni tok**.

### *Posebne zahteve*

Brez.

### *Predpogoji*

Administrator ekipe mora biti prijavljen v sistem.

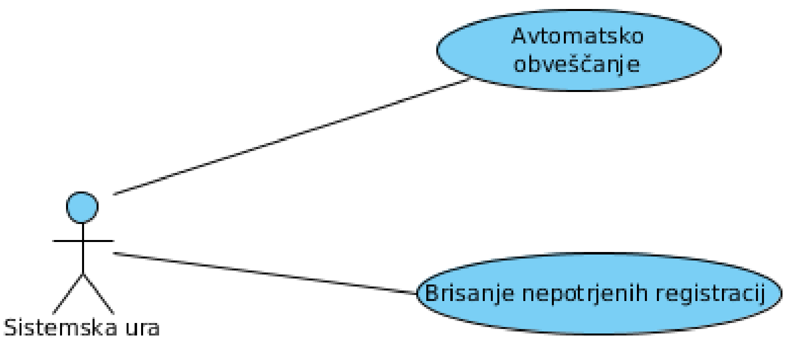
### *Popogoji*

Če je uspešno izveden kateri izmed podtokov **Dodajanje plačila**, **Urejanje plačila** ali **Brisanje plačila,** so podatki bazi dodani, spremenjeni oz. izbrisani. V nasprotnem primeru se stanje sistema ne spremeni.

### *Razširitvene točke*

Brez.

### 6.2.2.6 Diagram primerov uporabe za akterja "Sistemska ura"




*Slika 15: Diagram primerov uporabe za akterja »Sistemska ura«*

#### 6.2.2.6.1 Avtomatsko obveščanje

***Kratek opis***

Ta primer uporabe se sproži ob določenih časovnih terminih, omogoča pa obveščanje članov ekipe o bližajočih se dogodkih.

***Tok dogodkov***

***Glavni tok***

1. Sistemska ura ob določenem terminu sproži ukaz, ki obišče naslov URL pogleda za avtomatsko obveščanje.

2. Sistem poišče seznam ekip, ki imajo v prihodnjih dneh planirane dogodke.

3. Sistem za vsako ekipo:
    a. Preveri, ali nastavitve ekipe obveščanje dovoljujejo.
    b. Sestavi seznam igralcev, ki jim bo poslano sporočilo (glede na nastavitve lahko izloči npr. poškodovane igralce)
    c. Sistem pošlje sporočilo o dogodku vsem igralcem na seznamu.

***Posebne zahteve***

V sistemskem upravljalcu poslov (angl. *job scheduler)* mora biti definirano ponavljajoče se opravilo.

***Predpogoji***

Brez.

***Popogoji***

Brez.

***Razširitvene točke***

Brez.

##### 6.2.2.6.2 Brisanje nepotrjenih registracij

***Kratek opis***

Ta primer uporabe omogoča brisanje registracij, ki niso bile potrjene v določenem času.

***Tok dogodkov***

***Glavni tok***

1. Sistemska ura ob določenem terminu sproži ukaz, ki obišče naslov URL pogleda za avtomatsko brisanje registracij.

2. Sistem poišče seznam vseh registracij, ki niso bile potrjene v pravem času.

3. Sistem izbriše potekle registracije.

***Posebne zahteve***

V sistemskem upravljalcu poslov (angl. *job scheduler)* mora biti definirano ponavljajoče se opravilo

***Predpogoji***

Brez.

***Popogoji***

Če je primer uporabe uspešno izveden, so iz podatkovne baze odstranjene nepotrjene registracije, v nasprotnem primeru se stanje sistema ne spremeni.

***Razširitvene točke***

Brez.

### 6.2.3 Nefunkcionalne zahteve

#### 6.2.3.1 Cilji

Dodatne specifikacije vključujejo zahteve, ki jih v primerih uporabe ni lahko zajeti v modelu primerov. Dodatne specifikacije in model primerov uporabe skupaj zajemajo vse zahteve sistema.

#### 6.2.3.2 Obseg

Dodatne specifikacije definirajo nefunkcionalne zahteve sistema, kot so zanesljivost, uporabnost, učinkovitost, varnost in omejitve pri načrtovanju.

#### 6.2.3.3 Uporabnost

Sistem naj ima preprost in uporabniku prijazen vmesnik, saj je poenostaviti upravljanje z ekipami glavni cilj sistema.

#### 6.2.3.4 Zanesljivost
Sistem naj bo na voljo 98% časa. Podatki ne smejo biti izgubljeni, zato se mora redno izvajati varnostno kopiranje podatkovne baze.

#### 6.2.3.5 Učinkovitost
Sistem mora v začetni fazi podpirati istočasen dostop srednjega števila uporabnikov (okoli 100), po prvem mesecu uporabe pa se preveri statistiko obiskov in po potrebi poveča kapaciteto pomnilnika in/ali procesorske moči. Sistem mora biti zasnovan tako, da omogoča prilagodljivost v primeru visokega povečanja obiska.

#### 6.2.3.6 Vzdržljivost
Sistem mora biti robusten in razširljiv v skladu s potrebami in željami uporabnikov sistema.

#### 6.2.3.7 Varnost
Sistem mora biti varen pred vdori, različni tipi uporabnikov pa smejo imeti dostop do podatkov samo v skladu s pravicami, ki izhajajo iz njihove vloge.

#### 6.2.3.8 Omejitve pri načrtovanju
Sistem bo tekel na strežniku z operacijskim sistemom *linux*. Povezoval se bo z vmesnikom za pošiljanje elektronske pošte, v prihodnost pa je načrtovana tudi uporaba prehoda za pošiljanje SMS sporočil. Podpirati mora vse večje brskalnike (Internet Explorer 7+, Mozilla firefox, Opera, Google Chrome, Apple Safari).

### 6.2.4 Arhitektura

#### 6.2.4.1 Opis arhitekturnega vzorca MVC

Arhitekturni vzorec model-pogled-krmilnik oz. MVC (*angl. Model-View-Controller*) je med najpogosteje uporabljanimi vzorci na področju razvoja spletnih sistemov. Omogoča neodvisnost med podatkovno in predstavitveno domeno ter uporabniškimi akcijami. To nam omogoča ločen razvoj in testiranje posameznih komponent ter ponovno uporabljivost. [36]

Pri tem vzorcu imamo tri tipe komponent:

- **Model:**

  Vsebuje podatke in logiko za upravljanje teh podatkov. Odziva se na poizvedbe o svojem stanju s strani pogleda in spremenemeni svoje stanje glede na ukaze s strani krmilnika.
- **Pogled:**

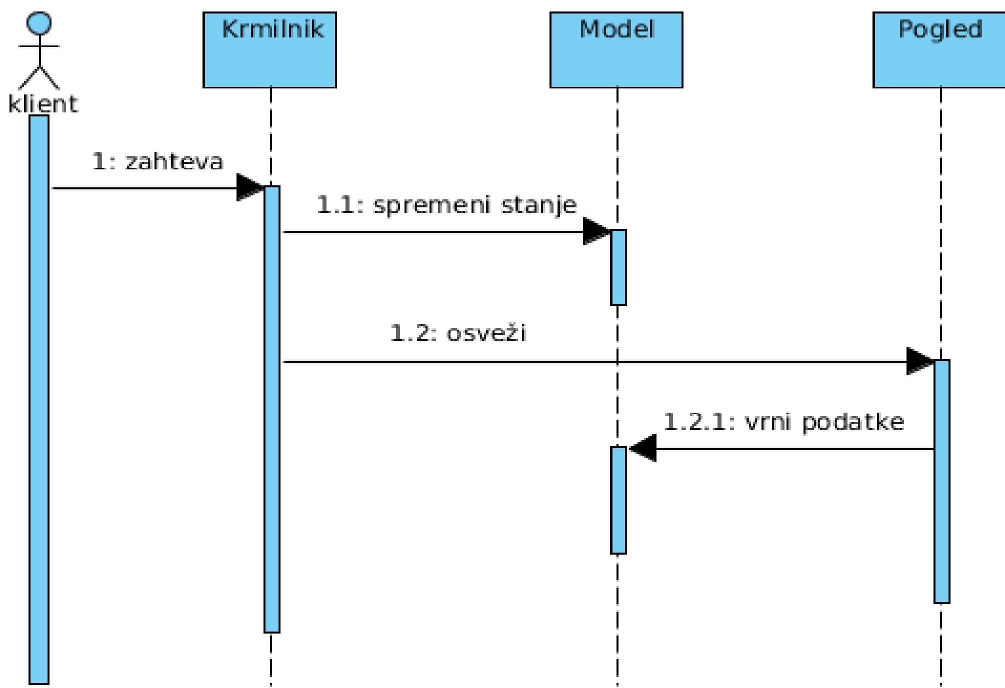
  Osredotoča se na prikaz modela uporabniku.
- **Krmilnik:**

  Krmilnik interpretira vhodne zahteve s strani uporabnika in jih posreduje pogledu ter

modelu, ki opravita ustrezne spremembe.

V grobem jih lahko razdelimo na dva osnovna tipa:

- **tip 1 (pasivni):**
  Pri tem tipu model ne obvešča pogleda o spremembi svojega stanja. Krmilnik po spremembi stanja objekta obvesti pogled, da se je model spremenil in je potrebna osvežitev. Pri spletnih aplikacijah se zaradi narave HTTP protokola uporablja predvsem ta tip vzorca MVC.




*Slika 16: Potek dogodkov pri pasivnem vzorcu MVC*

- **tip2 (aktivni):**
  Pri aktivnem tipu pa model sam obvesti pogled o spremembi stanja. Ponavadi ta povezava ni neposredna, ampak s pomočjo vzorca »opazovalec« (angl. Observer). Model v tem primeru vzdržuje seznam pogledov, ki so naročeni na spremembe njegovega stanja in jih ob spremembi obvesti.

### 6.2.4.2 Vzorec MVC v ogrodju Grok

Sama definicija MVC vzorca sicer dopušča različne interpretacije, tako da poznamo različne izpeljanke. V skupnostih, ki uporabljajo tehnologijo Zope ter v skupnosti priljubljenega ogrodja Django so mnenja, da vzorec MVC ni najbolj primeren za opis dogajanja pri spletnih aplikacijah. Večina implementacij ima namreč težave z ločitvijo krmilnika in pogleda. Njihova interpretacija ne vsebuje krmilnika, ampak samo model in pogled, pri čemer je html

predloga samo implementacijska podrobnost pogleda[8]. Avtorji ogrodja BFG utemeljujejo svoj pogled na vzorec MVC z dvema dejstvoma:

- pogled ne komunicira vedno neposredno z modelom, ampak v veliko primerih krmilnik pripravi podatke za lažji prikaz v pogledu
- če krmilnik vrne podatke v formatu JSON, html predloga ni potrebna. Kaj v tem primeru predstavlja pogled?

#### 6.2.4.2.1 Pretvorba naslova URL v akcije

V osnovi poznamo dva načina za pretvorbo naslova URL v ustrezne akcije:

- usmerjanje (angl. routing) ter
- obhod  (angl. traversal).

##### *Usmerjanje*

Pri tem načinu se naslov URL pretvori v klic ustreznega krmilnika(ali pogleda), ki glede na parametre v naslovu pridobi ustrezne podatke iz podatkovne baze in generira odgovor. Tako bi lahko npr. naslov *ekipe/7/igralci/2* sprožil klic krmilnika, ki bi na podlagi identifikacijske številke ekipe ter igralca vrnil model igralca in ga nato prikazal s pomočjo pogleda. Ta način je uporaben predvsem v kombinaciji z relacijskimi bazami, saj je primeren za objavo vsebin, ki niso hierarhično urejene, prednost tega pristopa pa je tudi, da imamo na enem mestu pregledno definirane vse možne poti.

##### *Obhod drevesa objektov*

Pri tem načinu pa ni eksplicitnega vzorca, ki bi določen naslov pretvoril v ustrezni krmilnik oz. pogled. Namesto tega naslov URL predstavlja hierarhijo oz. drevo objektov in sistem potuje od objekta do objekta in za zadnji objekt pokliče ustrezni pogled - v primeru *ekipe/7/igralci/2*  bi se tako sprožil pogled z imenom 'index' [9] za igralca št. 2. V primeru *ekipe/7/igralci/2/uredi* pa bi se sprožil pogled z imenom 'uredi', ki bi npr. generiral formo html za urejanje podatkov o igralcu. Omenjeni princip je podoben kot mape in datoteke na trdem disku. Mape so v tem primeru objekti, ki lahko vsebujejo druge objekte, imenujemo jih *vsebniki(angl. containers).* Dateke pa predstavljajo listi drevesa, ki jih imenujemo *modeli*. Vsak objekt v hierarhiji ima tudi povezavo na starša preko atributov __name__ (ime objekta v vsebniku) ter __parent__ (referenca na sam vsebnik), torej ima definirano lokacijo v hierarhiji. To nam med drugim tudi olajša omejevanje dostopa do posameznih segmentov aplikacije. Članu ekipe lahko npr. dodelimo dostop samo  do tistega poddrevesa, ki ima v vozlišču njegovo ekipo. Poleg tega ima ta način pretvorbe naslovov URL tudi to prednost, da je objava hierarhije objektov poljubne globine zelo preprosta.

Vendar v končni fazi samo poimenovanje določenih elementov vzorca MVC ni tako pomembno, koncepti so v obeh primerih enaki. V nadaljevanju bom uporabljal nestandarno poimenovanje tipov komponent vzorca, da bo v skladu s pogledi ustvarjalcev ogrodja.

---

8 Avtorji Djanga so zato svoje ogrodje slikovito opisali kot ogrodje MTV (angl. *Model-Template-View)*

9 Če ne navedemo imena pogleda, sistem poskuša prikazati pogled s privzetim imenom index.

#### 6.2.4.3 Prilagoditev arhitekturnega vzorca potrebam

Arhitekturni vzorec MVC je primeren za razvoj naše aplikacije, saj je podprt s strani ogrodja. Manjši odmik od formalne definicije si bom dovolil pri komunikaciji med predlogo in pogledom. Zaradi narave implementacije vzorca MVC v ogrodju Grok je kontekst predloge vedno samo en objekt, včasih pa je potrebno prikazati podatke tudi drugih, ki so s povezani s kontekstom. Predloga naj bi vsebovala čim manj poslovne logike, zato se bo iskanje in prilagoditev objektov za prikaz izvajalo v okviru pogleda, predloga pa bo komunicirala s pogledom samo za pridobitev rezultatov.

Komponentna arhitektura Zope omogoča gradnjo fleksibilnih sistemov z rahko povezanimi sestavnimi deli, zato bo vsa komunikacija med posameznimi kompontami potekala preko vmesnikov.

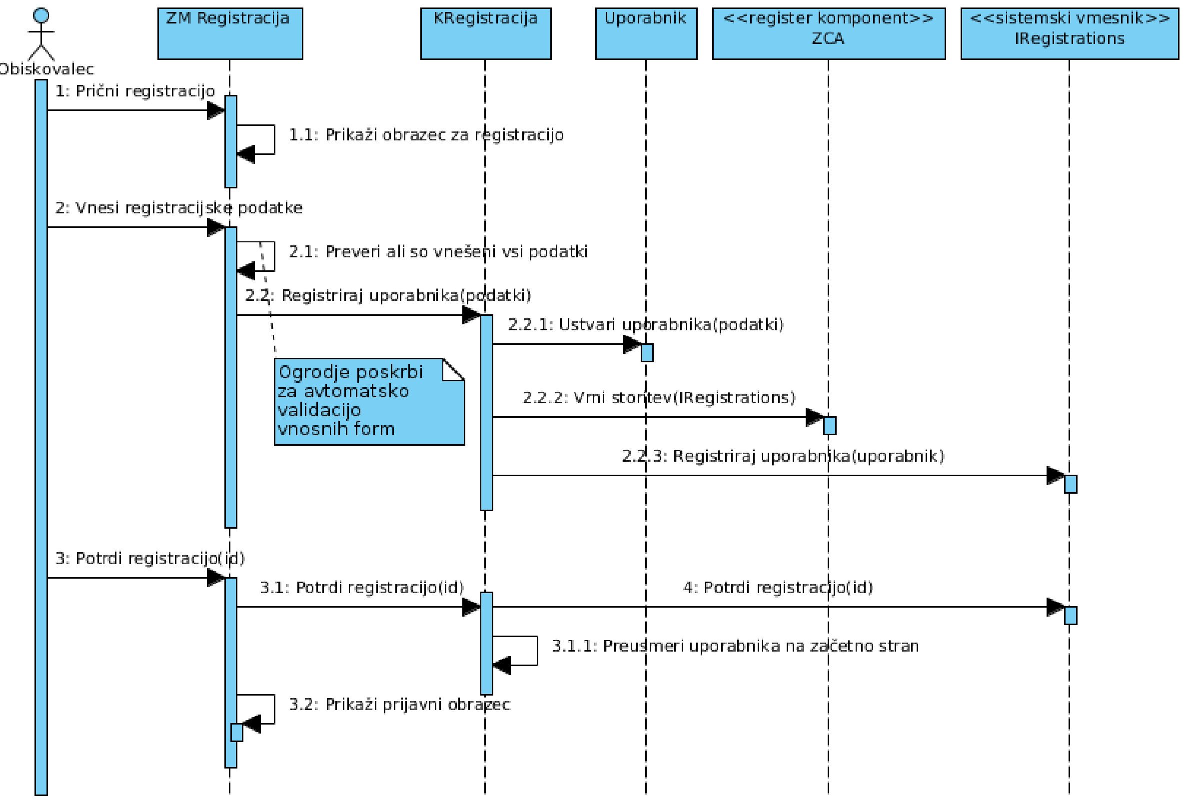
Izjemo bom naredil le pri kreiranju novih podatkovnih entitet. Lahko bi uporabili tovarne in odpravili odvisnost med pogledom in entiteto tudi v tem primeru, vendar dodaten napor ni smiseln. V primeru morebitne zamenjave implementacije entitete namreč ne bi bilo pretežko zamenjati samo tistega pogleda, ki ustvari entiteto. Ostalih pogledov (za urejanje, brisanje entitete itd.) nam ne bi bilo potrebno ponovno implementirati, saj bodo z entiteto povezani samo preko vmesnika.

Cilj je tudi ponovno uporabiti že obstoječe komponente, kjer je to mogoče, in poiskati priložnosti za ponovno uporabo posameznih delov naše aplikacije.

### 6.2.5Diagrami zaporedja

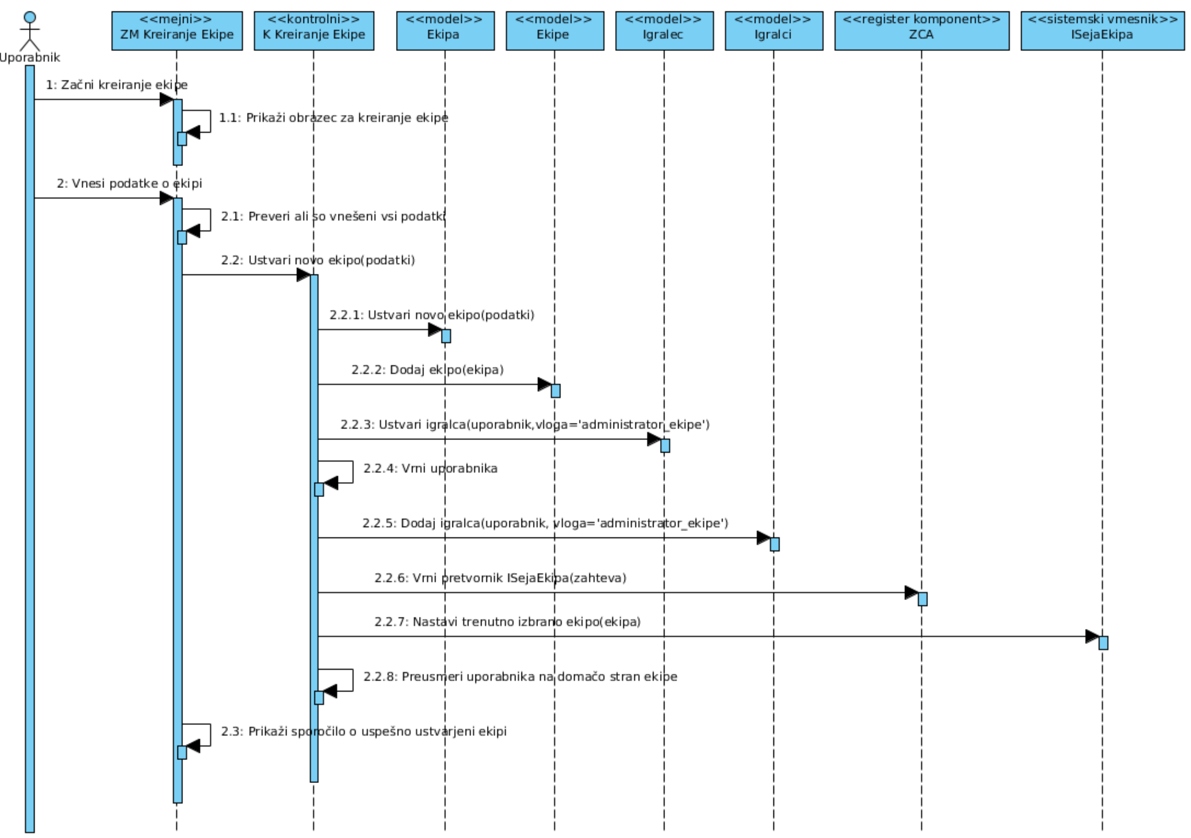
Diagram zaporedja prikazuje interakcije na osnovi časovnega zaporedja. Prikazuje, kako objekti sodelujejo in izmenjujejo sporočila v interakcijah na osnovi časovnih linij.[37]
V nadaljevanju podajam zanimivejše diagrame zaporedja za prikaz interakcij z registrom komponent.

### 6.2.5.1 Registracija



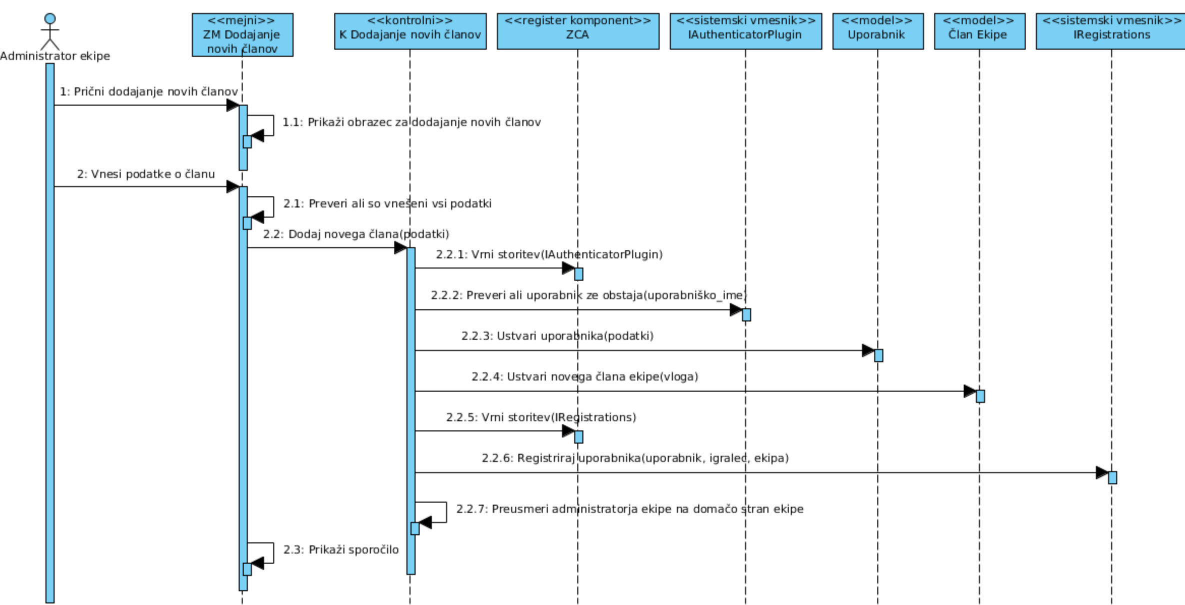

*Slika 17: Diagram zaporedja za primer uporabe »Registracija«*

## 6.2.5.2 Vnos ekipe



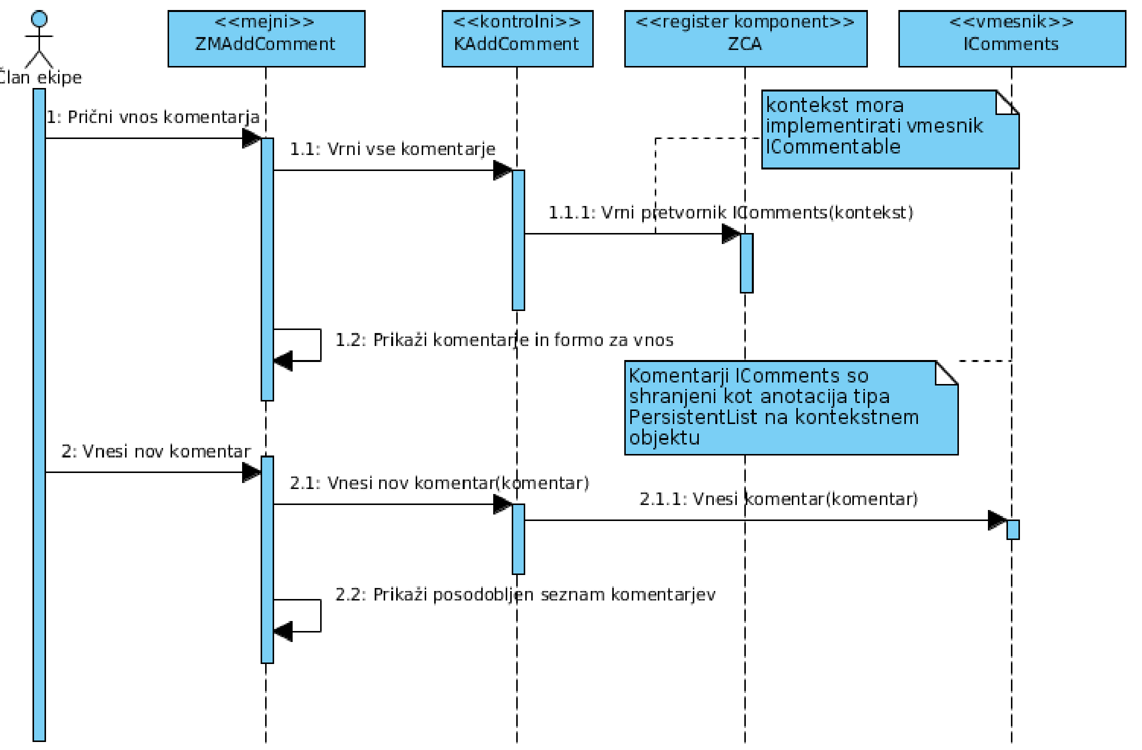

*Slika 18: Diagram zaporedja za primer uporabe "Vnos ekipe"*

### 6.2.5.3 Dodajanje novih članov



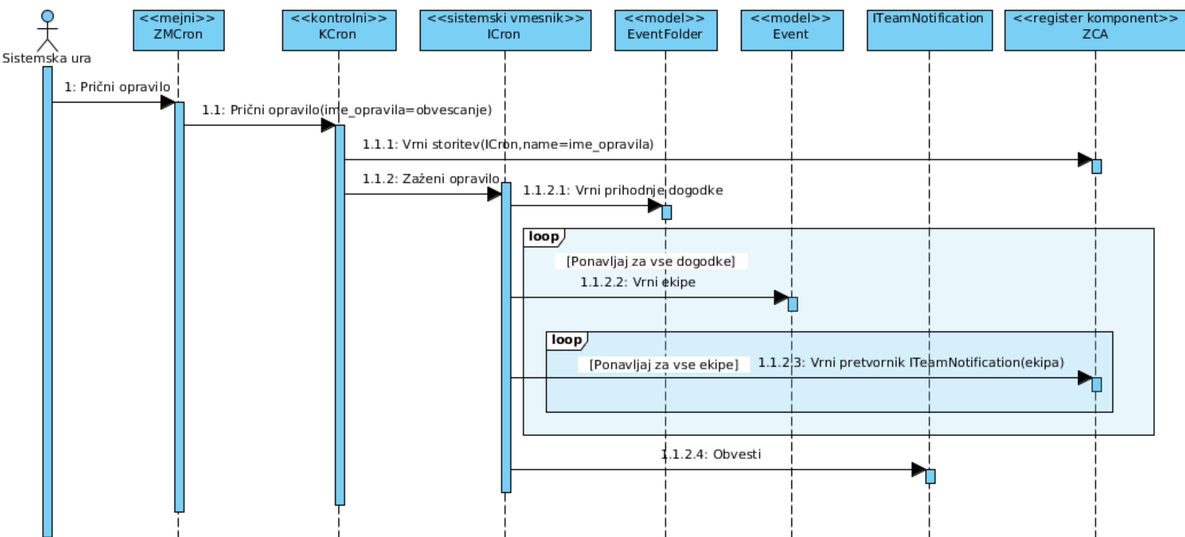

*Slika 19: Diagram zaporedja za primer uporabe "Dodajanje novih članov"*

### 6.2.5.4 Vnos komentarjev




*Slika 20: Diagram zaporedja za primer uporabe Vnos komentarjev*

### 6.2.5.5 Avtomatsko obveščanje




*Slika 21: Diagram zaporedja – Obveščanje pred dogodkom*

## 6.2.6Podatkovni model

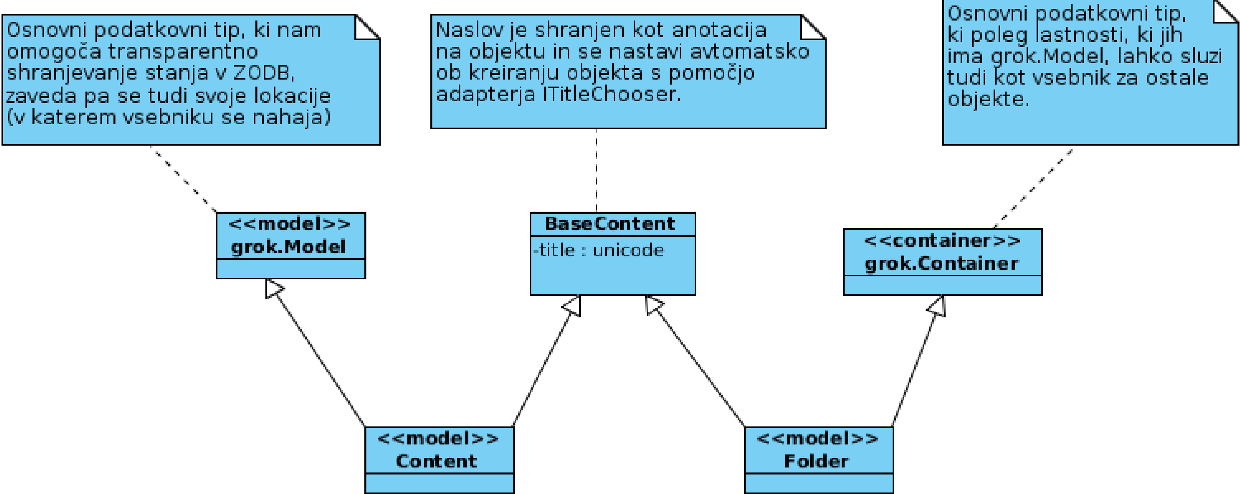
Značilnosti načrtovanja podatkovnega modela pri uporabi platforme Zope in baze ZODB:

- zaradi dinamične narave jezika Python atributom ni potrebno definirati tipa. Kljub temu bi bilo to morda smiselno, saj pogosto pri definiciji vmesnika entitete atributom določimo tip in ostale omejitve, kar nam omogoča avtomatsko validacijo in generiranje form

- pri objektni bazi ZODB imamo dva osnovna tipa podatkovnih entitet: vsebnike (angl. *containters*) in navadne podatkovne entitete, t.i. modele (angl. *models*). Razlikujejo se v tem, da vsebniki lahko vsebujejo ostale modele in vsebnike. Aplikacija pri ogrodju Grok je poseben tip vsebnika, ki poleg ostalih entitet vsebuje še lokalni register komponent.

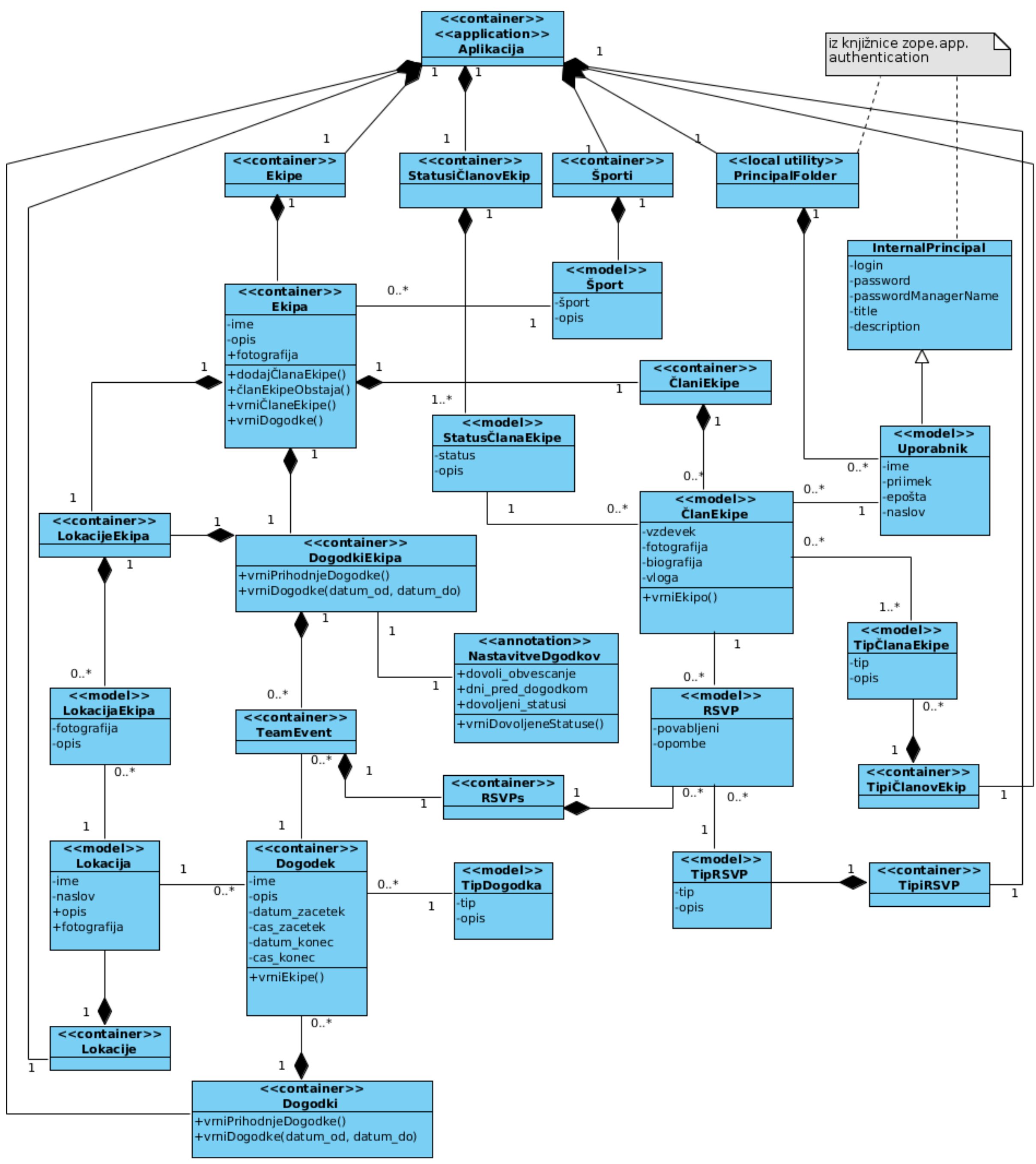
- poseben načina pretvarjanja naslova URL v akcije, ki je značilen za platformo Zope, vpliva tudi na načrtovanje podatkovnega modela, saj se hierahija podatkovnih entitet neposredno odraža v samem naslovu in tako željena oblika naslovov URL pogosto narekuje hierarhijo objektov.

- programski jezik Python nima pravih privatnih metod in atributov, zato ne uporabljamo standardnih metod *get* in *set* za pridobivanje oz. nastavljanje vrednosti atributov. Vsi so atributi javno dostopni, le po konvenciji se ime privatnih atributov začne z znakom _ (npr. _privatni_atribut), kar spoštujejo tudi orodja za dopolnjevanje kode, ki teh atributov ne prikažejo. Za dostop do objektov v vsebniku lahko uporabljamo standardno sintakso za dostop do elementov slovarja. Tako lahko npr. do določenega člana ekipe dostopamo z ukazom *clani_ekipe['ime_clana_ekipe']* Posledično imajo naši podatkovne entitete definiranih majhno število metod.




*Slika 22: Osnovna tipa entitet pri ogrodju Grok*

*Razširili smo ju z atributom* title, ki je namenjen prikazu v navigaciji.




*Slika 23: Podatkovni model aplikacije*

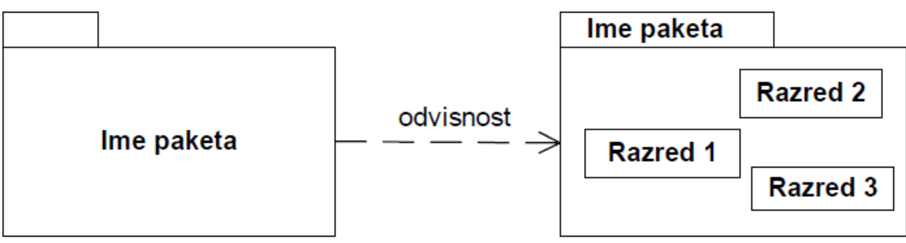
*Aplikacija je korensko vozlišče, ki vsebuje primerke ostalih entitet in lokalne storitve.*

### 6.2.7 Diagrami paketov

Paket predstavlja grupiranje razredov ali drugih elementov UML modela v visokonivojske enote z namenom zmanjšanja kompleksnosti modela.

Med dvema elementoma obstaja odvisnost, če sprememba definicije enega elementa povzroči spremembo drugega. Obstaja več vrst odvisnosti med razredi:

- razred pošlje sporočilo drugemu razredu (odvisnost tipa <<call>>)
- razred dostopa do določenih atributov drugega razreda (odvisnost tipa <<friend>>),
- razred kreira objekte drugega razreda (odvisnost tipa <<instantiate>>). [37]




*Slika 24 - Diagram paketov*

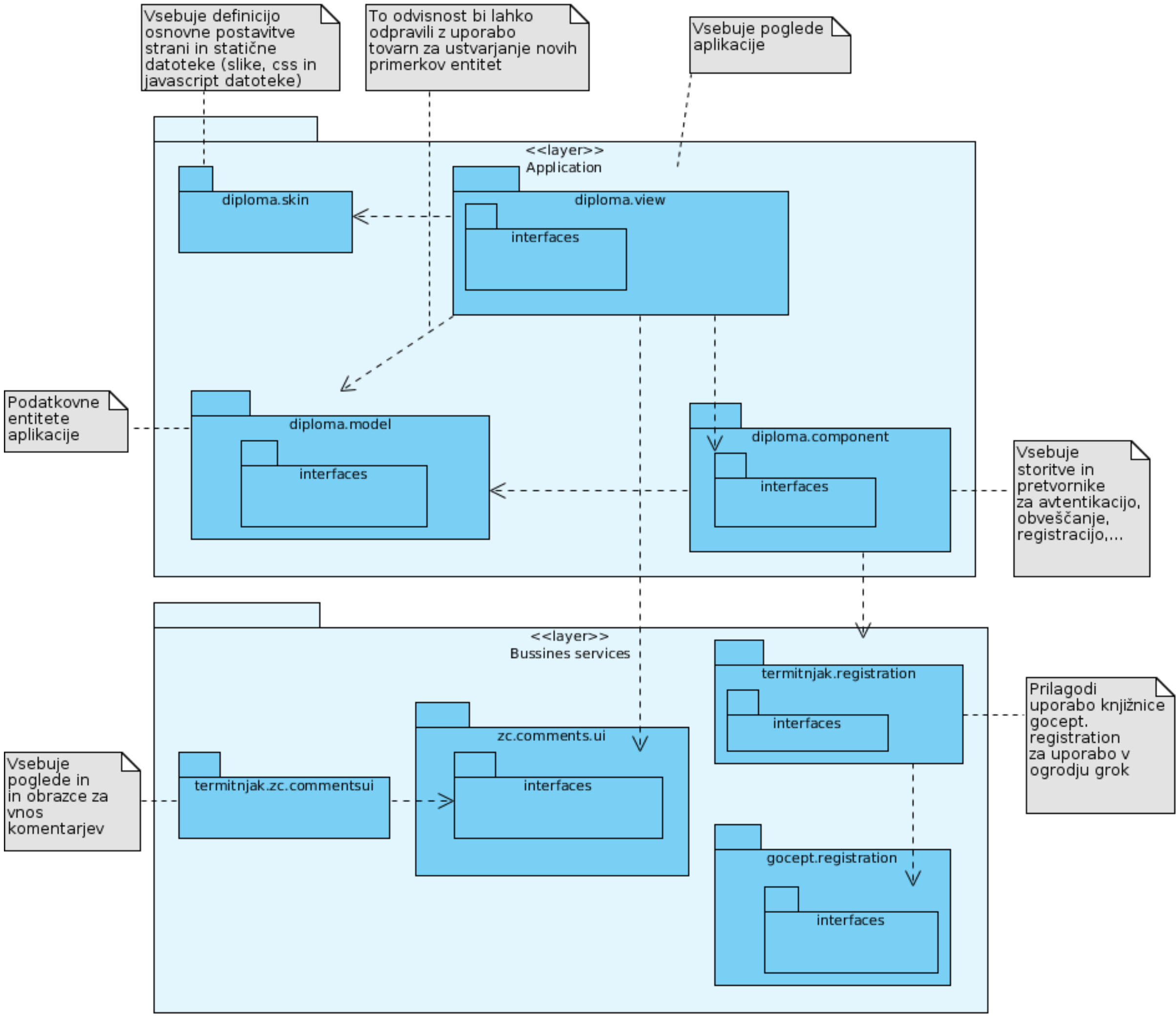
Na spodnji sliki podajam pregled statične zgradbe sistema v obliki paketov in odvisnosti med njimi. Kjer je narisana povezava med določenim paketom in paketom *interfaces* pomeni, da je odvisnost šibka, saj je paket odvisen samo od definicije vmesnika in ne od specifične implementacije. Naša aplikacija je v osnovi razdeljena na štiri dele:

- diploma.view – vsebuje poglede s predlogami
- diploma.model – vsebuje podatkovne entitete
- diploma.component – vsebuje razne koristne storitve in pretvornike
- diploma.skin – vsebuje komponente, povezane z osnovnih izgledom strani

Pri razvoju sem identificiral tudi komponente, ki jih lahko ponovno uporabimo, zato sem jih prestavil v ločena paketa.

- termitnjak.registration – prilagodi knjižnico gocept.registration za uporabo v ogrodju Grok.
- termitnjak.zc.commentsui – vsebuje prikazne komponente za knjižnico zc.comments. Prikazne komponente te knjižnice so namreč namenjene prikazu v administrativnem vmesniku ZMI, ki se v ogrodju Grok ne uporablja.

Komponente obeh paketov so s komponentami prilagojenih knjižnic povezane samo preko vmesnikov.




*Slika 25 – Diagram paketov za celotno aplikacijo*

## 6.3 Razvoj aplikacije v praksi

### 6.3.1 Priprava razvojnega okolja

Za uspešen razvoj informacijskih sistemov je poleg uporabe ustrezne metodologije pomembna tudi izbira orodij, ki nam omogočijo učinkovito izvajanje razvojnih ter podpornih aktivnosti. Sam sem pri razvoju uporabljal naslednja orodja:

#### 6.3.1.1 SVN

Preizkušeno orodje za nadzor nad verzijami. V zadnjem času se sicer vse bolj uveljavljajo distribuirana orodja (git, mercurial, ...), pri katerih imamo namesto zadnje verzije lokalno shranjen repozitorij s celotno zgodovino, v primerjavi s svn-jem pa ponuja večjo hitrost, skalabilnost in lažje ustvarjanje ter združevanje vejitev. Vendar omenjene lastnosti v mojem

primeru niso tako pomembne, poleg tega pa že imam izkušnje z orodjem svn, zato sem se odločil za slednjega.

#### 6.3.1.2 Visual Paradigm UML

Orodje CASE, ki ponuja standarden nabor funkcij za delo z UML diagrami. Plačljiva različica omogoča tudi generiranje kode iz modelov ter usklajevanje kode in modelov(angl. *round-trip*), slednje žal samo za programski jezik java.

#### 6.3.1.3 Eclipse z dodatkom pydev

Eclipse je uveljavljeno integrirano razvojno orodje (*angl. IDE - Integrated development environment*), ki preko vtičnikov ponuja podporo za razvoj v različnih programskih jezikih. Za jezik Python potrebujemo dodatek Pydev, ki omogoči funkcije, kot so barvanje in avtomatsko dopolnjevanje kode.

#### 6.3.1.4 Redmine

Redmine je orodje za vodenje projektov, komunikacijo med člani razvojne ekipe in sledenje napakam (angl. issue tracking). Pri razvoju sem ga uporabljal za nadzor nad napakami, ki jih je potrebno odpraviti ter za vodenje seznama funkcionalnosti, ki bi jih bilo dobro podpreti.

#### 6.3.1.5 Sphinx

Sphinx je orodje, ki omogoča pol-avtomatsko generiranje dokumentacije na podlagi vhodnih datotek v reStructuredText formatu ter besedila v izvorni kodi. ReStructuredText je preprost označevalni jezik, podobno kot HTML ali latex, vendar s poudarkom na berljivosti.
Izhodna datoteka je lahko v formatu HTML (tudi Windows HTML Pomoč), latex, pdf ali navadni tekstovni datoteki. Vizualna podoba je že v osnovi privlačna, lahko pa jo prilagodimo s pomočjo jezikov css in html. Poleg tega je besedilo z izvorno kodo obarvano, kar močno prispeva k berljivosti.
Predvsem format HTML je zelo uporaben, saj Sphinx s pomočjo preprostih označb v izvorni datoteki ustvari hierarhično strukturo dokumentacije s povezavami, kot tudi povezave med izbranimi elementi v samem tekstu. Ustvari tudi globalni indeks vsebine ter indeks vseh modulov, omogoča pa tudi iskanje po strani.

#### 6.3.1.6 Buildout

Skupnost Python za distribucijo koristnih knjižnic uporablja centraliziran repozitorij, ki se nahaja na naslovu http://pypi.python.org/. Trenutno vsebuje skoraj enajst tisoč knjižnic, večinoma v formatu, ki ga skupnost imenuje *egg*. Poleg izvorne kode se v paketu nahajajo še nekatere druge datoteke z metapodatki o sami knjižnici, kot je npr. verzija knjižnice in seznam knjižnic, od katerih je odvisna. To nam v navezi z orodji kot so npr. *easy_install* iz knjižnice *setuptools* ter *distribute* omogoča, da knjižnico namestimo z enim ukazom, vključno z vsemi podpornimi knjižnicami.

Problem se pojavi v primeru, ko imamo na enem računalniku veliko različnih projektov, vsak pa potrebuje različno verzijo določene knjižnice. Z omenjenimi orodji se knjižnica namreč namesti v mapo, kjer se nahaja sistemska namestitev Python-a in tako prihaja do konfliktov med verzijami.

Ena od rešitev je uporaba orodja *virtualenv,* s katerim lahko ustvarimo poljubno število virtualnih okolij Python, tako da lahko knjižnice namestimo ločeno od sistemske namestitve Pythona.
Za preproste projekte ta pristop zadostuje, pri večjih projektih pa je priporočena uporaba orodja *buildout,* ki so ga razvili pri podjetju Zope.

*Buildout* podobno kot kombinacija *virtualenv+easy_install* omogoča vzpostavitev izoliranega razvojnega okolja, vendar ponuja veliko dodatnih možnosti, s katerimi je zagotovljena ponovljivost vzpostavitve okolja.

```
project/
  bootstrap.py
  buildout.cfg
  .installed.cfg
  parts/
  develop-eggs/
  src/
  bin/
    buildout
    mypython
  eggs/
  downloads/
```

*Primer: Tipična datotečna struktura projekta, ki uporablja buildout*

Ko končamo z razvojem, lahko na preprost način vzpostavimo enako okolje tudi na testnem ali produkcijskem strežniku. Potrebujemo samo dve datoteki, ki sta zgoraj obarvani modro:

- *bootstrap.py:*
  To datoteko potrebujemo za vzpostavitev datotečne strukture in okolja *buildout*.

- *buildout.cfg:*
  To je glavna konfiguracijska datoteka. Vsebuje seznam knjižnic, ki jih potrebujemo, in razne konfiguracijske parametre. Pomembno in močno orodje okolja *buildout* so razširitve v obliki receptov (angl. *recipes*). Z njihovo pomočjo lahko na preprost način namestimo in konfiguriramo različne strežnike (paste, apache, ldap, zeo, ...), ustvarimo skripte za testiranje, varnostno kopiranje podatkovne baze, avtomatsko generiranje dokumentacije iz kode itd. [10]

Po zagonu skripte *bin/buildout* se iz centralnega strežnika in ostalih virov prenesejo ustrezne knjižnice v mapo *eggs/*[11]. V mapi *parts/* in *bin/* recepti ustvarijo potrebne datotek in skripte, vključno s skripto za zagon prilagojene verzije interpreterja Python, ki vsebuje vse knjižnice v mapi *eggs* ter našo izvorno kodo.

10 Seznam pomembnejših receptov se nahaja na strani http://www.buildout.org/docs/recipelist.html.
11 Če so zapakirane v formatu .zip,tar.gz ipd., se najprej prenesejo v mapo *downloads/.*

V mapi *develop-eggs/* se nahajajo povezave do map, kjer se nahaja naša izvorna koda, ponavadi je to kar mapa *src/,* datoteka *.installed.cfg* pa vsebuje podatke o trenutno nameščenih knjižnicah.

### 6.3.1.7 Namestitev ogrodja Grok

Za uspešno namestitev ogrodja je potrebno najprej namestiti Python ter orodje easy_install. Začetno strukturno nove aplikacije preprosto ustvarimo z orodjem *paster.* To orodje temelji na predlogah, ki glede na tip projekta ustvari ustrezne mape in datoteke in izvede ostale aktivnosti, ki so potrebne za vzpostavitev okolja.

Za delo v ogrodju Grok imamo na voljo ustrezno predlogo z imenom grok: s klicem ukaza *paster create -t grok ime_projekta* v ukazni vrstici se ustvari novo buildout okolje, v mapi *src/* pa ogrodje aplikacije. Za preprostejše ustvarjanje novih projektov si lahko namestimo tudi programsko skripto *grokproject***,** ki pokliče orodje paster z ustreznimi argumenti. Nov projekt tako ustvarimo preprosto s klicem ukaza *grokproject ime_projekta***.** Nato v novo ustvarjeni mapi zaženemo ukaz *bin/buildout,* ki prenese zahtevane knjižnice in izvede namestitev.

Za zagon strežnika uporabimo ukaz *bin/paster serve pot_do_konfiguracijske_datoteke*. Ponavadi imamo za razvoj in postavitev različni konfiguracijski datoteki. Pri razvoju uporabljamo datoteko *parts/etc/debug.ini,* pri postavitvi pa *parts/etc/deploy.ini.* Ukazu serve lahko pri podamo še argument - - *reload*, kar omogoči avtomatski ponovni zagon strežnika ob spremembi katere od datotek z izvorno kodo.

Ukaz grokproject za lažji začetek v mapi *src/* ustvari tudi preprosto predstavitveno aplikacijo, ki jo lahko namestimo s pomočjo administrativnega vmesnika. Do vmesnika dostopamo na naslovu, ki je definiran v datoteki buildout.cfg, privzeto nastavitev je *http://localhost:8080/.* Primerek aplikacije ustvarimo tako, da vnesemo ime aplikacije in pritisnemo na gumb »Create«. Naša aplikacija je nato dostopna na naslovu *http://localhost:8080/ime-aplikacije.*

Slika 26: Administrativni vmesnik Grok

## 6.3.2 Izvedba primerov uporabe

### 6.3.2.1 Podatkovna plast

Tabela 2: Seznam podatkovnih entitet aplikacije

| Diploma | Korensko vozlisce, ki vsebuje ostale entitete in register lokalnih komponent. |
|---|---|
| Teams | Vsebnik za ekipe. |
| TeamMemberStatuses | Vsebnik za statuse clanov ekip. |
| Sports | Vsebnik za objekte tipa Sport. |
| PrincipalFolder | Osnovna implementacija vsebnika in avtentikatorja za entitete tipa IInternalPrincipal. |
| InternalPrincipal | Principal je pri platformi Zope osnovni subjekt, ki nastopa v varnostnih interakcijah s sistemom. InternalPrincipal je trajnostna implementacija, ki subjekte shranjuje v ZODB. |
| Sport | Sport (nogomet, kosarka,...) |
| Team | Predstavlja športno ekipo. |
| TeamMembers | Vsebnik za clane ekipe. |
| TeamMemberStatus | (pripravljen na igro, poskodovan, odsoten,...) |
| User | Registrirani uporabnik sistema. |
| TeamMember | Potrjeni clan ekipe. |
| TeamEvents | Vsebnik za ekipne dogodke. |
| TeamLocations | Vsebnik za ekipne lokacije. |
| TeamMemberType | Tip clana ekipe (igralec, trener, ...) |
| EventSettings | Razred za shranjevanje nastavitev za dogodke(nastavitve avtomatskega obvescanja, ...) |
| TeamLocation | Ekipna lokacija. |
| RSVP | Prisotnost clana ekipe na dolocenem dogodku. |
| TeamEvent | Ekipni dogodek. |
| TeamMemberTypes | Vsebnik za tipe clanov ekip. |

Za vsak entitetni razred praviloma definiramo tudi vmesnik, ki opisuje atribute in operacije razreda. Kot smo omenili že v poglavju o komponentni arhitekturi, so vmesniki koristni iz večih razlogov:

- predstavljajo priročno dokumentacijo za programerje
- omogočajo rahlo povezavo med komponentami
- na podlagi vmesnikov lahko avtomatsko generiramo in validiramo vnosne obrazce

### 6.3.2.2 Plast poslovne logike

#### 6.3.2.2.1 Pogledi

Pogledi pri platformi Zope opravljajo tudi vlogo krmilnika. V poglavju o arhitekturi smo govorili o razlogih za nestandardno poimenovanje. Za razliko od večine ostalih ogrodij pogled (oz. krmilnik) ni odgovoren za iskanje ustrezne podatkovne entitete, ki je posredovana predlogi za prikaz. Pogled je komponenta, ki nadgradi podatkovno komponento s predstavitveno zmožnostjo.[7] V tehničnem smislu je pogled večkratni pretvornik, ki prilagodi kontekstni objekt (primerek določene podatkovne komponente) ter zahtevo. Pogled ima ponavadi tudi pripadajočo predlogo, ki skrbi za prikaz podatkov.

##### *6.3.2.2.1.1 Vnosni obrazci*

Poleg navadnih pogledov imamo na voljo posebne vrste pogledov, ki omogočajo prikaz, dodajanje, brisanje in urejanje podatkovnih entitet. Uporaba avtomatiziranega generiranja in validiranja vnosnih obrazcev zelo pospeši razvoj aplikacije. Spodaj podajam primer za dodajanje in urejanje lokacij.

Vsak obrazec mora imeti definiran vsaj kontekst, tj. tip objekta, za katerega je prikazan, ter polja, ki so vezana na definicijo atributov v vmesniku.

```
class AddLocation(DiplomaForm):
    grok.context(ILocationFolder)
    form_fields = grok.Fields(ILocation)

    @grok.action('add')
    def add(self, **data):
        location = Location()
        self.applyData(location, **data)
        self.context[location.name] = location

        self.flash(_("Location added."))
        self.redirect(self.url(self.context))

class EditLocation(DiplomaEditForm):
    grok.context(ILocation)
    form_fields = grok.Fields(ILocation)
```

*Primer 14: Prikaz uporabe vnosnih form pri ogrodju Grok*

##### *6.3.2.2.1.2 Dostop do podatkov*

V pogledu  lahko dostopamo do kontekstnega objekta preprosto s klicem *self.context.* Zaradi hierahične ureditve podatkovne baze imamo dostop tudi do vseh njegovih sinov, pri čemer uporabljamo standardno sintakso slovarjev jezika Python.

Če želimo izvajati učinkovite poizvedbe pri velikem številu objektov, ki se nahajajo na poljubni lokaciji v hierarhiji, je potrebno uporabiti storitev ICatalog. Za uporabo kataloga je

potrebno najprej definirati ustrezne indekse.

```
class EventIndexes(grok.Indexes):
    grok.site(IDiploma)
    grok.context(IEvent)
    start_date = grok.index.Field()
    end_date = grok.index.Field()
    description = grok.index.Text()
```

*Primer 15: Definiranje indeksov za dogodke*

Na podlagi definiranih indeksov lahko nato izvajamo željene poizvedbe.

```
catalog = component.getUtility(ICatalog)
self.results = catalog.searchResults(start_date=datetime(2010,14,10))
```

*Primer 16: Uporaba kataloga za iskanje dogodkov*

##### 6.3.2.2.2 Storitve in pretvorniki

Tabela 3: Komponente plasti poslovne logike

| **Vmesnik** | **Opis** |
| --- | --- |
| diploma.component.interfaces.ICron | Definira splošno administrativno opravilo, ki se izvede s pomočjo storitve *cron* v okviru operacijskega sistema linux. |
| diploma.component.interfaces.IMailer | Globalna storitev za pošiljanje e-pošte |
| diploma.component.interfaces.ITeamSession | Pretvornik za shranjevanje in pridobivanje trenutne ekipe iz seje. |
| z3c.objpath.interfaces.IObjectPath | Globalna storitev za določanje absolutne poti objekta, ki je potrebna za uporabo knjižnice za relacije. |
| zope.pluggableauth.interfaces.ICredentialsPlugin | Storitev za ekstrakcijo uporabnikovih prijavnih podatkov iz zahteve. Za avtentikacijo uporabimo že obstoječo implementacijo vmesnika IAuthenticatorUtility (PrincipalFolder) |

#### 6.3.2.3 Predstavitvena plast

Pri ogrodju Grok je privzeti jezik za izdelavo dinamičnih predlog TAL/TALES, lahko pa uporabljamo tudi druge, npr. Jinja, Mako itd. Za ponovno uporabljivost predlog imamo na voljo dva osnovna mehanizma[7, 11]:

- *makri*:
  Makri so majhne enote predstavitvene kode, namenjene ponovni uporabi v drugih predlogah. Definiramo jih s pomočjo jezika Macro Expression Template Attribute Language (METAL).

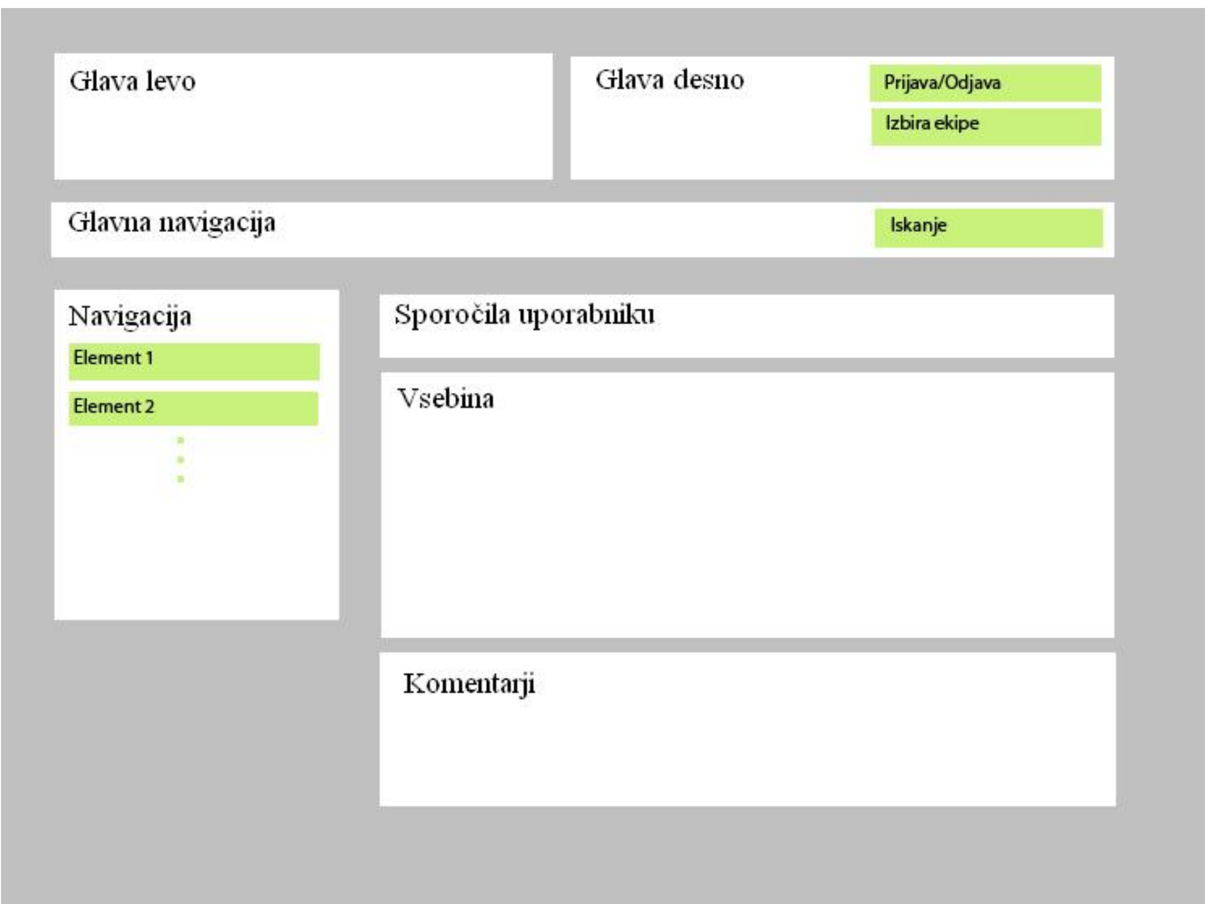
- *ponudniki vsebin (angl. content providers):*
  Makre je v veliki meri zamenjala uporaba ponudnikov vsebin. Podobno kot makri tudi ponudniki vsebin predstavljajo enote predstavitvene kode, vendar so v primerjavi z njimi prilagodljivejši. Registrirani so za določen tip vsebin ali pogledov[12], prikaz pa lahko podobno kot pri pogledih dodatno omejimo z definiranjem pravic, ki jih mora imeti uporabnik.

  Vsak ponudnik vsebin vsebuje poljubno število manjših ponudnikov vsebin (angl. *viewlets)*, zato jih imenujemo tudi upravljalci manjših ponudnikov vsebin (angl. *viewlet managers).* Naloga upravljalcev je agregiranje, filtriranje, urejanje vrstnega reda in prikaz manjših ponudnikov vsebin. [7, 11]:

##### 6.3.2.3.1 Osnovna postavitev

Za izvedbo dinamičnih predlog predstavitvene plasti sem uporabljaljezik TAL/TALES, za vključitev posameznih sestavnih delov strani pa ponudnike vsebin. Osnovna postavitev naše aplikacije je tako sestavljena iz množice upravljalcev, ki vsebujejo posamezne manjše ponudnike vsebin.

[12] Ponudnik vsebin je realiziran kot večkratni pretvornik, ki kot argumente sprejme kontekstni objekt, zahtevo ter pogled.




*Slika 27: Osnovna struktura strani.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      lang="en"
      i18n:domain="diploma">

<head>
        <tal:block tal:content="structure provider:headslot" />
</head>
<body>
<div id="bg-wrapper">
        <div id="wrapper">
                <div id="header">
                        <div id="header-left" tal:content="structure provider:header-left" />
                        <div id="header-right" tal:content="structure provider:header-right" />
                        <div class="clear"></div>
                </div>
                <div id="navigation" tal:content="structure provider:navigation" />
                <div id="contentWrapper">
                  <div id="content">
                    <div id="columnRight">
                      <div id="messages"  tal:content="structure provider:messages" />
                      <div id="documentContent" tal:content="structure view/content" />
                      <div id="comments" tal:content="structure provider:comments" />
                    </div>
                    <div id="columnLeft">
                      <tal:content tal:content="structure provider:actions" />
                    </div>
                  </div>
                </div>
              </div>
        </div>
    </body>
</html>
```

*Primer 17:Glavna predloga z osnovno postavitvijo*

###### 6.3.2.3.1.1 Prikaz vsebine podatkovne entitete

Za prikaz vsebine podatkovne entitete najpogosteje uporabljamo kar avtomatiziran prikaz s pomočjo prikaznih form:

```
class TeamView(grok.DisplayForm):
   grok.context(ITeam)
   grok.name('index')
   grok.require("diploma.ViewTeam")
   form_fields = grok.Fields(ITeam).omit('__parent__')
   form_fields['image'].custom_widget = ImageDisplayWidget
```

*Primer 18: Implementacija prikazne forme za ekipo*

Privzeto predlogo, ki jo ogrodje uporablja za prikaz, lahko po želji tudi prilagodimo. Po konvenciji predloge shranjujemo v mapo z imenom, ki je kombinacija imena modula in niza *_templates*. Tako npr. predloge za ekipo shranjujemo v mapo *team_templates*.

```
<table class="listing">
<thead>
   <tr>
   <th class="label-column"> </th>
   <th> </th>
   </tr>
</thead>
<tbody>
   <tal:block repeat="widget view/widgets">
   <tr tal:define="odd repeat/widget/odd"
      tal:attributes="class python: odd and 'odd' or 'even'">
      <td class="fieldname">
      <tal:block content="widget/label"/>
      </td>
      <td>
      <input tal:replace="structure widget" />
      </td>
   </tr>
   </tal:block>
</tbody>
<tfoot>
   <tr class="controls">
   <td colspan="2" class="align-right">
      <input tal:repeat="action view/actions"
      tal:replace="structure action/render" />
   </td>
   </tr>
</tfoot>
</table>
```

*Primer 19: Privzeta predloga prikazne forme*

###### 6.3.2.3.1.2 Navigacija

Pri navigaciji sem si pomagal z uporabo knjižnice megrok.navigation. Navigacija je realizirana kot upravljalec manjših ponudnikov vsebin, ki predstavljajo elemente navigacije.

```
class Actions(grok.ViewletManager):
   grok.name('actions')

class ActionsMenu(megrok.menu.Menu):
   grok.name('actions')
   grok.title('Actions menu')
```

```
class Actions(grok.Viewlet):
    grok.viewletmanager(Actions)

    def getActions(self):
        menu = component.getUtility(IBrowserMenu, 'actions')
        return menu.getMenuItems(self.context, self.request)
```

*Primer 20: Definiranje navigacijskega menija*

```
<ul id="actions" class="acitem">
      <li id="navigation-title">
           <a tal:attributes="href python:view.url(context)"
              tal:content="context/title|nothing">Title</a>
       </li>
       <li tal:repeat="action viewlet/getActions">
            <img class="action-icon" tal:attributes="src action/icon" />
            <a tal:attributes="href action/action"
            tal:content="action/title"></a>
      </li>
  </ul>
```

*Primer 21: Predloga za prikaz navigacije*

Vnos elementa v navigacijo je preprost. Na pogledu, ga želimo prikazati v navigaciji, uporabimo naslednji ukaz :

*megrok.menu.menuitem(name,icon,order)*

- *name:*
  Definiramo lahko več primerkov navigacijskega menija, vsak je enolično določen z imenom.
- *icon:*
  Naslov URL, kjer se nahaja ikona, ki jo želimo uporabiti za prikaz elementa v navigaciji.
- *order:*
  Vrstni red vnosa v navigaciji.

```
class EditEventSettings(grok.EditForm):
    #...
    megrok.menu.menuitem('actions',icon=SETTINGS_ICON_URL,order=100)
```

*Primer 22: Primer definicije elementa navigacijskega menija*

Ker so pogledi vezani na določen kontekst in imajo definirane varnostne zahteve, je bilo zelo preprosto realizirati prikaz možnih akcij v navigaciji glede na vlogo člana ekipe. Vnos je v navigaciji prikazan samo v primeru, ko ima član ekipe ustrezne pravice za ogled določenega pogleda (npr. pogleda za urejanje podatkov o ekipi).

##### 6.3.2.3.1.3 Prikaz sporočil uporabniku

Realizacija prikaza sporočil uporabniku je preprosta. Uporabimo funkcijo *flash,* ki jo definirajo pogledi Grok. Potrebno je samo definirati ponudnika vsebin, ki agregira poslana sporočila in jih nato prikaže s pomočjo predloge.

### 6.3.2.4 Realizacija nefunkcionalnih zahtev

#### 6.3.2.4.1 Varnost

Kot smo že omenili v razdelku 3.3.6, platforma Zope za avtentikacijo uporabnikov ponuja prilagodljivo storitev PAU.

Pri ogrodju grok si lahko uporabo storitve poenostavimo s knjižnico megrok.login. Knjižnica definira in registrira vtičnik za pridobitev prijavnih podatkov iz seje ter vtičnik za avtentikacijo, ki omogoča samoregistracijo uporabnikov, brez potrjevanja.

Omenjene knjižnice pri izdelavi aplikacije nisem uporabil, saj sem hotel implementirati robustnejši sistem za registracijo, poleg tega pa uporaba same storitve PAU ni zapletena. Definirati moramo vtičnik za pridobitev podatkov ter vtičnik za avtentikacijo. Osnovni vtičniki za avtentikacijo na podlagi seje so že implementirani, potrebno jih je samo konfigurirati in integrirati.

Privzeta varnostna politika je definirana v datoteki *parts/etc/site.zcml* in je v nasprotju z varnostno politiko pri ogrodju BlueBream zelo odprto naravnana. Vsak anonimni uporabnik ima dostop do vseh delov aplikacije, če jih izrecno ne zavarujemo. Lahko pa vzpostavimo strožjo začetno politiko, ki anonimnim uporabnikom onemogoči dostop do celotne aplikacije. Nato definiramo lastno varnostno politiko, ki ureja dostope do posameznih segmentov aplikacije.

Postopek definiranja in uporabe lastne varnostne politike v ogrodju Grok je zelo preprost, kar mi je precej olajšalo realizacijo omejevanja dostopa glede na vloge članov ekip.

Potrebni so naslednji koraki:

1. **Definiranje pravic**

Pravice definiramo preprosto z razširjanjem razreda grok.Permission ter prirejanjem imena, ki je enolični identifikator pravice.

```
class ManageTeam(grok.Permission):
    grok.name('diploma.ManageTeam')

class ManageEvents(grok.Permission):
    grok.name('diploma.ManageEvents')
```

*Primer 23: Primer definicije pravic*

## 2. Združevanje pravic v vloge

Pravice lahko združujemo v vloge, kar poenostavi prirejanje pravic uporabnikom.

```
class TeamAdmin(grok.Role):
    grok.name('diploma.TeamAdmin')
    grok.permissions('diploma.EditProfile','diploma.View',
                'diploma.ViewTeam',
                'diploma.AddTeam','diploma.EditTeam',
                'diploma.ManagePlayers','diploma.ManageEvents',
                'diploma.ManagePayments','diploma.ManageTeam')
```

*Primer 24: Primer združevanja pravic v vloge*

## 3. Zaščita pogledov

Medtem ko nekatera ogrodja omejujejo dostop do podatkovnih entitet (ogrodje Zope 2 omogoča celo določanje pravic dostopa za vsak primerek posebej), so pri ogrodju Grok zaščiteni pogledi. Pogled zaščitimo z uporabo ukaza *grok.requires()* in imenom pravice, ki jo mora imeti uporabnik, da lahko do njega dostopa.

```
class EditTeam(DiplomaEditForm):
    grok.require('diploma.ManageTeam')
```

*Primer 25: Zaščita pogledov z ukazom grok.requires()*

## 4. Omogočanje dostopa uporabnikom do določenega dela aplikacije

V prejšnjem koraku smo omejili dostop do obrazca za urejanje ekipe. Trenutno do njega ne more dostopati nihče[13]. Za vsakega uporabnika je namreč potrebno določiti, do katerih delov aplikacije sme dostopati.

To storimo s prirejanjem vloge uporabniku za določen objekt aplikacije. S tem mu omogočimo dostop do pogledov objekta in pogledov vseh njegovih sinov, seveda če njegova vloga vsebuje pravico, ki je zahtevana za dostop do določenega pogleda.

```
role_manager = IPrincipalRoleManager(team)
role_manager.assignRoleToPrincipal('diploma.TeamAdmin', user.login)
```

*Primer 26: Prirejanje vloge »Adminitrator ekipe« uporabniku za določeno ekipo*

```
role_manager = IPrincipalRoleManager(grok.getApplication())
role_manager.assignRoleToPrincipal('diploma.SiteMember', user.login)
```

*Primer 27: Prirejanje vloge »Uporabnik« novemu uporabniku za celotno aplikacijo*

---

[13] Pravzaprav to velja samo za subjekte, definirane v okviru naše aplikacije. Subjekt 'zope.Manager', ki je definiran v privzeti varnostni politiki, ima namreč dostop do celotne podatkovne baze.

### 6.3.2.5 Implementacija PU

#### 6.3.2.5.1 PU Registracija

Za implementacijo registracije sem uporabil knjižnico *gocept.registration*. Najprej jo je bilo potrebno prilagoditi za uporabo v ogrodju Grok, saj knjižnica ob potrditvi registracije pošlje dogodek, ki ga sistem za upravljanje dogodkov ne zazna. Zato je bilo potrebno ustvariti novo implementacijo dogodka, ki realizira vmesnik IRegistrationConfirmed ter novo implementacijo storitve za registracijo IRegistrations, ki ob registraciji pošlje naš dogodek. Ostale komponente knjižnice lahko ponovno uporabimo, saj niso odvisne od specifične implementacije. Prilagojene komponente sem prestavil v knjižnico termitnjak.registration za ponovno uporabo v prihodnjih projektih.
V modulu za registracijo na aplikacijski ravni (diploma.component.registration.registration) pa se na dogodek potrditve registracije odzovemo tako:

```
@grok.subscribe(IUserRegistration,IRegistrationConfirmed)
def handleUserRegistration(registration,event):
   """After the user confirms his registration, we add
   him to our user folder and assign him the default role
   for site members.
   """
   users = component.getUtility(IAuthenticatorPlugin, 'users')
   user = registration.data['user']
   users[user.login] = user
   role_manager = IPrincipalRoleManager(grok.getApplication())
   role_manager.assignRoleToPrincipal('diploma.SiteMember', user.login)
```

*Primer 28: Implementacija naročnika, ki poskrbi za dodajanje uporabnika v sistem po potrditvi registracije*

#### 6.3.2.5.2 PU Vnos komentarjev

Za vnos komentarjev sem uporabil že obstoječo knjižnico zc.comments. Knjižnica vsebuje tudi poglede za prikaz in vnos komentarjev, vendar niso primerni za uporabo v ogrodju Grok. Zato sem ustvaril lastne predstavitvene komponente in jih prestavil v posebno knjižnico *termitnjak.zc.commentsui*, saj bo komponente moč ponovno uporabiti v prihodnjih projektih. Knjižnica *zc.comments* definira vmesnik *ICommentable*, ki označuje vsebino, ki jo lahko komentiramo. V našo glavno predlogo sem vključil ponudnika vsebine, ki prikaže dogodke, vendar samo na objektih, ki implementirajo ta vmesnik. Deklaracijo implementacije vmesnika lahko opravimo pri definiciji razreda, za katerega želimo omogočiti komentarje, druga možnost pa je prirejanje vmesnika že obstoječim objektom z ukazom alsoProvides(). Komentarji so shranjeni v obliki trajnostnega seznama kot anotacija[14] na objektu, ki implementira vmesnik ICommentable.

```
class CommentsManager(grok.ViewletManager):
```

[14] Anotacije predstavljajo neko dodatno vsebino, ki jo lahko shranimo na vsak objekt, ki implementira vmesnik IAnnotable.

```
    grok.context(Interface)
    grok.name('comments')

class Comments(grok.Viewlet):
    grok.context(ICommentable)
    grok.viewletmanager(CommentsManager)
    grok.order(0)

    def update(self):
        self.comments = IComments(self.context)
```

*Primer 29: Definicija ponudnika vsebine, ki prikaže komentarje*

#### 6.3.2.5.3 PU Avtomatsko obveščanje

Obveščanje članov ekipe pred dogodki sem realiziral z uporabo storitve *cron* v okviru operacijskega sistema linux. Nov posel sem definiral s pomočjo recepta za okolje buildout, ki omogoča definiranje poslov v konfiguracijski datoteki *buildout.cfg*.

```
[crontab_eventnotification]
recipe = z3c.recipe.usercrontab
times = 0 0 * * *
command =

wget -O - --http-user=${debug_ini:user} –http-password=${debug_ini:password}
http://${debug_ini:host}:${debug_ini:port}/${debug_ini:virtual_hostname}/cron?name=even
t-notification
```

Parameter *times* določa kdaj se bo izvedlo obveščanje, v našem primeru imamo nastavljeno ob polnoči vsak dan.

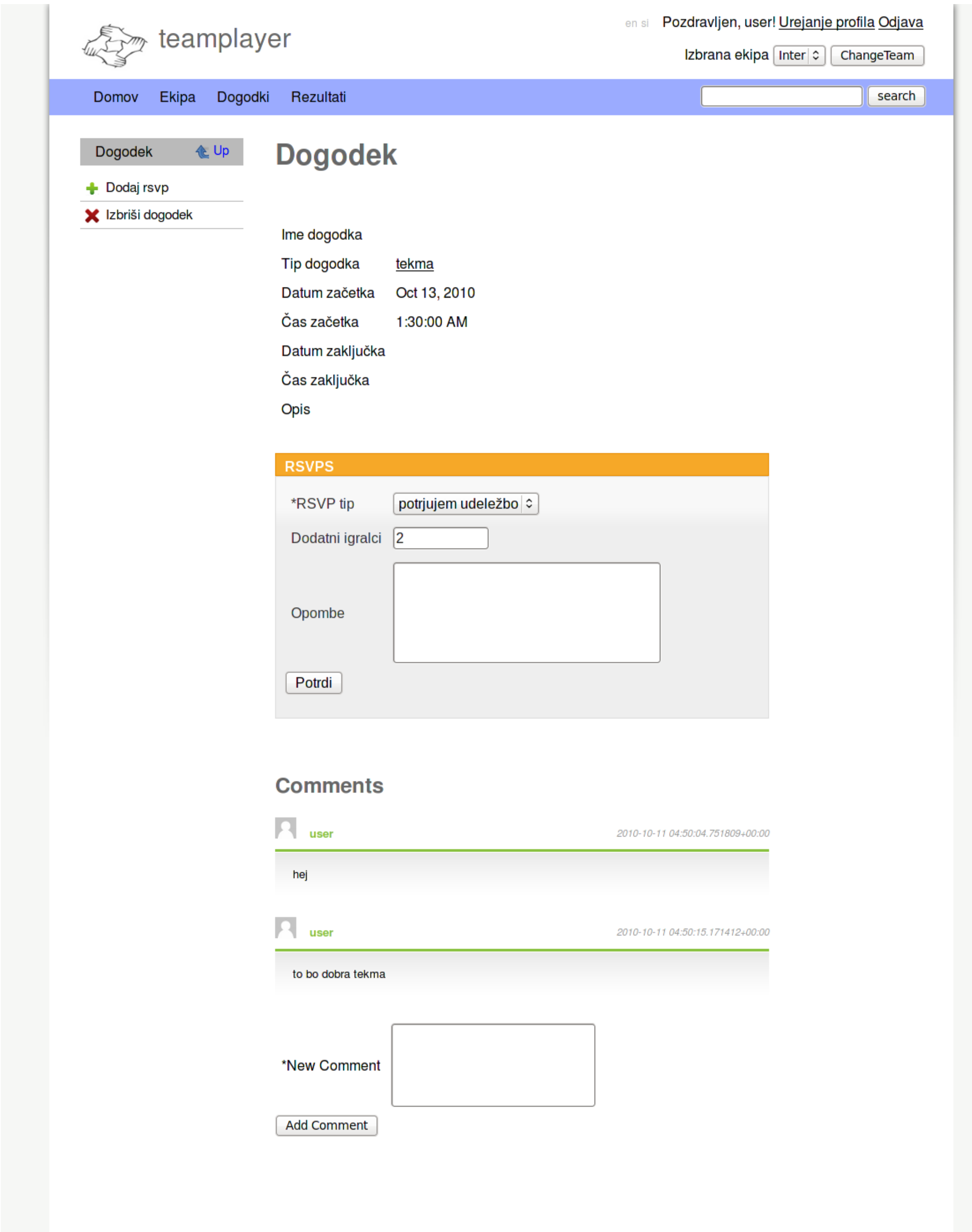
Parameter *command* določa ukaz, ki se bo izvedel. Z ukazom *wget* pokličemo naslov URL naše aplikacije, točneje pogleda z imenom *cron.* Pogled poišče poimenovano storitev z imenom 'event-notification', in jo zažene.

### 6.3.2.5.4 Prikaz generiranih predlog

#### *6.3.2.5.4.1 Prijava*

*Slika 28: Izgled strani za primer uporabe »Prijava«*

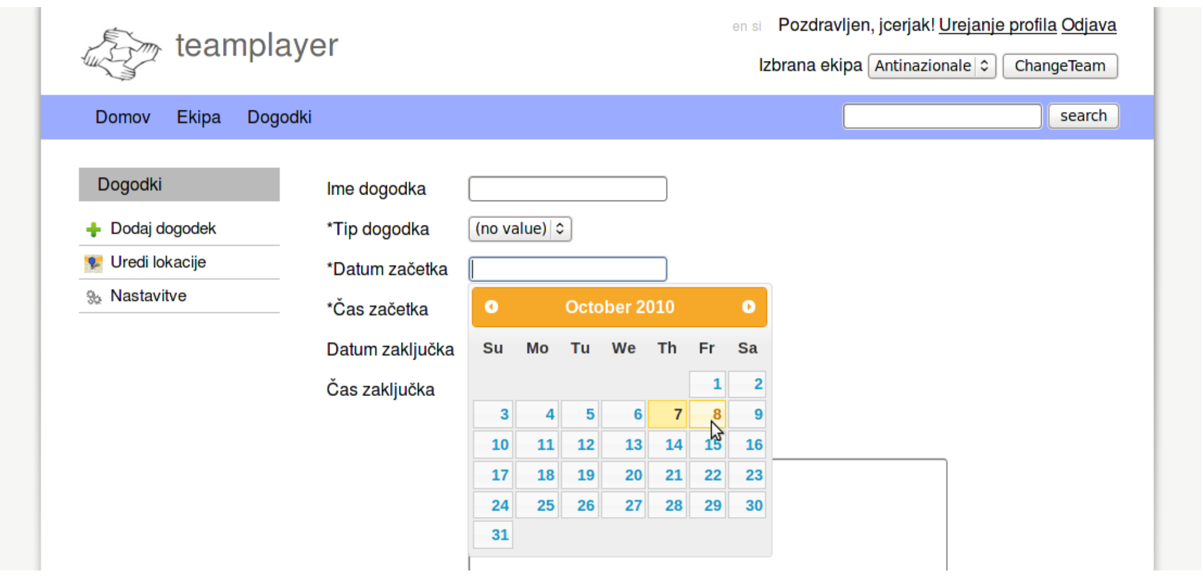
### 6.3.2.5.4.2 Pregled dogodka in urejanje udeležbe




*Slika 29: Izgled strani za primer uporabe »Pregled dogodka in urejanje udeležbe«*

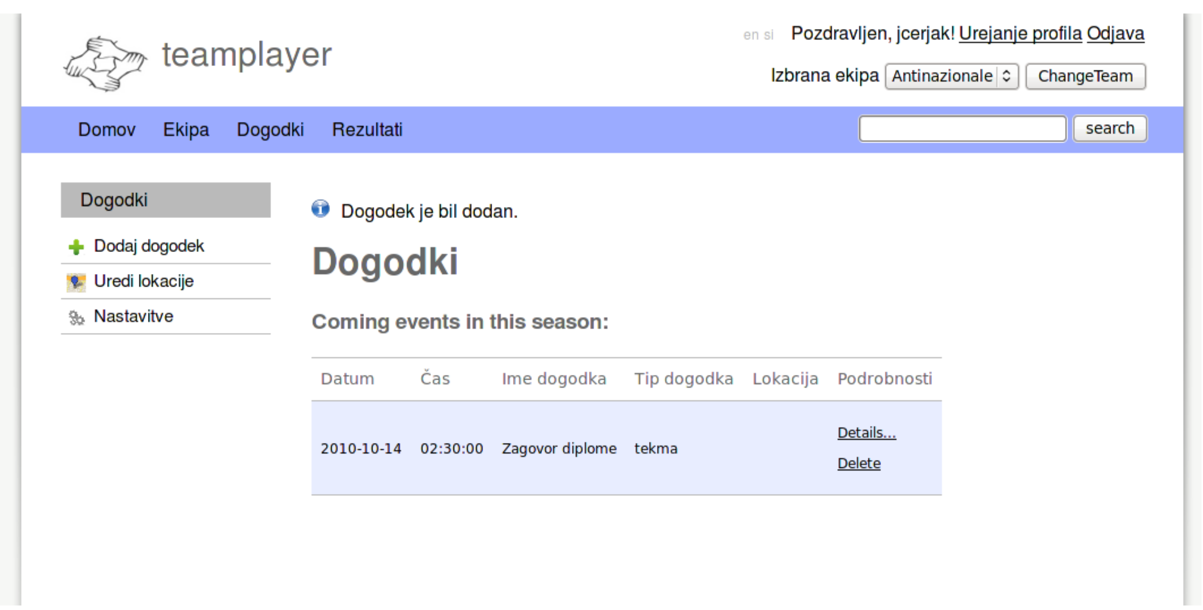
*Komentarji se prikažejo samo na tistih straneh, pri katerih kontekstni objekt implementira vmesnik ICommentable. V našem primeru je to stran z dogodki, lahko pa bi na preprost način vključili komentarje tudi na katero drugo stran s prirejanjem vmesnika ICommentable kontekstnem objektu z ukazom alsoProvides()*

### 6.3.2.5.4.3 Dodajanje dogodka




*Slika 30: Izgled strani za primer uporabe »Dodajanje dogodka«*

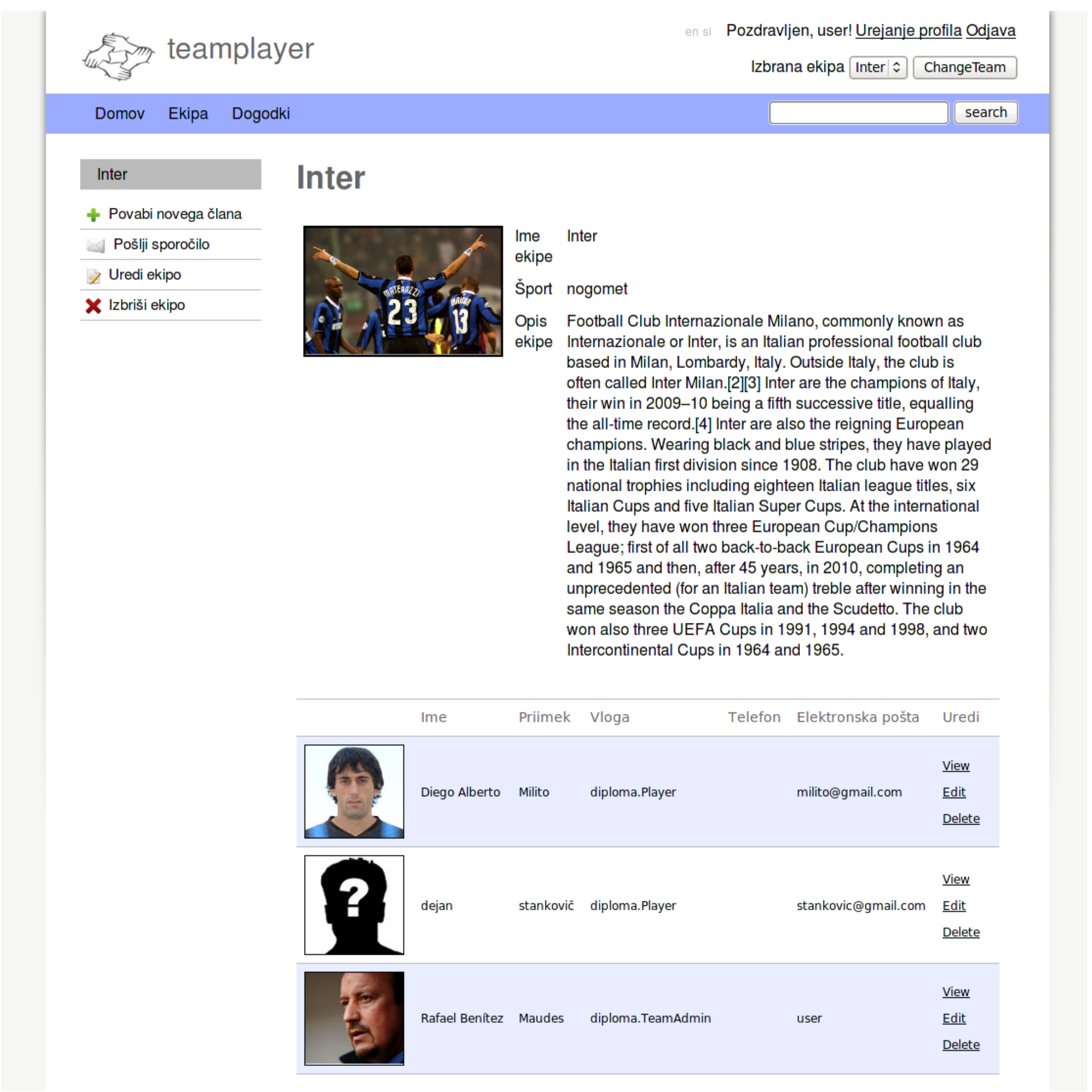
### 6.3.2.5.4.4 Pregled dogodkov




*Slika 31: Izgled strani za primer uporabe »Pregled dogodkov«*

*Prikaz vsebuje tudi sporočilo uporabniku, ki smo ga poslali s klicem metode flash v pogledu.*

### *6.3.2.5.4.5 Urejanje ekipe*




*Slika 32: Izgled strani za primer uporabe »Urejanje ekipe«*

# 7. Sklepne ugotovitve

Spletna aplikacijska ogrodja razvijalcem olajšajo mnoga opravila pri razvoju spletnih sistemov. Ponudba na trgu je velika, zato izbira ustreznega ogrodja ni preprosto opravilo. Pri uporabi vsakega ogrodja je potrebno začetno uvajalno obdobje, ko razvijalec spoznava izbrano platformo, in ponavadi mora uspešno izpeljati več projektov, da lahko izkoristi vse prednosti, ki jih platforma ponuja.

Pri platformi Zope je uvajalno obdobje daljše kot pri primerljivih ogrodjih skupnosti Python, saj mora razvijalec razumeti delovanje komponentne arhitekture in se naučiti rokovanja z objektno bazo. Slednje v mnogih primerih ne predstavlja velike ovire, saj je uporaba baze ZODB praviloma zelo transparentna. Pri razvoju aplikacije sem ugotovil, da je delo s hierarhično urejenimi objekti preprosto. Sintaksa za dostop do objektov je enaka kot pri uporabi slovarja v jeziku Pyhon, podatkovna baza pa v primeru sprememb objektov sama poskrbi za ustrezne posodobitve.

V določenih primerih pa uporaba baze ZODB ni tako transparentna. Problem se pojavi pri realizaciji poljubnih povezav med objekti. Za učinkovito izvajanje poizvedb nad tovrstnimi relacijami je potrebno relacije definirati s pomočjo posebne knjižnice. Tudi izvajanje kompleksnejših poizvedb z veliko pogoji ni tako preprosto, saj mora programer sam poskrbeti za definiranje potrebnih indeksov, poleg tega pa poizvedovalni mehanizem ni tako prilagodljiv kot jezik SQL pri relacijskih bazah. Prednosti baze ZODB so tako najbolj očitne pri delu z manj strukturiranimi podatki, ki jih lahko oblikujemo v poljubno globoke hierarhije.

Glavna prednost platforme je po mojem mnenju komponentna arhitektura Zope, ki omogoča gradnjo prilagodljivih in razširljivih sistemov. Brez prevelikega dodatnega truda pri razvoju smo izdelali aplikacijo, ki omogoča spremembe od zunaj, brez posegov v izvorno kodo. Z uporabo tovarn pa bi dosegli skoraj popolno zamenljivost komponent. Sicer omenjena fleksibilnost pri razvoju sistemov z majhnim številom razvijalcev in različnih postavitev ni tako bistvenega pomena. Izvorno kodo lahko namreč spreminjamo neposredno. Prednost, ki jo v tem primeru ponuja ZCA, je možnost preproste zamenjave komponent pri aktivnosti testiranja. Tako lahko zamenjamo implementacijo storitve za pošiljanje elektronske pošte in namesto, da bi sporočila pošiljali, jih samo izpisujemo na zaslon.

Prilagodljivost, ki jo ponuja komponentna arhitektura, je pomembna predvsem pri razvoju večjih sistemov in komponent, namenjenih za ponovno uporabo. Pri izdelavi predstavljene aplikacije sem lahko prilagodil delovanje že obstoječih komponent za vnos komentarjev in registracijo, saj sta bili razviti z mislijo na razširljivost in nista odvisni od določene implementacije.

Na tem mestu velja še enkrat poudariti, da uporaba podatkovne baze ZODB in komponentne arhitekture ZCA ni vezana na platformo Zope, ampak ju lahko uporabljamo v poljubnem okolju, ki uporablja programski jezik Pyhton.

Največ težav pri razvoju aplikacije sem imel s pomanjkanjem uporabniške dokumentacije za posamezne komponente. Veliko razvijalcev se zadovolji že z izdelavo dokumentacijskih testov. Ti pogosto vsebujejo testiranje funkcionalnosti, ki za končnega uporabnika niso pomembne in otežujejo razumevanje delovanja komponente. Tudi sama skupnost, ki sodeluje

pri projektu Grok, je majhna v primerjavi s skupnostmi priljubljenejših ogrodij. Na spletu je na voljo malo virov, povezanih z ogrodjem, komunikacija pa poteka večinoma samo preko poštnih seznamov.

Prostora za izboljšavo prototipa aplikacije je še veliko. Zelo koristno bi bilo omogočiti pošiljanje obvestil preko sporočil SMS, komunikacija pa bi potekala tudi v obratni smeri, npr. za sporočanje udeležbe za določen dogodek. Aplikacija bi bila tako precej bolj uporabna, saj usklajevanje terminov pogosto poteka v zadnjem trenutku. Implementirali bi lahko tudi komponente za vodenje evidence dogodkov in plačil ter grafični prikaz rezultatov dogodkov. Morda največja pridobitev pa bi bila vpeljava sistema za socialno mreženje, ki bi omogočal komunikacijo in organizacijo dogodkov med različnimi ekipami ter iskanje novih članov.

# Seznam slik

*Slika 1:* Administrativni vmesnik ZMI (Zope Managemenent Interface) platforme Zope 2 .... 8
Slika 2: Primer dokumentacijskega testa iz knjižnice zope.formlib ........ 12
Slika 3: Ogrodja, ki uporabljajo Zope Toolkit Library (ZTK) ........ 13
Slika 4: ZCA kot register za iskanje storitev ........ 14
Slika 5: Prikaz načrtovalskega vzorca »Pretvornik« ........ 19
Slika 6: Struktura podatkovne baze ZODB ........ 29
Slika 7: Shranjevanje podatkov z uporabo datotečnega sistema FileStorage ........ 32
Slika 8: Shranjevanje podatkov s pomočjo strežnika ZEO in izenačevanja bremena ........ 33
Slika 9: Primerjava omrežne topologije med sistemoma ZEO ter NEO ........ 34
Slika 10: Diagram primerov uporabe za akterja »Obiskovalec ........ 38
Slika 11: Diagram primerov uporabe za akterja »Uporabnik« ........ 41
Slika 12: Diagram primerov uporabe za člana ekipe ........ 45
Slika 13: Diagram primerov uporabe za administratorja dogodkov ........ 52
Slika 14: Diagram primerov uporabe za administratorja ekipe ........ 57
Slika 15: Diagram primerov uporabe za akterja »Sistemska ura« ........ 63
Slika 16: Potek dogodkov pri pasivnem vzorcu MVC ........ 66
Slika 17: Diagram zaporedja za primer uporabe »Registracija« ........ 69
Slika 18: Diagram zaporedja za primer uporabe "Vnos ekipe" ........ 70
Slika 19: Diagram zaporedja za primer uporabe "Dodajanje novih članov" ........ 71
Slika 20: Diagram zaporedja za primer uporabe Vnos komentarjev ........ 71
Slika 21: Diagram zaporedja – Obveščanje pred dogodkom ........ 72
Slika 22: Osnovna tipa entitet pri ogrodju Grok ........ 73
Slika 23: Podatkovni model aplikacije ........ 74
Slika 24 - Diagram paketov ........ 75
Slika 25 – Diagram paketov za celotno aplikacijo ........ 76
Slika 26: Administrativni vmesnik Grok ........ 79
Slika 27: Osnovna struktura strani. ........ 84
Slika 28: Izgled strani za primer uporabe »Prijava« ........ 92
Slika 29: Izgled strani za primer uporabe »Pregled dogodka in urejanje udeležbe« ........ 93
Slika 30: Izgled strani za primer uporabe »Dodajanje dogodka« ........ 94
Slika 31: Izgled strani za primer uporabe »Pregled dogodkov« ........ 94
Slika 32: Izgled strani za primer uporabe »Urejanje ekipe« ........ 95

# Seznam tabel

Tabela 1: Seznam pomembnejših dogodkov ........ 27
Tabela 2: Seznam podatkovnih entitet aplikacije ........ 80
Tabela 3: Komponente plasti poslovne logike ........ 82

# Seznam primerov

Primer 1: Definiranje vmesnika s pomočjo knjižnice zope.interface ..................................... 17
Primer 2: Definicija atributov v vmesniku z uporabo knjižnice zope.schema ........................ 17
Primer 3: Implementacija vmesnika ........................................................................................ 18
Primer 4: Prikaz uporabe metod implementedBy ter providedBy ........................................... 18
Primer 5: Realizacija pretvornika za izbiro unikatnega imena v vsebniku ............................. 21
Primer 6: Primer registracije pretvornika z uporabo jezika ZCML ........................................ 22
Primer 7: Implementacija in registracija pretvornika z uporabo ogrodja Grok ....................... 23
Primer 8: Definicija in registracija pretvornika za izbiro unikatnega imena novice ............... 24
Primer 9: Primer uporabe tovarne za kreiranje novega uporabnika ........................................ 26
Primer 10: Primer uporabe dogodkov ....................................................................................... 27
Primer 11: Sporočanje trajnostnemu mehanizmu, da je prišlo do spremembe ........................ 30
Primer 12: Definicija trajnostnega razreda .............................................................................. 31
Primer 13: Prikaz operacij CRUD pri bazi ZODB .................................................................. 31
Primer 14: Prikaz uporabe vnosnih form pri ogrodju Grok ..................................................... 81
Primer 15: Definiranje indeksov za dogodke ........................................................................... 82
Primer 16: Uporaba kataloga za iskanje dogodkov .................................................................. 82
Primer 17:Glavna predloga z osnovno postavitvijo ................................................................. 85
Primer 18: Implementacija prikazne forme za ekipo ............................................................... 85
Primer 19: Privzeta predloga prikazne forme .......................................................................... 86
Primer 20: Definiranje navigacijskega menija .......................................................................... 87
Primer 21: Predloga za prikaz navigacije ................................................................................. 87
Primer 22: Primer definicije elementa navigacijskega menija ................................................. 87
Primer 23: Primer definicije pravic .......................................................................................... 88
Primer 24: Primer združevanja pravic v vloge ......................................................................... 89
Primer 25: Zaščita pogledov z ukazom grok.requires() ........................................................... 89
Primer 26: Prirejanje vloge »Adminitrator  ekipe« uporabniku za določeno ekipo ................ 89
Primer 27: Prirejanje vloge »Uporabnik« novemu uporabniku  za celotno aplikacijo ............ 89
Primer 28: Implementacija naročnika, ki poskrbi za dodajanje uporabnika v sistem po potrditvi registracije ................................................................................................................ 90
Primer 29: Definicija ponudnika vsebine, ki prikaže komentarje ............................................ 91

# Seznam uporabljenih virov


[1] *The Web Site for the Zope Community*. 2010; Dostopno na: http://en.wikipedia.org/wiki/Zope/, http://www.zope.org/.

[2] *Software framework*. 2010; Dostopno na: http://en.wikipedia.org/wiki/Software_framework.

[3] M. Fowler. *Inversion of control*. 2005; Dostopno na: http://martinfowler.com/bliki/InversionOfControl.html.

[4] I. Ricotti. *Introduction to Dependency Injection and Aspect Oriented Programming*. 2008.

[5] *Web application framework*. 2010; Dostopno na: http://en.wikipedia.org/wiki/Web_application_framework.

[6] M. Lutz. *Programming Python*. 2006; Dostopno na: http://books.google.com/books?id=5zYVUIl7F0QC&lpg=PA1130&ots=HUb87eoQwI&pg=PA1130#v=onepage&q&f=false.

[7] P.v. Weitershausen. *Web Component Development with Zope 3*. 2008.

[8] *Common criteria*. 2010; Dostopno na: http://en.wikipedia.org/wiki/Common_Criteria.

[9] *Comparing Open Source Content Management Systems: WordPress, Joomla, Drupal and Plone*. 2009; Dostopno na: http://idealware.org/reports/comparing-open-source-content-management-systems-wordpress-joomla-drupal-and-plone.

[10] M. Baiju, *A Comprehensive Guide to Zope Component Architecture*. 2009.

[11] C.d.l. Guardia, *Grok 1.0 Web Development*. 2010.

[12] *Flavours of Zope* 2010; Dostopno na: http://ozzope.org/what-is-zope.

[13] *Component-based software engineering*. 2010; Dostopno na: http://en.wikipedia.org/wiki/Component-based_software_engineering.

[14] *BlueBream's documentation*. Zope concepts 2010; Dostopno na: http://bluebream.zope.org/doc/1.0/concepts.html.

[15] K. Johnson. *Interfaces in Python*. 2006; Dostopno na: http://mail.python.org/pipermail/tutor/2006-June/047513.html.

[16] *Glossary*. 2010; Dostopno na: http://docs.python.org/glossary.html#term-duck-typing.

[17] T. Guido van Rossum. *Introducing Abstract Base Classes*. ABC vs. Duck Typing 2009; Dostopno na: http://www.python.org/dev/peps/pep-3119/#abcs-vs-duck-typing.

[18] T. Guido van Rossum. *Introducing Abstract Base Classes*. 2009; Dostopno na: http://www.python.org/dev/peps/pep-3119/.

[19] C.d.l. Guardia, *Grok 1.0 Web Development*.

[20] *Adapter pattern*. 2010; Dostopno na: http://en.wikipedia.org/wiki/Adapter_pattern.

[21] *Adapter pattern*. Dostopno na: http://www.oodesign.com/adapter-pattern.html.

[22] R.H. Erich Gamma, Ralph Johnson, John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

[23] M. Aspeli. *Adapters*. 2009; Dostopno na: http://plone.org/products/dexterity/documentation/manual/five.grok/core-components/adapters.

[24] *A whirlwind tour of Zope 3*. 2009; Dostopno na: http://plone.org/documentation/manual/developer-manual/archetypes/appendix-practicals/b-org-creating-content-types-the-plone-2.5-way/a-whirlwind-tour-of-zope-3.

[25] *Core components* 2010; Dostopno na: http://plone.org/products/dexterity/documentation/manual/five.grok/core-components/referencemanual-all-pages.
[26] *Component architecture* 2010; Dostopno na: http://plone.org/documentation/manual/plone-community-developer-documentation/components/referencemanual-all-pages.
[27] S. Richter, *The Zope 3 Developers Book.*
[28] *NoSql*. 2010; Dostopno na: http://nosql-database.org/.
[29] *Object-relational impendance mismatch.* 2010.
[30] T. Neward. *The Vietnam of Computer Science*. 2006; Dostopno na: http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx.
[31] *ACID*. 2010; Dostopno na: http://en.wikipedia.org/wiki/ACID.
[32] *NEOPPOD Distributed Transactional NoSQL Object Database*. 2010; Dostopno na: www.neopodd.org.
[33] *Object database*. 2010; Dostopno na: http://en.wikipedia.org/wiki/Object_database.
[34] *ZODB vs Relational Database: a simple benchmark*. 2007; Dostopno na: http://pyinsci.blogspot.com/2007/09/zodb-vs-relational-database-simple.html.
[35] P. Winkler. *To ZODB or Not to ZODB*. 2009; Dostopno na: http://www.coactivate.org/projects/topp-engineering/blog/2009/03/20/to-zodb-or-not-to-zodb.
[36] *Model–View–Controller*. 2010; Dostopno na: http://en.wikipedia.org/wiki/Model%E2%80%93View%E2%80%93Controller.
[37] *Enotna metodologija razvoja informacijskih sistemov*. 2000.