UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Tine Lesjak

Razvoj naprednih storitev za GIS

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentorica: doc. dr. Mojca Ciglarič

Ljubljana, 2009

original izdane teme

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Tine Lesjak,

z vpisno številko 63030315,

sem avtor diplomskega dela z naslovom:

Razvoj naprednih storitev za GIS

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom

  doc. dr. Mojce Ciglarič
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 13. oktobra 2009 Podpis avtorja:

Zahvala

Zahvaljujem se mentorici doc. dr. Mojci Ciglarič, ki mi je omogočila kakovostno izvesti to delo.

Posebno se zahvaljujem bratrancu Borutu Lesjaku za pomoč in izredno motivacijo, saj sva skupaj naredila večidel izpitov.

Podjetju Autronic, d.o.o., iz Ljubljane, še posebej direktorju Saši Albertu, se iskreno zahvaljujem za vso podporo pri sodelovanju na zanimivih razvojnih projektih skoraj ves čas študija.

Za ideje, algoritme in rešitve se zahvaljujem sošolcu in sodelavcu Ivanu Fućaku.

Zadnja, a najbolj pomembna zahvala gre staršema, saj sta edina, ki sta vse skupaj omogočila. Hvala vama!

Kazalo

Seznam uporabljenih kratic in simbolov

API	Application Programming Interface - aplikacijski programski vmesnik
BBOX	Bounding Box - omejujoč pravokotnik
DAO	Database Access Object - objekt s pripravljenimi metodami za dostop do podatkovne zbirke
GIS	Geographic Information System - geografski informacijski sistem
GiST	Generalized Search Tree - posebna vrsta indeksa, ki se uporablja za prostorske podatke
GPS	Global Positioning System - sistem za svetovno določanje položaja
HQL	Hibernate Query Language - povpraševalni jezik orodja Hibernate za podatkovne zbirke
HTTP	Hypertext Transfer Protocol - transportni protokol za hiperbesedilo
IDE	Integrated Development Environment - razvojno okolje
JAR	Java Archive - arhiv kode v javi
JDBC	Java Database Connectivity - API za dostop do podatkovne zbirke v javi
JPA	Java Persistence API - API za opis podatkov za podatkovne zbirke v javi
JVM	Java Virtual Machine - navidezni stroj za pogon javine kode
LBS	Location-Based Service - lokacijsko odvisna storitev
OGC	Open Geospatial Consortium - konzorcij za standardizacijo upravljanja geografskih podatkov
ORM	Object-Relational Mapping - preslikovanje med objekti in relacijami
POI	Point of Interest - interesantna točka
POJO	Plain Old Java Object - vrsta javinega objekta
SQL	Structured Query Language - povpraševalni jezik za delo s podatkovnimi zbirkami
SRID	Spatial Reference Identifier - oznaka prostorskega referenčnega koordinatnega sistema

SSL	Secure Sockets Layer - kriptografski protokol za zaščito podatkov pri omrežni komunikaciji
STS	SpringSource Tool Suite - ime konkretnega razvojnega okolja
TLS	Transport Layer Security - kriptografski protokol za zaščito podatkov pri omrežni komunikaciji, naslednik SSL
WAR	Web Application Archive - arhiv spletne aplikacije v javi
WGS	World Geodetic System - standardni geodetski referenčni sistem
XML	Extensible Markup Language - standardni označevalni jezik
XSD	XML Schema Definition - standardni jezik za sheme XML

1 Povzetek

V zadnjih letih smo doživeli preporod na področju geografskih informacij. Navigacijske naprave so prodrle v vsakdanje življenje. Prav tako množica brezplačnih spletnih zemljevidov, ki ponujajo prvovrstne storitve. Zato jih uporabniki kar nekako pričakujemo in celo zahtevamo. Do podobnega sklepa smo prišli v podjetju, zato sem razvil tri napredne storitve za geografski informacijski sistem, ki so trenutno zelo priljubljene.

Storitev geokodiranja poišče naslove, ceste ali zgradbe, ki ustrezajo iskalnemu nizu. Uporabnik izbere iskalca in nastavi njegove lastnosti, med drugim ime kraja, ki ga išče. Pri iskanju sem se dotaknil tudi celobesedilnega iskanja, ki ga zaradi več razlogov nisem uporabil.

Storitev nasprotnega geokodiranja poišče najbližje naslove, ceste ali zgradbe, ki se nahajajo v neposredni bližini določene lokacije. Uporabnik izbere iskalca, nastavi koordinate lokacije in delovni radij, v katerem se išče. Pri tem je zelo pomembna metoda računanja najkrajših razdalj. Najbolje se je izkazalo neposredno računanje razdalj na modelu sfere, ki je podobna površini Zemlje. V delu so izčrpno preizkušene tudi druge metode.

Storitev usmerjanja izračuna najcenejšo pot iz ene lokacije v drugo. Uporabnik določi začetno in končno lokacijo ter funkciji ocenjevanja poti. Uporabljen algoritem je A*, ki pospeši računanje z uporabo hevrističnega predvidevanja. Ena funkcija ocenjuje že obdelano pot, druga hevristično ocenjuje preostalo pot do cilja.

Vse tri storitve so zgrajene v obliki strežniških storitev. Uporabljajo isti, na novo razvit podatkovni model, v katerem se podatki ne podvajajo. Model lahko hrani podatke z vsega sveta in je dovolj fleksibilen za morebitne razširitve.

Podatki cest, naslovov in zgradb so hranjeni v relacijski podatkovni zbirki. Lahko imajo več geografskih imen. Med zbirko in strežnikom se pretvarjajo s pomočjo objektno relacijske preslikave. Ves sistem temelji na javanski tehnologiji.

Sistem je delujoč, a nikakor ne dokončen, saj je možnosti za izboljšave še ogromno.

Ključne besede:

geografski informacijski sistem, geokodiranje, nasprotno geokodiranje, usmerjanje, podatkovni model

2 Abstract

In the recent years we have experienced a renaissance in the field of geographic information. Navigation devices have penetrated into the everyday life. There is also an abundance of free web maps bundled with first-class services. Therefore, we expect them and even demand them. We came to a similar conclusion in our company, so I implemented three advanced services for the geographic information system that are very popular at the moment.

The geocoding service finds addresses, streets or buildings that match a given search string. User chooses a locator and sets its properties, including a name of the place they are looking for. I also considered the full text searching, but which I didn't use for several reasons.

The reverse geocoding service finds the nearest addresses, streets or buildings located in the nearby vicinity of a specified location. User chooses a locator, sets location coordinates and the working radius to be searched in. A method for computing the shortest distances is very important here. The best such method has proven to be a computation of distances on an Earth-shaped spherical model. The thesis contains other extensively tested methods, too.

The routing service calculates the least expensive route from one location to another. User specifies a start and a goal location and two path evaluation functions. The A* algorithm is used, which speeds up the computation by using heuristic assumptions. One function estimates the path already handled; the second heuristically estimates the remaining path to the goal.

All three services are built in the form of server services. They all use the same, newly developed, data model, in which the data is not duplicated. The model stores data universally for any location in the world and is flexible enough for any future expansions.

Streets', buildings' and addresses' data is stored in a relational database and can use multiple geographical names. Object-relational mapping is used for communicating the data between the database and the server. The whole system is based on the Java technology.

The result of this thesis is a fully-operational system, although it goes without saying, that the potential for improvements is enormous.

Key words:

geographic information system, geocoding, reverse geocoding, routing, data model

3 Uvod

V svojem diplomskem delu sem se lotil geografskega informacijskega sistema (GIS).

GIS je obširen računalniško voden informacijski sistem za zajemanje, hranjenje, urejanje, analiziranje, prikazovanje in deljenje prostorskih podatkov. Znotraj področja analiziranja podatkov med drugim spadajo tudi orodja, ki sem jih razvil za podjetje in jih bom predstavil v tem delu.

V podjetju Autronic, d.o.o., iz Ljubljane, pri katerem sem delal, smo razvijali sistem sledenja objektov v realnem času, kjer je prikaz lokacij na karti ključnega pomena. Prišli smo do točke, ko smo v sistem želeli vključiti več naprednih orodij, ki so si jih zaželele stranke. Ta prinašajo tudi konkurenčno prednost. To so geokodiranje, nasprotno geokodiranje in usmerjanje. Poleg samih orodij smo nujno potrebovali tudi nov podatkovni model, v katerem lahko hranimo obstoječe geografske podatke.

Geokodiranje pride prav uporabnikom, ki želijo izvedeti, kje se nahaja določen geografski kraj (npr. naslov, cesta, zgradba ipd.), pri tem pa poznajo samo ime kraja. Geokodiranje se uporablja s t.i. *iskalcem*, ki ga uporabnik natančno nastavi, tudi po kateri vrsti krajev naj išče.

Za naprednejše iskanje bi lahko uporabili celobesedilno iskanje, ki sem ga tudi preizkusil.

Nasprotno geokodiranje določeni lokaciji poišče najbližje geografske kraje. Primer uporabe: lastniku vozila se namesto koordinat v odjemalcu prikaže najbližji naslov, kjer stoji vozilo. Enako kot geokodiranje se tudi nasprotno geokodiranje uporablja z *iskalci*, ki natančno definirajo iskanje. Nasprotno geokodiranje išče najbližje kraje na podlagi najkrajših razdalj med dvema lokacijama na modelu sfere, ki predstavlja površino Zemlje. Iskati je možno naslove, ceste in zgradbe.

Usmerjanje uporabniku izračuna najcenejšo pot iz začetne lokacije v ciljno. Najcenejša pot je lahko najkrajša ali pa najhitrejša. Uporablja se hevristični usmerjevalni algoritem A*, ki mu uporabnik lahko nastavlja bistvena parametra: funkcijo ocenjevanje že obdelane poti in funkcijo hevrističnega ocenjevanja poti do cilja.

Ključno pri vseh orodjih je, da so zgrajena v obliki strežniških **storitev**, ki uporabljajo le en

podatkovni model. Podatki se ne podvajajo in so dostopni na enem mestu. To bistveno pripomore tako k vzdrževanju podatkovne zbirke kot samih podatkov.

V podatkovni model lahko vstavimo podatke z vsega sveta in jih med seboj tudi analiziramo. Zaželeno je, da je dovolj fleksibilen, da ga lahko kasneje enostavno razširimo. Podatkovni model definira domeno in vnaprej pripravljene dostope do podatkov v zbirki.

Pri razvoju podatkovnega modela in storitev sem upošteval nekaj dejstev, ki veljajo za podjetje. Obstoječ grafični odjemalec je namizna aplikacija, spisana v tehnologiji Java. Podjetje ima lastne geografske podatke, ki so jih kupili ali izdelali sami, zato pridejo v poštev samo poceni ali zastonjske rešitve, ki so hkrati dovolj fleksibilne za prilagajanje in preproste za vzdrževanje in uporabo. Sistem je zato prilagojen za namiznega odjemalca v javi, uporablja lastne podatke, je fleksibilen za morebitne razširitve in je čim bolj preprost.

Veliki GIS-i so domena le nekaterih velikih podjetij in državnih agencij. Slednje so načeloma zaprte vase. Zelo razširjeni spletni zemljevidi sicer omogočajo geokodiranje in usmerjanje, a so prilagojeni le za splet. Namizne in strežniške aplikacije so večinoma velike in okorne, cene so zelo visoke.

Sistem sem napisal v programskem jeziku Java, ki ga obvladam, je objektno usmerjen, najbolj priljubljen in zaželen s strani podjetja. Uporabljena je običajna relacijska podatkovna zbirka PostgreSQL skupaj z dodatkom za geografske podatke PostGIS, ki je ekonomična in preverjena izbira. Za poenostavitev je zelo primerna tehnologija preslikave relacijskih podatkov v objekte - na tem področju kraljuje knjižnica Hibernate, ki svoje naredi tudi s posebnim objektno usmerjenim poizvedovalnim jezikom HQL. Vse aplikacijske dele je bilo najlažje med seboj povezati z ogrodjem Spring Framework. Ta vsebuje tudi množico orodij, ki olajšajo razvoj, med drugim tudi tehnologijo HTTP Invoker, s katero je izvedena povezava med strežnikom in odjemalcem. Vse tehnologije so na voljo brezplačno.

Celoten sistem je sestavljen modularno iz treh končnih paketov; to so strežniški deli (storitve) v obliki spletnih aplikacij. Te za delovanje potrebujejo vsebnik. Vsi paketi poleg definicij storitev za odjemalce za delovanje potrebujejo še lastne in zunanje knjižnice.

Sistem je delujoč, a za uporabno delovanje potrebuje lastne geografske podatke, ki mu niso priloženi.

V naslednjem poglavju si bomo podrobneje pogledali, kakšni so bili problemi, nameni, razlogi in cilji sistema. Dotaknili se bomo tudi možnih alternativ.

V poglavju Načrt sistema si bomo ogledali, kaj vse sistem zaokrožuje, vsebuje in na kakšnih tehnologijah so zgrajeni njegovi temelji.

Sledijo poglavja o glavnih delih sistema: v prvem poglavju bo beseda tekla o zgradbi in uporabi podatkovnega modela, v ostalih poglavjih pa si bomo podrobneje ogledali vse tri storitve (geokodiranje, nasprotno geokodiranje in usmerjanje), predvsem algoritme, ter njihovo zmogljivost.

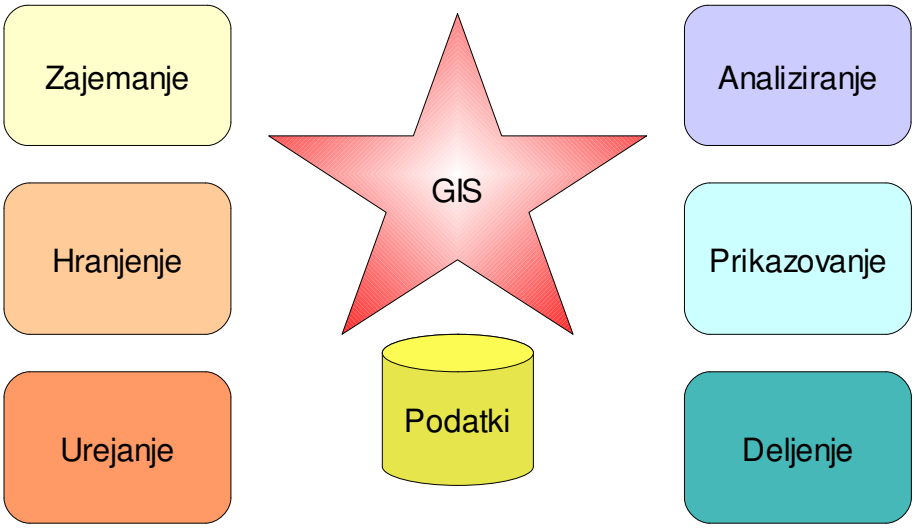
V sklepnih poglavjih so zbrane izkušnje, prednosti, slabosti in predlogi za nadaljnje delo.

V dodatku A so osnovna navodila za uporabo sistema, predvsem iz razvojnega vidika.

Izvorna koda se nahaja na priloženi CD plošči.

4 Opis problema

GIS je računalniško voden informacijski sistem za zajemanje, hranjenje, urejanje, analiziranje (med drugim tudi iskanje), prikazovanje in deljenje podatkov, ki predstavljajo lokacije, - podatki so geografski [3, 7]. V GIS spada množica orodij, ki so za to potrebna ali koristna. Ta množica je zelo velika. Na trgu je ogromno brezplačnih in komercialnih orodij, ki svoje delo opravljajo dobro.



Slika 4.1: Dejavnosti GIS-a okoli podatkov.

GIS je lahko zelo koristen za podjetje, ki s svojo dejavnostjo sega tudi na geografsko področje. Eno takšnih je Autronic, d.o.o., iz Ljubljane, pri katerem sem skoraj ves čas študija imel opravka z geografskimi podatki. Razvijali smo sistem sledenja objektov v realnem času, kjer je prikaz lokacij na karti ključnega pomena.

V podjetju smo prišli do točke, ko smo želeli sistem začiniti tudi z več naprednimi orodji. Za večino orodij so stranke izrazile željo, nekatere tudi zahtevo. V vsakem primeru pa prinašajo **konkurenčno prednost**. Tri takšna orodja predstavljam v tem delu.

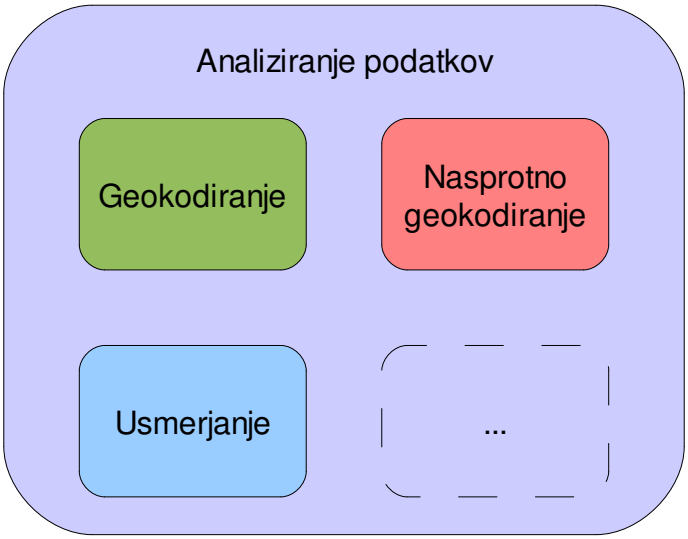
4.1 Funkcionalne zahteve

Prvo orodje je s strani strank zelo zaželeno. Prav pride uporabnikom, ki želijo izvedeti, kje se nahaja določen geografski kraj (npr. naslov, cesta, zgradba ipd.). V ustrezno opremljenem odjemalcu vnesejo iskalni niz, orodje jim poišče ustrezne zadetke, grafični odjemalec pa jih prikaže na zemljevidu. Orodje se imenuje **geokodiranje**. Ključno pri tem orodju je, da je zgrajen v obliki strežniške storitve, ki uporablja le en podatkovni model za vsa orodja GIS. O prednostih takšne zasnove malce kasneje.

Drugo orodje smo do delujočega stanja razvili že čisto na začetku, saj brez njega strankam ne bi imeli veliko za ponuditi. Orodje okoli podanih koordinat lokacije poišče najbližje geografske kraje. Imenuje se **nasprotno geokodiranje**. Uporabniška izkušnja je naslednja: lastniku vozila se namesto težko predstavljivih geografskih koordinat v grafičnem odjemalcu prikaže najbližji naslov, kjer stoji njegovo vozilo.

Prva implementacija je bila vgrajena v sam odjemalec, nestandardna in nepovezana z drugimi orodji. Namen nove implementacije je bil, da bi uporabljala iste podatke kot nasprotno geokodiranje (isti podatkovni model) in da bi bila zgrajena v obliki strežniške storitve.

Tretje orodje se imenuje **usmerjanje**. Uporabniku izračuna najcenejšo pot iz ene lokacije v drugo. Kaj za uporabnika pomeni najcenejša pot, lahko sam izbere, načeloma pa orodje podpira izračun najkrajše ali najhitrejše poti. V podjetju usmerjanja nismo imeli namena razviti že v tej fazi, a je sodelavec in sošolec Ivan Fućak s svojim diplomskim delom *Usmerjanje v Geografskem informacijskem sistemu* [3] zelo olajšal nadaljnje delo. Orodje, tako kot oba druga, uporablja isti podatkovni model in je zasnovano v obliki strežniške storitve.



Slika 4.2: Orodja za analiziranje geografskih podatkov.

Prednosti uporabe enega **podatkovnega modela** za vsa orodja GIS so očitne. Geografski podatki se ne podvajajo in vsi so dostopni na enem mestu. To bistveno pripomore tako k vzdrževanju podatkovne zbirke kot samih podatkov. Če je podatkovni model tudi dobro narejen, je prednosti še več. Ena od ključnih lastnosti je tudi ta, da ni omejen samo na lokalne

podatke. Vanj lahko vstavimo podatke z vsega sveta, kot takšne pa jih lahko med seboj tudi analiziramo, ne glede na to, kje se nahajajo. Zaželena je tudi določena stopnja fleksibilnosti, saj je GIS zelo živahno področje, ki zahteva nenehne izboljšave, tudi nova orodja. Tako je morebitne razširitve modela kasneje enostavneje dodati.

Prednosti zasnove orodja v obliki strežniške storitve skupaj z enim podatkovnim modelom so naslednje:

- strežnik je enolično dosegljiv vsem odjemalcem: možno je posodobiti, popraviti, zamenjati ali premestiti strežnik, ne da bi odjemalci vedeli za to,
- strežnik je načeloma lažje optimizirati za določeno strojno opremo in ponuja večji nadzor nad sredstvi in dostopom do podatkov,
- tehnologije strežnik-odjemalec so že dolgo časa na voljo in so zelo zrele glede enostavnosti uporabe in varnosti,
- na strežnik se lahko priključijo različni odjemalci in
- podatki so centralizirani, kar olajša njihovo vzdrževanje in zmanjšuje njihovo podvajanje.

Pri razvoju podatkovnega modela in storitev sem upošteval nekaj dejstev, ki veljajo za podjetje. Podjetje že ima **obstoječega grafičnega odjemalca**, in sicer gre za namizno aplikacijo, spisano v tehnologiji Java. To pomeni, da je bilo zelo zaželeno, da je storitev temu primerno prilagojena. Podjetje ima **lastne geografske podatke**, ki so jih kupili ali izdelali sami. V poštev pridejo samo poceni ali zastonjske rešitve, ki so hkrati dovolj fleksibilne za prilagajanje in preproste za vzdrževanje in uporabo.

Pri pregledu obstoječih rešitev razmeroma hitro ugotovimo, da ni veliko ponudnikov in orodij, ki bi vsaj približno ustrezali tem zahtevam.

4.2 Pregled obstoječih rešitev

Veliki GIS-i so domena le nekaterih velikih podjetij in državnih agencij. Slednje sicer javnosti ponujajo določene podatke, a njihov sistem je večinoma lastno razvit in **zaprt**. Navadnim spletnim uporabnikom je prvi na veliko odprl oči Google, ki je s svojim spletnimi zemljevidi in aplikacijo Google Zemlja prikazal svet v natančni interaktivni podobi. Kmalu so mu sledila druga velika podjetja (npr. Microsoft in Yahoo). Zemljevidi postajajo vsak dan bolj natančni, kar se gre zahvaliti vedno večjemu številu satelitov in podjetjem, ki so specializirana za ponujanje digitalnih geografskih podatkov za cel svet (npr. Tele Atlas in Navteq). Mimogrede, cene teh podatkov so za majhna podjetja astronomsko visoke.

Politika in pogoji uporabe njihovih storitev, ki sedaj vključujejo tudi geokodiranje, nasprotno geokodiranje in usmerjanje, so **prilagojeni za splet**. Na primer, Googleove zemljevide je možno uporabljati samo v spletnem brskalniku, vključiti jih smemo samo v javno dostopno spletno stran [17]. Podobno velja tudi pri drugih ponudnikih. To pa v podjetju prinaša težavo, saj jih ni mogoče uporabiti v zaprtem namiznem javinem odjemalcu. Alternativo ponuja Microsoft, in sicer prek standardne spletne storitve [13], vendar proti visokem plačilu.

Po drugi strani pa je podjetij, ki bi prodajala celovite GIS-e kot programsko opremo, **zelo malo**. Njihove aplikacije so večinoma **velike**, **okorne** in **zelo drage**. Da ne bo pomote, govorimo o GIS-ih, ki ponujajo vsaj ta tri orodja (geokodiranje, nasprotno geokodiranje in

usmerjanje) z možnostjo vključitve lastnih podatkov. Ponudba ostalih orodij, kot so orodja za vstavljanje, urejanje in prikazovanje geografskih podatkov je velikanska. Najdemo množico od odprtokodnih do plačljivih [14].

Še največji ponudnik programske opreme GIS, ki vključuje ta tri orodja, je podjetje ESRI, ki prodaja strežnik ArcGIS Server. Tega je možno kupiti v različici, ki vključuje zelo napredna orodja, a govorijo o ceni nekaj deset tisoč ameriških dolarjev [2]. Podobno je verjetno tudi z drugima podjetjema, ki precej skrivata svoje funkcionalnosti: Intergraph in GeoConcept.

Od programskih kosov, ki bi pripomogli k izdelavi GIS-a, moram omeniti odprto okolje GRASS, ki na enoten način omogoča urejanje najrazličnejših geografskih podatkov in njihovo analiziranje, med drugim vsebuje tudi orodja za izdelavo želenih storitev. Gre samo za razvojno okolje, ki bi se ga teoretično dalo uporabiti v strežniku.

Zanimiv je strežnik Gisgraphy, ki so ga razvijalci zelo približali zasnovi tega sistema, saj so uporabili praktično enake tehnologije, a niso razvili usmerjanja. Usmerili so se predvsem na uvoz podatkov in uporabo prosto dostopnih podatkov. Njihov podatkovni model se mi zdi vse kaj drugega kot preprost.

Sistem	Za	Proti
Spletni zemljevidi in storitve različnih ponudnikov (Google Maps, Microsoft Bing Maps, Yahoo! Maps idr.)	zelo natančni podatki za cel svet, zelo dobro dokumentiran programski vmesnik API, vedno posodobljeni podatki	ni možno uporabiti v zaprtih namiznih odjemalcih v javi zaradi pogojev uporabe in tehnologije ajax (večina), ni možno vključiti lastnih podatkov (razen nekateri proti plačilu), omejitev uporabe (nekateri)
ESRI ArcGIS Server Standard	zelo napredne in premišljene storitve	visoka cena
GRASS	odprtokoden, 25 letna zrelost	ni strežnik, ni podatkovnega modela
Gisgraphy	odprtokoden, uporaba znanih tehnologij	ni usmerjanja, podatkovni model ni preprost

Tabela 4.1: Dobre in slabe lastnosti obstoječih sistemov.

Po pregledu obstoječih alternativ je jasno, da je izdelava lastnega sistema s ključnimi storitvami geokodiranja, nasprotnega geokodiranja in usmerjanja najcenejša, saj je prilagojena določenim zahtevam in na koncu koncev prinaša obilo veselja pri razvoju ter to diplomsko delo.

Sistem ima naslednje lastnosti:

- je prilagojen za (namiznega) odjemalca v javi,
- uporablja lastne podatke,
- uporabljene tehnologije so zastonj,
- je fleksibilen za morebitne razširitve in
- je čim bolj preprost.

Zanimanje za prostorske informacije je vedno večje. GIS-i vztrajno zavijajo v vode tako imenovanih **lokacijsko odvisnih storitev** (angl. Location-Based Services - LBS) [4, 7]. Te nudijo uporabnikom podatke, ki ustrezajo njihovi trenutni okolici. Ker so v uporabi novi načini komuniciranja, kot so mobilna/brezžična omrežja, majhne prenosne naprave (mobilni telefoni), od LBS-jev pričakujemo nekaj več kot od klasičnih GIS-ov: visoko zmogljivost (veliko mobilnih uporabnikov), zanesljivost (neprekinjeno delovanje), realno časovno odzivnost in dinamično prilagodljivost (lokacija uporabnika), mobilnost in odprtost (podporo splošno sprejetih standardov) ter varnost. LBS ponujajo mobilni operaterji (npr. Mobitelov Lokus [20]), vse več pa je tudi namenskih aplikacij (npr. storitev Latitude pri Google Maps za mobilne naprave [16]).

Konzorcij OGC (Open Geospatial Consortium) je pripravil odprt standard OpenLS [22], ki definira vmesnike za lažjo izmenjavo podatkov med množico mobilnih naprav in storitvami LBS, med drugim tudi za storitev geokodiranja, nasprotnega geokodiranja in usmerjanja. Ta sistem bi lahko prilagodili temu standardu.

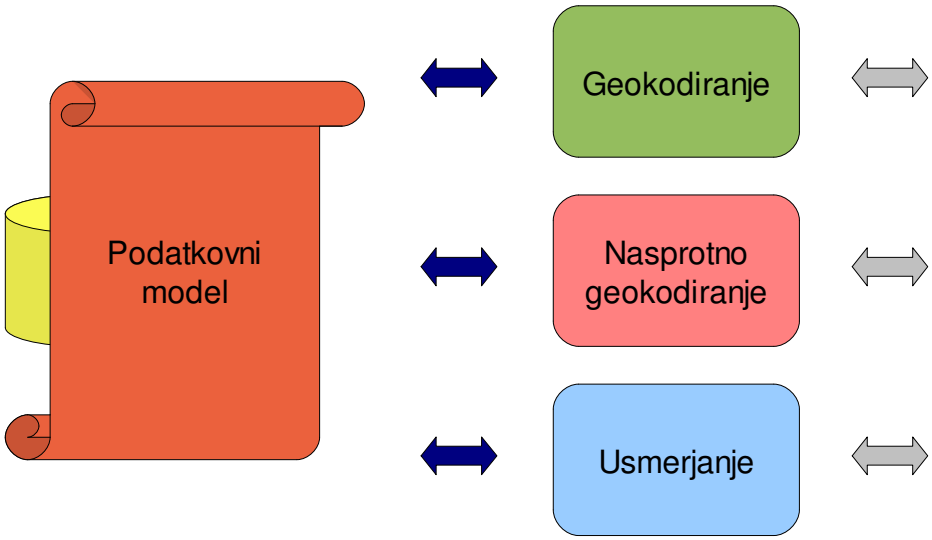
5 Načrt sistema

V tem poglavju si bomo ogledali, kaj vse sistem zaokrožuje, vsebuje in na kakšnih tehnologijah so zgrajeni njegovi temelji. Predvsem kratek uvod v posamezno uporabljeno tehnologijo je pomemben, če želimo razumeti algoritme posamezne storitve in če morebiti želimo sistem vključiti v svojega lastnega.

5.1 Sestava

Sistem je sestavljen iz štirih glavnih komponent:

- podatkovnega modela,
- storitve geokodiranja,
- storitve nasprotnega geokodiranja in
- storitve usmerjanja.



Slika 5.1: Sestava sistema.

Vse storitve so prilagojene za podatkovni model in so zato od njega tudi odvisne. Vsako

komponento sem posebej podrobno opisal v naslednjih poglavjih.

Kljub temu, da sistem združuje več komponent, je še vedno le del celotne uporabniške izkušnje. To bi lahko razdelili na vsaj tri dele. Prvi je odjemalec za prikaz podatkov, s katerim uporabnik krmili in opazuje ves sistem. Ta je neposredno odvisen od storitev (drugi del), ki jih nudi strežnik. Strežnik uporabniške ukaze izvaja nad podatki, ki predstavljajo tretji del.

Sistem povsem zaokrožuje drugi del uporabniške izkušnje, do celote mu manjkajo podatki in implementacija na strani odjemalca. Oboje sem za uspešno testiranje in opravljanje meritev zmogljivosti potreboval.

Dodatno sem razvil grafičnega odjemalca za testiranje, ki nazorno prikazuje zmogljivost sistema, rabi pa tudi kot primer uporabe sistema na uporabniški strani.

5.2 Tehnologije

Vsak sistem je skupek tehnologij, ki so že razvite in zelo olajšajo in skrajšujejo razvoj potrebnih funkcij, ki so ponavadi za razvijalce precej suhoparne. Bistveno pri tem pa je, da uporaba zrelih tehnologij zelo zmanjšuje možnost vnosa človeških napak v sistem.

V sistemu so uporabljene zelo pestre tehnologije. Vse so sodobne, celo zelo nove, a preverjene.

Programski jezik je Java. Je objektno usmerjen, moderen, razmeroma preprost, varen in nevtralen do platforme [4]. Java je tudi najbolj priljubljena izbira med programerji [21] in na koncu koncev je priporočena s strani podjetja. Nekaj skript je napisanih v javi sorodnem skriptnem programskem jeziku Groovy.

Sistem potrebuje **podatkovno zbirko** (angl. database), ki omogoča hranjenje in analizo nad geografskimi podatki. Uporabljena je običajna relacijska podatkovna zbirka **PostgreSQL**, ki skupaj z dodatkom za geografske podatke **PostGIS**, standardizira obdelavo prostorskih podatkov. PostGIS je najbolj smotrna izbira, saj je zastonj, preverjena in funkcionalno ne zaostaja za velikimi. Prostorski podatki so zahtevnejši od običajnih, saj nastopajo v več dimenzijah (običajno v dveh, lahko pa z dodatnim podatkom o višini tudi v treh). Indeksira se jih s posebno vrsto indeksa, ki se imenuje GiST (Generalized Search Tree). GiST ima to lastnost, da lahko indeksira nenavadne podatke, med njimi tudi večdimenzijske prostorske podatke.

Vez med aplikacijo in podatkovno zbirko je v javinem svetu tako imenovan gonilnik, spisan v programskem vmesniku JDBC (Java Database Connectivity). Komunikacija z zbirko je prek JDBC rahlo zapletena in razvoj zamuden, zato se danes uporabljajo višje nivojske tehnologije. Zelo primerna je **tehnologija preslikave relacijskih podatkov v objekte**, ki so običajni v objektnih programskih jeziki. Tehnologija ORM (Object-Relational Mapping), resnično, skoraj do skrajnosti, poenostavi delo s podatki v relacijski podatkovni zbirki.

Uporabil sem preizkušeno knjižnico **Hibernate**, ki svoje naredi tudi s posebnim **poizvedovalnim jezikom HQL** (angl. Hibernate Query Language), ki je objektno usmerjen.

Kako ORM deluje, je najbolje prikazati na primeru.

Recimo, da imamo preprost razred v javi, ki definira objekt *Student*:

```
public class Student {
  private Long id;
  private String name;

  public Long getId() {
    return this.id;
  }

  public void setId(Long id) {
    this.id=id;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String name) {
    this.name=name;
  }
}
```

Koda 5.1: Preprost razred Student v javi.

Objekt je enostaven, brez konstruktorjev s parametri in omogoča zunanji dostop do lastnosti prek *getterjev* in *setterjev*. Takšnemu objektu pravimo **zrno** (angl. JavaBean* oz. krajše kar bean) in je tipičen predstavnik POJO (Plain Old Java Object) [8, 9].

V relacijski podatkovni zbirki pripravimo entiteto (tabelo) *Student*, ki ima naslednja atributa (stolpca):

- *id* kot primarni (indeksiran) ključ in
- *name* kot niz znakov.

Poizvedba HQL, s katero bi iz baze pridobili vse študente z imenom "Janez", je preprosta:

```
from Student where name='Janez'
```

Koda 5.2: Poizvedba HQL, ki najde vse študente z imenom "Janez".

Njen ekvivalent v jeziku SQL, bi bil:

```
select * from Student where name='Janez'
```

Koda 5.3: Poizvedba SQL, ki najde vse študente z imenom "Janez".

V pomnilnik aplikacije pa bi se naložili ob klicu:

```
List<Student> students=session.createQuery("from Student where name=?")
  .setString(1, "Janez").list();
```

* JavaBean (in POJO) je po dogovoru objekt, ki ga je možno serializirati. Objekt, podan v primeru, ni takšen.

Koda 5.4: Nalaganje študentov z imenom "Janez" iz podatkovne zbirke v pomnilnik aplikacije s pomočjo poizvedbe HQL.

Tako imamo v seznamu *students* vse podatke o študentih Janezih, ki se nahajajo v bazi, ne da bi se ukvarjali s komunikacijo z zbirko ali pretvarjanjem entitet v objekte. To za nas stori Hibernate, pri tem pa uporablja vrsto zanimivih funkcij, kot je predpomnilnik (angl. cache), tako da je hitrost na zavidljivi ravni in vsekakor višja, kot če bi razvijalec podobno poskusil izvesti sam prek JDBC-ja.

Definirati moramo še **preslikavo objektov v entitete**, tako da Hibernate natančno ve, s kakšnimi tipi podatkov ima opravka in kako so objekti oz. entitete med seboj povezani. To lahko storimo na dva načina: preslikave podamo v datoteki XML ali jih neposredno označimo na objektih. Slednje je z novejšo tehnologijo oznak v javi (Java Annotations) zelo prikladno. Vsak način ima svoje dobre in slabe lastnosti. V sistemu se uporabljajo oznake, ki so večinoma definirane v vmesniku JPA (angl. Java Persistence API), nekaj pa je lastnih Hibernateovih. Oznake imajo to dobro lastnost, da na morebitne napake ali nevarnosti opozori že prevajalnik oz. že razvojno okolje.

Objektu *Student* bi takole označili preslikovanje:

```
@Entity
public class Student {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Long id;

  @Column
  private String name;

  // ...
}
```

Koda 5.5: Preslikovanje razreda Student v entiteto. Getterji in setterji so izpuščeni, ker so enaki kot pri kodi 5.1.

Ker sistem dela s posebnimi podatki - geografskimi, zanje Hibernate nima pripravljenih funkcij za preslikovanje. Zanje obstaja knjižnica **Hibernate Spatial**, ki se postavi med PostgreSQL JDBC gonilnikom in Hibernateom kot posebno narečje zbirke.

Dober sistem mora nujno imeti učinkovito, preprosto in pregledno zasnovo, da je skupek vseh tehnologij in lastne programske kode sploh obvladljiv. Tukaj na pomoč priskoči odlično **ogrodje Spring Framework**, ki deluje kot močno in stabilno lepilo. Njihova tehnologija omogoča, da lahko vse programske dele aplikacije preprosto povežemo skupaj z eno datoteko XML. Pri tem je pomembno le, da je vsak programski del, na najnižjem nivoju je to objekt, napisan na preprost način, v obliki zrn, kot je, npr., naš *Student*. V datoteki XML definiramo tudi **lastnosti posameznih zrn**, ki se vstavijo (temu pravijo *Dependency Injection*) šele po njihovi tvorbi (kar je ravno obratno od vstavljanja lastnosti ob tvorbi s konstruktorjem, zato temu pravijo *Inversion of Control*) [23].

Skrajno enostavna datoteka XML bi izgledala takole:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="janez" class="Student">
    <property name="id">
      <value>512</value>
    </property>
    <property name="name">
      <value>Janez Krajnski</value>
    </property>
  </bean>

</beans>
```

Koda 5.6: Zrno janez v Springovi datoteki XML, ki predstavlja eno instanco objekta Student z ID-jem 512 in imenom "Janez Krajnski".

Rezultat njene uporabe v ogrodju Spring pa bi bila ena instanca objekta *Student* z ID-jem "512" in imenom "Janez Krajnski".

Knjižnica Spring ima prvorazredno podporo ravno za Hibernate, zato je delo z njim tako rekoč nezaznavno. Celotno konfiguracijo lahko opišemo v eni datoteki XML in že imamo podporo za dostop do baze DAO (Database Access Object). Neznanka *session* v primeru nalaganje vseh študentov (koda 5.4) tako hitro postane le ena izmed zrn, dodatna podpora DAO (*HibernateTemplate*) pa še vse skupaj poenostavi:

```
List<Student> students=getHibernateTemplate().find(
  "from Student where name=?", "Janez");
```

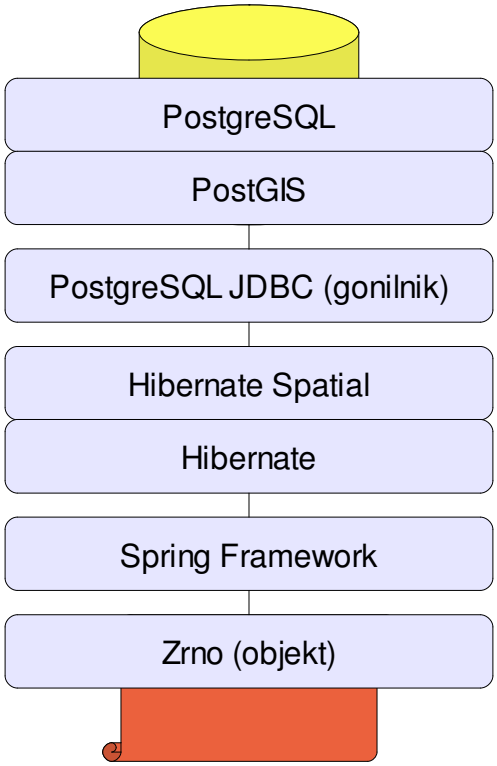
Koda 5.7: Nalaganje študentov z imenom "Janez" iz podatkovne zbirke v pomnilnik aplikacije s pomočjo poizvedbe HQL in Springove podpore DAO (HibernateTemplate).

Ogrodje Spring priskoči na pomoč tudi pri **povezavi med odjemalcem in strežnikom**. Povezava je narejena s tehnologijo **HTTP Invoker**. Ta je zelo podobna javini (Sunovi) lastni tehnologiji RMI (Remote Method Invocation), omogoča pa klicanje metod na daljavo tako, da sploh ne opazimo, da je določena metoda na drugi strani omrežja. HTTP Invoker pred klicem oddaljene metode morebitne parametre serializira (angl. serialize), to je pretvori v nezamenljiv niz zlogov, in jih pošlje v obliki zahteve (angl. request) prek standardnega spletnega protokola HTTP (Hypertext Transport Protocol). Strežnik metodo izvede in v odgovoru (angl. response) vrne njene morebitne rezultate (serializirane). Ker HTTP Invoker uporablja protokol HTTP, so morebitne motnje na omrežju nezaznavne, tudi ob prekinitvi povezave zna HTTP Invoker samodejno vzpostaviti novo. Dodatna prednost uporabe HTTP-ja je tudi ta, da gre za enega najbolj uporabljanih protokolov (brskanje po spletnih straneh), zato so po navadi požarni zidovi do njega prizanesljivi.

Če na kratko povzamem; med aplikacijo in podatkovno zbirko sistem uporablja celo vrsto tehnologij, ki skrajno poenostavljajo delo s podatki. Tako se lahko programer bolje osredotoči

na sam problem, ki ga želi rešiti. Nivoji so naslednji:

1. PostgreSQL
2. PostGIS
3. PostgreSQL JDBC (gonilnik)
4. Hibernate Spatial
5. Hibernate
6. Spring Framework
7. Zrno (objekt)



Slika 5.2: Nivoji tehnologij v sistemu od podatkovne zbirke (zgoraj) do domenskih objektov (zrna).

5.3 Okolje

Za razvojno okolje (angl. Integrated Development Environment - IDE) sem na koncu uporabljal brezplačno orodje **SpringSource Tool Suite** (STS), ki je maksimalno prilagojeno uporabi ogrodja Spring (izdelovalec ogrodja je podjetje SpringSource). Temelj STS-ja je platforma Eclipse.

V veliko pomoč pri razreševanju odvisnosti od knjižnic (pravijo jim *dependencies*) je bilo orodje **Apache Maven**.

Za testiranje sem uporabljal ogrodje JUnit, ki je preprosto in že vgrajeno v STS.

Vsa orodja so brezplačna za osebno in komercialno rabo, večina je odprtokodnih. Nobeno od orodij ni vezano na vrsto operacijskega sistema, pač pa na inačico javinega pogonskega okolja.

Celoten sistem je spisan v programskem jeziku Java, različice 6.0, zato za pogon (angl.

runtime environment) potrebuje JVM (Java Virtual Machine) različice vsaj 1.6. Nekaj skript je napisanih v skriptnem programskem jeziku Groovy, različice 1.6, ki pa so namenjene samo začetni konfiguraciji sistema.

Strežniški deli (storitve) za delovanje **potrebujejo vsebnik** (angl. servlet container), saj so spisane v obliki spletnih aplikacij. Dva najbolj znana sta Apache Tomcat in Mortbay Jetty. Slednji se v vdelani (angl. embedded) obliki uporablja tudi pri testiranju storitev.

5.4 Paketi

Celoten sistem je sestavljen modularno iz treh končnih paketov, vsak paket predstavlja svojo storitev. Vse tri storitve so odvisne od knjižnic (angl. library, dependency), med katerimi so tudi lastne. Med njimi ena vsebuje vse o podatkovnem modelu. Ostale knjižnice so od zunanjih razvijalcev, vse pa so brezplačne.

Seznam končnih paketov, strežnikov storitev, ki se **neposredno poženejo v okolju**:

- tinel-gis-geocoding-server - vsebuje storitev geokodiranja
- tinel-gis-reversegeocoding-server - vsebuje storitev nasprotnega geokodiranja
- tinel-gis-routing-server - vsebuje storitev usmerjanja

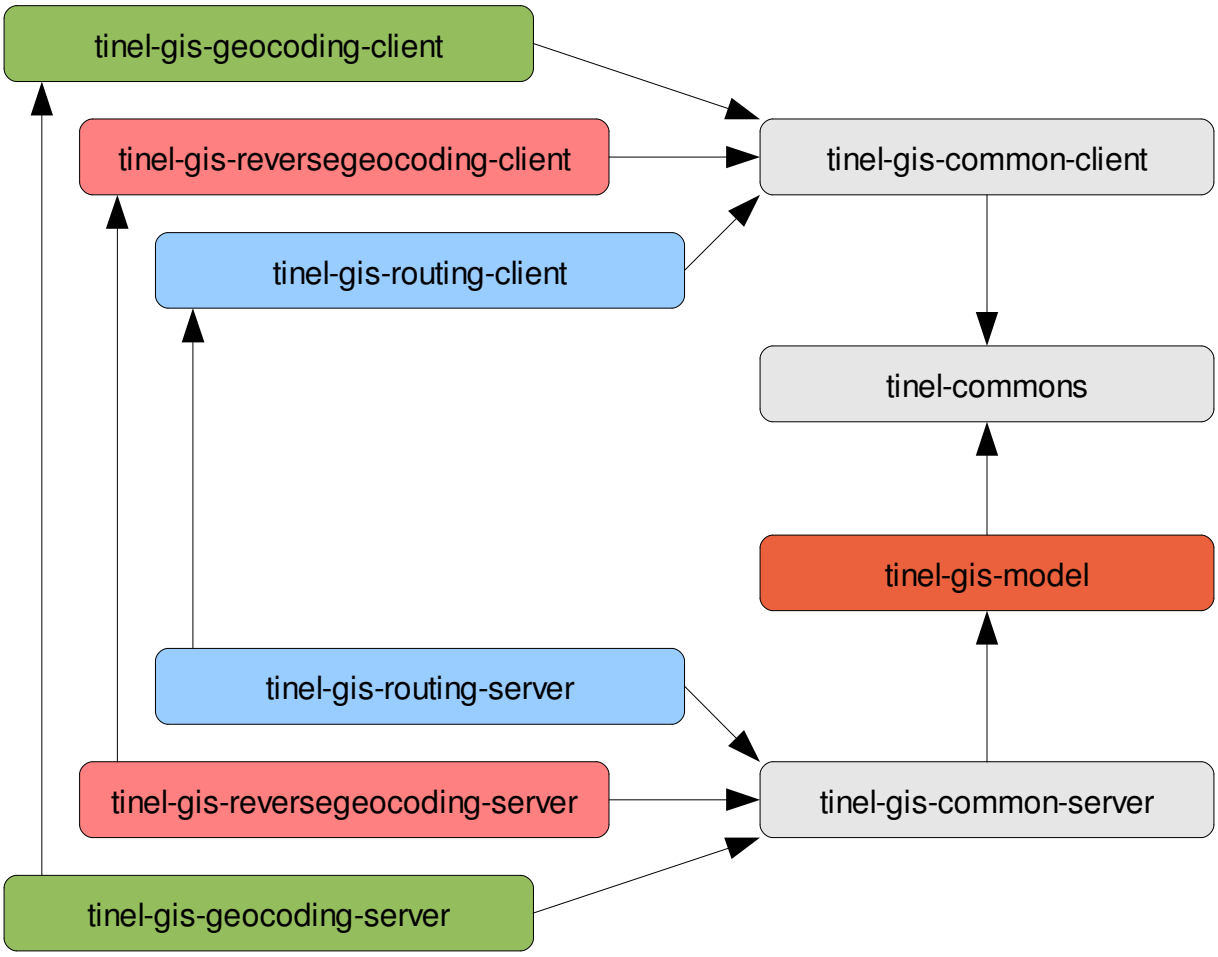
Vsi strežniki implementirajo vmesnike storitev, ki jih izrabljajo **odjemalci**:

- tinel-gis-geocoding-client - vsebuje vmesnike storitve geokodiranja
- tinel-gis-reversegeocoding-client - vsebuje vmesnike storitve nasprotnega geokodiranja
- tinel-gis-routing-client - vsebuje vmesnike storitve usmerjanja

Vsi paketi poleg definicij storitev za odjemalce za delovanje potrebujejo še **lastne knjižnice**:

- tinel-commons - splošna lastna orodja
- tinel-gis-common-server - splošna orodja za strežnike
- tinel-gis-common-client - splošna orodja za odjemalce
- tinel-gis-model - podatkovni model (preslikava ORM, DAO)

Celoten sistem je torej sestavljen iz desetih lastnih paketov.



Slika 5.3: Paketi, iz katerih je sestavljen sistem. Zgoraj so paketi, ki jih potrebujejo odjemalci, spodaj so implementacije strežniških storitev.

Dodatno sem za potrebe testiranja izdelal **grafičnega odjemalca**, ki zna uporabljati vse tri storitve, nahaja pa se v paketu:

- tinel-gis-client

Priložen je tudi primer uporabe celobesedilnega iskanja pri storitvi geokodiranja, nahaja pa se v paketu:

- tinel-gis-geocoding-compass

Poleg lastnih paketov, sistem uporablja še naslednje **zunanje knjižnice** (naštete so samo najbolj pomembne):

- hibernate - jedro Hibernate
- hibernate-annotations - dodatne oznake Hibernate za preslikavo ORM
- hibernate-spatial - dodatek za prostorske podatke za Hibernate
- hibernate-spatial-postgis - narečje PostGIS za Hibernate za prostorske podatke
- spring - ogrodje Spring Framework
- jts - implementacija prostorskih podatkov in orodij po standardih OGC

Za dnevniške vnose:

- commons-logging - vsebuje vmesnike za pisanje dnevniških vnosov

- log4j - stroj za pisanje dnevniških datotek

Za potrebe testiranja pa:

- junit - ogrodje za enotsko testiranje (angl. unit testing)
- spring-test - ogrodje za integracijsko testiranje (angl. integration testing)
- easymock - pripomočki za enotsko testiranje

Testni odjemalec potrebuje še cel kup knjižnic GeoTools, ki jih potrebuje predvsem za prikaz oz. projekcijo prostorskih podatkov. Priložen primer uporabe celobesedilnega iskanja pa ogrodje Compass, ki vsebuje iskalni stroj.

Vsi trije glavni paketi se s pomočjo orodja Maven zapakirajo v tako imenovane datoteke **WAR** (Web Application Archive), ki vsebujejo prevod, vse potrebne knjižnice in nastavitvene podatke za neposredno poganjanje v vsebniku.

Ostali paketi so predstavljeni zgolj kot knjižnice za nadaljnjo uporabo, zato se zapakirajo v datoteko JAR (angl. Java Archive).

Vsi paketi z izvorno kodo in knjižnicami so priloženi na CD plošči.

5.5 Vključitev v lastni sistem

Sistem je zasnovan modularno. Če ga želimo vključiti oz. uporabiti v svojem že obstoječem ali novem sistemu, moramo **na odjemalčevi strani**:

1. uporabiti knjižnice tinel-gis-geocoding-client, tinel-gis-reversegeocoding-client ali tinel-gis-routing-client, odvisno od tega, katero storitev potrebujemo,
2. vzpostaviti HTTP Invokerja za vsako storitev posebej in
3. uporabiti pripravljene metode storitev.

```
<!-- HTTP Invoker proxy -->
<bean id="geocodingService"
  class="org.springframework.remoting.httpinvoker
    .HttpInvokerProxyFactoryBean" lazy-init="true">
  <property name="serviceUrl">
    <value>http://localhost:9082/tinel-gis-
      geocoding/remoting/GeocodingService</value>
  </property>
  <property name="serviceInterface">
    <value>net.tinelstudio.gis.geocoding.service.GeocodingService</value>
  </property>
</bean>
```

Koda 5.8: Primer vzpostavitve HTTP Invokerja za storitev geokodiranja v obliki Springovega zrna v datoteki XML.

Ob uporabi zrna *geocodingService*, se bo HTTP Invoker samodejno povezal na strežniško storitev (na naslovu *http://localhost:9082*).

```
@Autowired
private GeocodingService geocodingService;

@Test
public void testGeocodingService() throws Exception {
  Locator locator=...;
  List<?> results=this.geocodingService.find(locator);
}
```

Koda 5.9: Primer uporabe zrna storitve geokodiranja, ki smo ga prej definirali v Springovi datoteki XML (koda 5.8). Ustvarjanje iskalca (locator) je zaradi večje preglednosti izpuščeno.

V zgornjem primeru (koda 5.9) prepustimo Springu, da ustvari *GeocodingService* (z oznako *@Autowired*). *GeocodingService* je v resnici samo vmesnik (angl. interface) storitve geokodiranja, klici njegovih metod pa se prek HTTP Invokerja prenašajo na strežnik. Pred klicem metode *find*, je treba definirati še iskalca (*locator*), kar sem zaradi večje preglednosti izpustil.

Na strežniški strani moramo prilagoditi ali podatkovni model obstoječim podatkov ali podatke podatkovnemu modelu. V obeh primerih nas zanima samo knjižnica tinel-gis-model, kjer so definirane preslikave ORM in DAO. Če bomo prilagodili podatkovni model obstoječim podatkom, se bomo lotili popravljanja preslikav ORM in prilagajanja DAO, v kolikor imamo namen prilagoditi obstoječe podatke novemu podatkovnemu modelu, si bomo verjetno pripravili nekaj skript za pretvarjanje podatkov, v pomoč pa nam bo obstoječi DAO.

6 Podatkovni model

Glavni cilj podatkovnega modela je **poenotenje strukture** podatkov tako, da ga vse storitve lahko izkoriščajo in da se podatki **nikoli ne podvajajo**. S tem pridobimo na **preglednosti** (vse storitve uporabljajo isti model brez izjem) in **enostavnem vzdrževanju**.

Podatkovni model, ki je skupen več storitvam, zna biti zahtevna zadeva, sploh če želimo vanj vstaviti vrsto različnih geografskih podatkov iz različnih koncev sveta. Namreč, razlika v cestno naslovnem sistemu, npr., v ZDA in pri nas je kar precejšnja. Ravno zaradi tega ne obstaja idealen model, ki bi hkrati veljal za cel svet, temveč so GIS-i **tipično lokalne narave**.

Eden takšnih je opisan na spletnih straneh ESRI-ja [12], ki so ga izdelali za lokalne namene kanadskega mesta Calgary, a je bil v veliko pomoč pri snovanju tega.

Skušal sem narediti preprost podatkovni model, ki je primeren za večino dežel, njegova vrlina pa je predvsem enostavnost. Hrani podatke v različnih nivojih, relacijah. Model je karseda uniformen, načeloma lahko vanj vstavimo geografske **podatke z vsega sveta**. Je tudi zelo **fleksibilen**, tako da ga je možno razširiti z dodatnimi in zahtevnejšimi podatki.

6.1 Domena

Elementom, ki vsebujejo edinstvene (unikatne) podatke, v podatkovnem žargonu pravimo podatkovna domena (angl. data domain). V našem primeru so ti elementi na aplikacijski strani preprosto **objekti**, na strani podatkovne zbirke pa **entitete**. Med seboj se preslikujejo s pomočjo že opisane tehnologije objektno relacijskega preslikovanja ORM.

Domenski objekti za sistem so spisani v javinem paketu net.tinelstudio.gis.model.domain.

6.1.1 GeoName

Osrednji domenski objekt je *GeoName*. Predstavlja vsa takšna in drugačna **geografska imena** (geoimena) skupaj z vrsto podatka.

GeoName:

- *name* - ime (niz znakov)
- *type* - vrsta podatka (niz znakov)

Vrsta podatka (*type*) je lahko **naslov** (ADDRESS), **cesta/ulica** (STREET), **zgradba** (BUILDING), **mesto/vas** (TOWN), **regija/dežela/provinca/prefektura** (REGION), **država** (COUNTRY) ali **kontinent** (CONTINENT). Podatki so čim bolj podobni Google Maps API-ju [18].

Novo vrsto podatka je možno enostavno dodati. Na primer, če imamo podatke o železnicah, bi dodali novo vrsto RAILWAY, ali če imamo podatke o vodnih linijah, kot so reke in potoki, bi dodali vrsto WATERLINE.

Vrsta STREET v grobem predstavlja imena cest oz. ulic, ki nimajo naslovov (avtoceste, magistralke) in imajo le oznake (npr. "A1", "E57"). Vrsta ADDRESS predstavlja imena cest oz. ulic, ki imajo ulične naslove (npr. "Tržaška cesta").

V imenu je za razliko od nekaterih drugih podatkovnih modelov zapisano kar celotno ime (skupaj s "cesta", "ulica" ipd.) in indeksirano po celem imenu. S tem se pri ugotavljanju polnih imen (npr. risanje cest na karto) izognimo težavnim delom, ko je lahko ulica zapisana na različne načine (npr. "Ulica 5. maja" in "Gubčeva ulica").

```
package net.tinelstudio.gis.model.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;

/**
 * The main entity providing only geographic names.
 *
 * @author TineL
 */
@Entity
public class GeoName extends BaseEntity {

  @Column(nullable=false)
  private String name;

  @Column(nullable=false)
  @Enumerated(EnumType.STRING)
  private Type type;

  /**
   * The GeoName types.
   *
   * @author TineL
   */
  public enum Type {

    /** The geographic name at the address level accuracy. */
    ADDRESS,

    /** The geographic name at the street level accuracy. */
```

```
    STREET,

    /** The geographic name at the building (premise, property) level
        accuracy. */
    BUILDING,

    /** The geographic name at the town (city, village) level accuracy. */
    TOWN,

    /**
     * The geographic name at the region (state, province, prefecture,
     * etc.) level accuracy.
     */
    REGION,

    /** The geographic name at the country level accuracy. */
    COUNTRY,

    /** The geographic name at the continent level accuracy. */
    CONTINENT
  }

  // ...
}
```

Koda 6.1: Domenski objekt GeoName. Getterji in setterji so izpuščeni.

6.1.2 Address

Naslov (*Address*) je eden najpomembnejših domenskih objektov. V prvi vrsti se uporablja tako rekoč povsod, pri vseh storitvah, saj vsebuje najbolj natančen in za uporabnike zanimiv podatek: lokacijo na **hišno številko** natančno.

Address:

- *point* - prostorska lokacija (točka)
- *geoNames* - pripadajoča geoimena (GeoName; kardinalnost je več:več)
- *houseNumber* - hišna številka (niz znakov)
- *note* - morebitna opomba (niz znakov)

Predstavlja en naslov s točno določeno geografsko lokacijo.

Zaradi fleksibilne zasnove (relacija več:več z objektom *GeoName*) lahko ima naslov **več imen** (ime ulice, ime mesta, ime države, ime kontinenta ...). Lahko ima tudi več imen iste vrste, npr., ulic. Prek geoimen s pravilno poizvedbo lahko hitro pridemo do, npr., dejanske ceste. To velja tudi za ceste in zgradbe.

```
package net.tinelstudio.gis.model.domain;

import java.util.Set;

import javax.persistence.Column;
import javax.persistence.Entity;
```

```
import javax.persistence.ManyToMany;

import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;
import org.hibernate.annotations.Type;

import com.vividsolutions.jts.geom.Point;

/**
 * The Address entity.
 *
 * @author TineL
 */
@Entity
public class Address extends NotedEntity {

  @Column(nullable=false)
  @Type(type="org.hibernatespatial.GeometryUserType")
  private Point point;

  @Column(nullable=false)
  @ManyToMany
  @Cascade(value=CascadeType.SAVE_UPDATE)
  private Set<GeoName> geoNames;

  @Column(nullable=false)
  private String houseNumber;

  // ...
}
```

Koda 6.2: Domenski objekt Address. Getterji in setterji so izpuščeni.

6.1.3 Street

Domenski objekt *Street* (cesta) predstavlja vse ceste in ulice z naslovi ob levi in desni strani. En element je vedno odsek ceste od enega vozlišča (angl. node) do drugega. Odsek ceste je sestavljen iz ene ali več linij.

Križišče cest mora biti vozlišče, da bo storitev usmerjanja dajala pravilne rezultate.

Nekatere daljše ceste imajo veliko odsekov, ker imajo veliko križišč (npr. Tržaška, Celovška, Dunajska ipd.).

En odsek ceste lahko ima **več imen** (npr. več uličnih imen in dodatno še oznako (avto)ceste).

Street:

- *lineString* - prostorski lokacija odseka (niz linij)
- *geoNames* - pripadajoča geoimena (GeoName; kardinalnost je več:več)
- *leftAddressRanges* - serija naslovov na levi strani (AddressRange; kardinalnost je več:več)
- *rightAddressRanges* - serija naslovov na desni strani (AddressRange; kardinalnost je več:več)
- *lengthMeters* - dolžina odseka v metrih (*integer*)

- *level* - nivo ceste (*integer*)
- *oneWay* - ali je cesta enosmerna (*boolean*)
- *startNode* - začetno vozlišče (StreetNode; kardinalnost je več:ena)
- *endNode* - končno vozlišče (StreetNode; kardinalnost je več:ena)
- *note* - morebitna opomba (niz znakov)

Atribut *level* pove velikost oz. pomembnost ceste, ki pride prav tako pri usmerjanju kot pri upodabljanju (angl. rendering) ceste na karto pri različnih povečavah (manjše ceste se ne rišejo pri majhnih povečavah). Za upodabljanje sta mišljena tudi atributa *leftAddressRange* in *rightAddressRange*.

```
package net.tinelstudio.gis.model.domain;

import java.util.Set;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.ManyToMany;
import javax.persistence.ManyToOne;

import net.tinelstudio.gis.common.dto.StreetDto.Level;

import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;
import org.hibernate.annotations.Type;

import com.vividsolutions.jts.geom.LineString;

/**
 * The Street entity.
 *
 * @author TineL
 */
@Entity
public class Street extends NotedEntity {

  @Column(nullable=false)
  @Type(type="org.hibernatespatial.GeometryUserType")
  private LineString lineString;

  @Column
  @ManyToMany
  @Cascade(value=CascadeType.SAVE_UPDATE)
  private Set<GeoName> geoNames;

  @Column
  @ManyToMany
  @Cascade(value=CascadeType.SAVE_UPDATE)
  private Set<AddressRange> leftAddressRanges;

  @Column
  @ManyToMany
  @Cascade(value=CascadeType.SAVE_UPDATE)
  private Set<AddressRange> rightAddressRanges;
```

```
  @Column(nullable=false)
  private Integer lengthMeters;

  @Column(nullable=false)
  /*
   * Warning: Street levels are persisted as ordinal integers from
   * enumeration.
   * Add new enumeration literals only to the end, do not delete any, or
   * else levels will be confused in the DB.
   */
  @Enumerated(EnumType.ORDINAL)
  private Level level;

  @Column
  private Boolean oneWay;

  @ManyToOne(optional=false)
  @Cascade(value=CascadeType.SAVE_UPDATE)
  private StreetNode startNode;

  @ManyToOne(optional=false)
  @Cascade(value=CascadeType.SAVE_UPDATE)
  private StreetNode endNode;

  // ...
}
```

Koda 6.3: Domenski objekt Street. Getterji in setterji so izpuščeni.

AdressRange:

- *fromAddress* - začetni naslov serije (Address; kardinalnost je več:ena)
- *toAddress* - končni naslov serije (Address; kardinalnost je več:ena)

Predstavlja serijo naslovov, ki imajo nekaj skupnega: so na isti cesti (npr. od "Tržaška cesta 1" do "Tržaška cesta 125").

```
package net.tinelstudio.gis.model.domain;

import javax.persistence.Entity;
import javax.persistence.ManyToOne;

/**
 * The AddressRange entity.
 *
 * @author TineL
 */
@Entity
public class AddressRange extends BaseEntity {

  @ManyToOne(optional=false)
  private Address fromAddress;

  @ManyToOne(optional=false)
  private Address toAddress;

  // ...
```

```
}
```

Koda 6.4: Domenski objekt AddressRange. Getterji in setterji so izpuščeni.

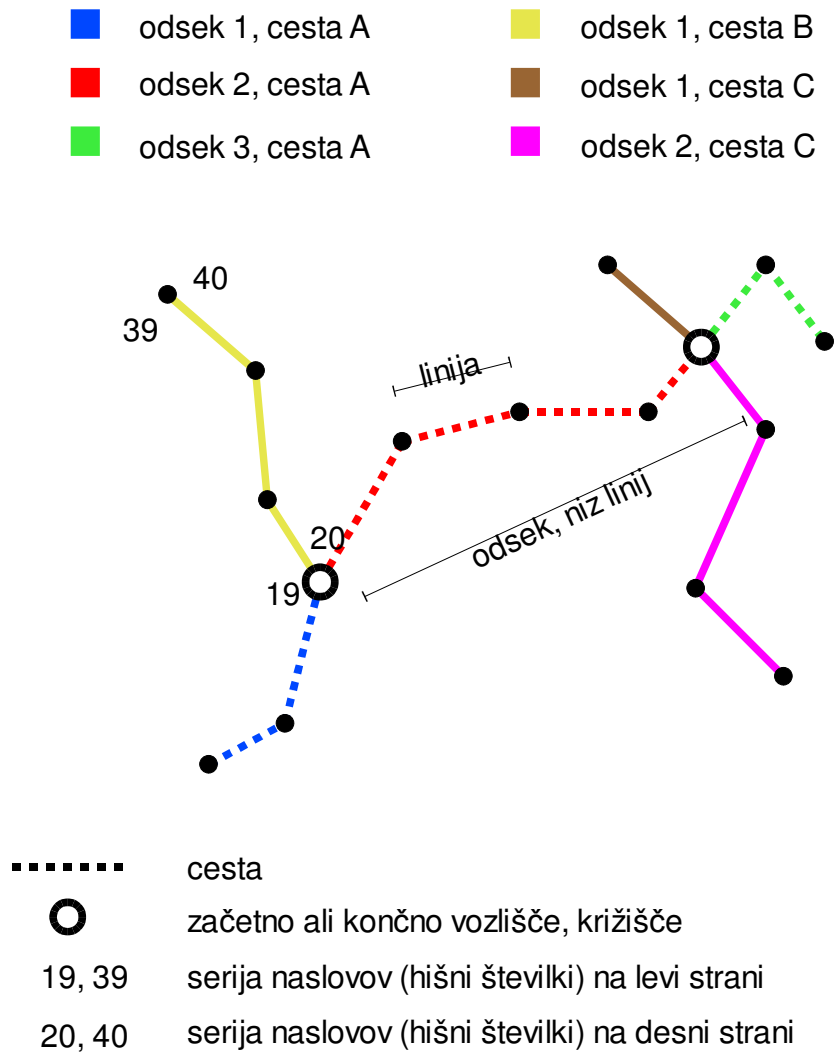
StreetNode:

- *point* - prostorska lokacija (točka)

Predstavlja vse začetne in končne točke vseh odsekov cest. Uporablja se pri usmerjanju za iskanje povezav med odseki.

```
package net.tinelstudio.gis.model.domain;

import javax.persistence.Column;
import javax.persistence.Entity;

import org.hibernate.annotations.Type;

import com.vividsolutions.jts.geom.Point;

/**
 * The StreetNode entity.
 *
 * @author TineL
 */
@Entity
public class StreetNode extends BaseEntity {

  @Column(nullable=false)
  @Type(type="org.hibernatespatial.GeometryUserType")
  private Point point;

  // ...
}
```

Koda 6.5: Domenski objekt StreetNode. Getterji in setterji so izpuščeni.



Slika 6.1: Zgradba ceste. Slika prikazuje cesto, cestne odseke ali nize linij, vozlišča in serije naslovov.

6.1.4 Building

Zgradba (*Building*) je površinski geografski element, ki lahko predstavlja marsikaj: osebne hiše, poslovne objekte, javne centre in tovarne. Lahko ima **več splošnih imen** (npr. ime muzeja, bolnišnice, avtobusne postaje, gradu, zanimive zgradbe ...).

Building:

- *polygon* - prostorska površina (poligon)
- *geoNames* - pripadajoča geoimena (GeoName; kardinalnost je več:več)
- *note* - morebitna opomba (niz znakov)

```
package net.tinelstudio.gis.model.domain;

import java.util.Set;
```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.ManyToMany;

import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;
import org.hibernate.annotations.Type;

import com.vividsolutions.jts.geom.Polygon;

/**
 * The Building entity.
 *
 * @author TineL
 */
@Entity
public class Building extends NotedEntity {

  @Column(nullable=false)
  @Type(type="org.hibernatespatial.GeometryUserType")
  private Polygon polygon;

  @Column
  @ManyToMany
  @Cascade(value=CascadeType.SAVE_UPDATE)
  private Set<GeoName> geoNames;

  // ...
}
```

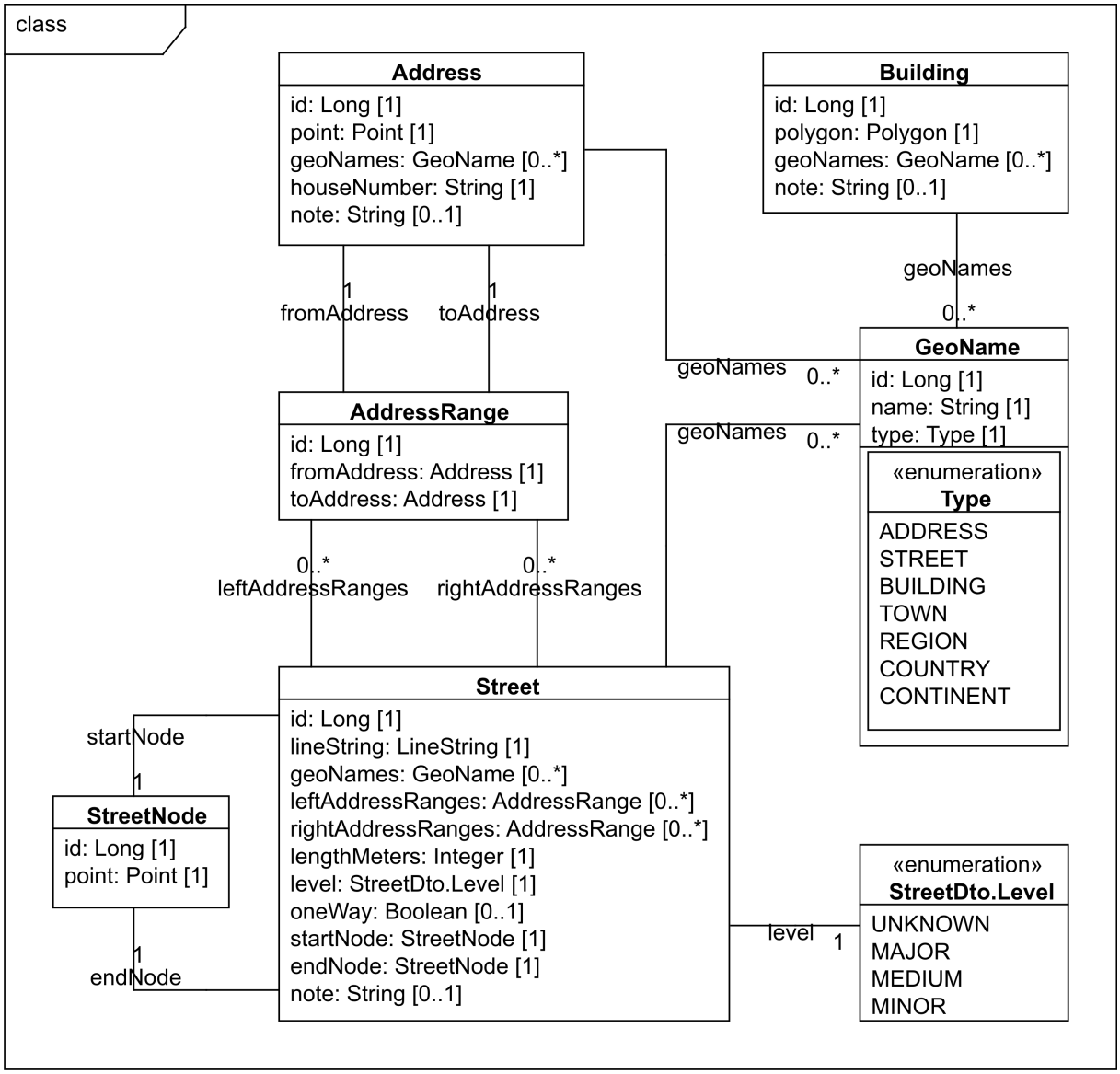
Koda 6.6: Domenski objekt Building. Getterji in setterji so izpuščeni.

Prav vsi domenski objekti imajo še atribut:

- *id* - identifikacija,

ki ga dedujejo od razreda *BasicEntity* ali *NotedEntity*. *NotedEntity* vsebuje poleg atributa *id*, še atribut *note*.

Identifikacija je entiteti primarni ključ.



Slika 6.2: Razredni diagram domenskih objektov.

Slika 6.3: Primer upodobitve geografskih krajev, ki jih predstavljajo domenski objekti. Rdeči kvadratki so naslovi, rumene črte so ceste, oranžni krogci so vozlišča in sivi liki so zgradbe.

6.2 Model entitetnih razmerij

Model entitetnih razmerij (angl. Entity-Relationship Model) je abstraktna predstavitev podatkov v konceptualni obliki. Uporablja se za **načrtovanje in prikaz modela** neposredno v podatkovni zbirki.

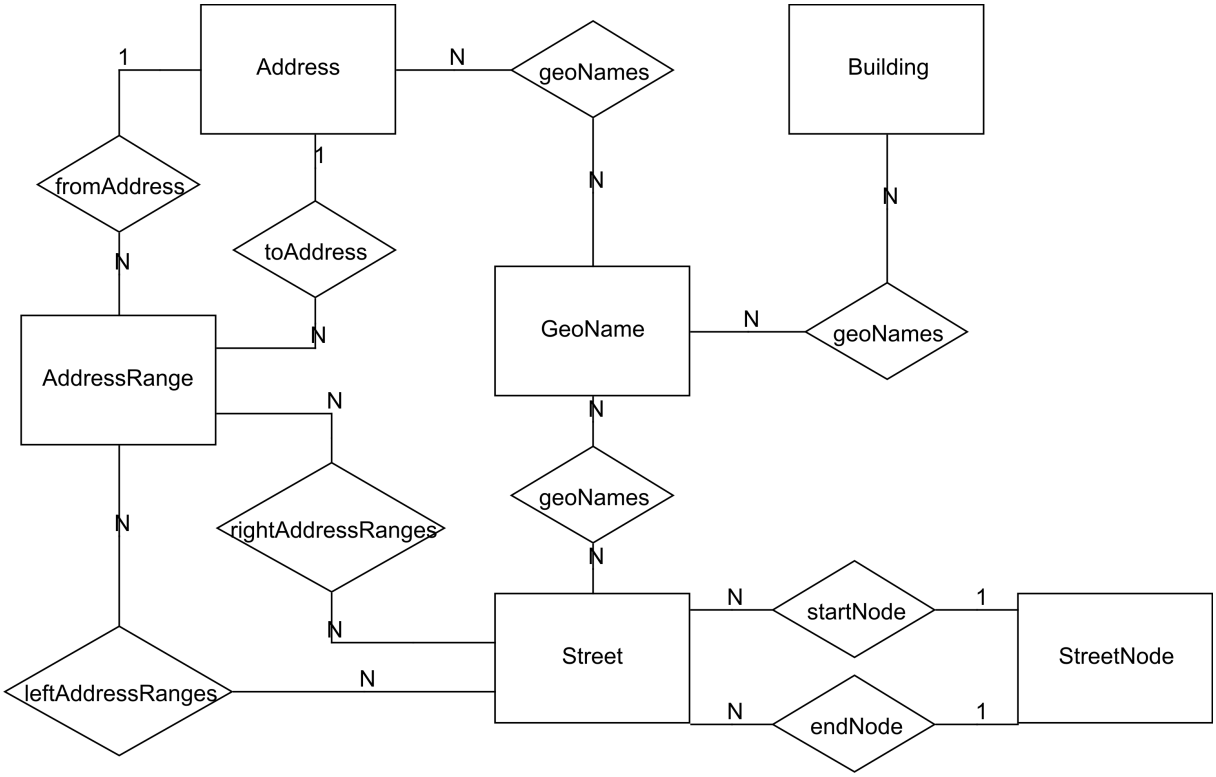
Ker je sistem zasnovan z druge strani, to je s strani aplikacije, se domenski objekti nezamenljivo preslikajo v entitete in razmerja med njimi (ORM). Model v sami podatkovni zbirki lahko ustvarimo ročno oz. prek skripte SQL. Lahko pa ustvarimo le prazno zbirko in vso ostalo delo prepustimo kar Hibernateu.

V skriptnem programskem jeziku Groovy, ki je soroden javi, sem napisal skripto, ki ustvari prazno podatkovno zbirko z vsemi entitetami, njihovimi razmerji in potrebnimi indeksi. Deli kode se lahko uporabijo za morebitno kasnejše spreminjanje modela, ko so podatki že vstavljeni.

Podatkovni model je v skladu s standardi OGC oz. v skladu s zahtevami PostGIS (ki upoštevajo standarde OGC). Ti med drugim definirajo dve obvezni dodatni tabeli [24]:

- *geometry_columns*, ki vsebuje podatke o prostorskih podatkih (ime sheme, tabele in stolpca, dimenzija, SRID, vrsta podatka), ki se nahajajo v entitetah, in
- *spatial_ref_sys*, ki vsebuje seznam podprtih geografskih referenčnih sistemov.

PostGIS obe tabeli aktivno interno uporablja, kadar kličemo njegove funkcije. Zato obstaja tudi posebna funkcija *AddGeometryColumn*, ki doda stolpec s prostorskimi podatki v tabelo. Slednje skripta, ki ustvari podatkovno zbirko z entitetami, upošteva.



Slika 6.4: Diagram entitetnih razmerij podatkovnega modela. Tabeli geometry_columns in spatial_ref_sys nista prikazani, ker sta predvsem administrativnega značaja.

6.3 Dostop do podatkov (DAO)

Sistem ima nekaj vnaprej pripravljenih vmesnikov, ki definirajo preizkušene dostope do podatkov v zbirki (DAO). Nahajajo se v javinem paketu net.tinelstudio.gis.model.dao.

Osnovni objekti DAO so za vsak domenski objekt posebej, definirajo pa tipične operacije za manipulacijo s podatki:

- nalaganje podatkov iz baze (angl. load),
- shranjevanje oz. posodabljanje podatkov v bazo (angl. save) in
- brisanje podatkov iz baze (angl. delete).

```
package net.tinelstudio.gis.model.dao;

import java.util.Collection;
import java.util.List;

/**
 * Defines a Domain Access Object (DAO) with typical load, save & delete
 * operations.
 *
 * @author TineL
 * @param <E> the domain object
 */
```

```
public interface LoadSaveDeleteDao<E> {

  E load(Long id);
  List<E> loadAll();

  void save(E element);
  void saveAll(Collection<E> elements);

  boolean delete(Long id);
  void deleteAll();
}
```

Koda 6.7: LoadSaveDeleteDao definira tipične operacije za manipulacijo objektov v podatkovni zbirki. Komentarji so izpuščeni.

Sistem ima v enem vmesniku, imenovanem *FindingDao*, definirane vse ključne, visoko učinkovite **metode za iskanje podatkov**, kar s pridom uporabljajo vse storitve. V enem vmesniku so zato, da ima razvijalec ob morebitni spremembe podatkovnega modela ali ob izboljšanju iskalnih poizvedb vse na enem mestu.

```
package net.tinelstudio.gis.model.dao;

import java.util.List;

import net.tinelstudio.gis.model.domain.Address;
import net.tinelstudio.gis.model.domain.Building;
import net.tinelstudio.gis.model.domain.Street;
import net.tinelstudio.gis.model.domain.StreetNode;
import net.tinelstudio.gis.model.domain.GeoName.Type;

import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.geom.Point;

/**
 * Defines a Domain Access Object with special finding operations.
 *
 * @author TineL
 */
public interface FindingDao {

  List<Address> findAddresses(List<String> names, List<String>
    houseNumbers, int maxResults);

  List<Address> findDistinctAddressesByType(List<String> names, Type type,
    int maxResults);

  List<Street> findStreets(List<String> names, int maxResults);

  List<Street> findDistinctStreetsByType(List<String> names, Type type,
    int maxResults);

  List<Building> findBuildings(List<String> names, int maxResults);

  List<Building> findDistinctBuildingsByType(List<String> names, Type type,
    int maxResults);

  List<Address> findNearestAddresses(Point point, int maxDistanceMeters,
```

```
    int maxResults);

  List<Street> findNearestStreets(Point point, int maxDistanceMeters,
    int maxResults);

  List<Building> findNearestBuildings(Point point, int maxDistanceMeters,
    int maxResults);

  StreetNode findNearestStreetNode(Point point, int maxDistanceMeters);

  List<Address> findNearestAddressesApprox(Geometry place,
    int maxDistanceMeters, int maxResults);

  List<Street> findNearestStreetsApprox(Geometry place, int
    maxDistanceMeters, int maxResults);

  List<Building> findNearestBuildingsApprox(Geometry place,
    int maxDistanceMeters, int maxResults);

  StreetNode findNearestStreetNodeApprox(Geometry place, int
    maxDistanceMeters);

  List<Street> findConnectedStreets(StreetNode streetNode);
}
```

Koda 6.8: FindingDao definira metode za iskanje ustreznih geografskih krajev po podatkovni zbirki. Komentarji so izpuščeni.

Vsi vmesniki so implementirani za uporabo s Hibernateom (nahajajo se v javinem paketu net.tinelstudio.gis.model.dao.hibernate). Na pomoč jim priskoči Springova predloga *HibernateTemplate*, s katero je možno napisati prav vse poizvedbe. *HibernateTemplate* hkrati še poenostavlja najbolj osnovno iskanje in skrbi za transakcije.

6.4 Morebitne razširitve

Primer bolj uporabne razširitve podatkovnega modela bi bila dodatna atributa v domeni zgradbe (*Building*):

- *addresses* - pripadajoči (pod)naslovi (Address; kardinalnost je več:več)
- *streets* - pripadajoče ceste (Street; kardinalnost je več:več)

```
@Column
@ManyToMany
@Cascade(value=CascadeType.SAVE_UPDATE)
private Set<Address> addresses;

@Column
@ManyToMany
@Cascade(value=CascadeType.SAVE_UPDATE)
private Set<Street> streets;
```

Koda 6.9: Dodatna atributa addresses in streets v domenskem objektu Building.

Zgradba v resničnem svetu načeloma lahko ima **več naslovov** oz. podnaslovov. Na primer, vhod iz ene strani zgradbe in vhod iz druge strani zgradbe. Prav tako je zgradba na vogalu ulice obdana z **več cestami**.

Med bolj običajne razširitve spadajo **novi domenski objekti**, za vsako vrsto podatka. Če bi dodali ozke tekoče vode, ki predstavljajo reke in potoke, bi lahko naredili tako:

V *GeoName* bi dodali novo vrsto podatka, npr., "WATERLINE".
Definirali bi nov domenski objekt:

WaterLine:

- *lineString* - prostorska lokacija linije (niz linij)
- *geoNames* - pripadajoča geoimena (GeoName; kardinalnost je več:več)
- *note* - morebitna opomba (niz znakov)

```
package net.tinelstudio.gis.model.domain;

import java.util.Set;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.ManyToMany;

import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;
import org.hibernate.annotations.Type;

import com.vividsolutions.jts.geom.LineString;

/**
 * The WaterLine entity.
 *
 * @author TineL
 */
@Entity
public class WaterLine extends NotedEntity {

  @Column(nullable=false)
  @Type(type="org.hibernatespatial.GeometryUserType")
  private LineString lineString;

  @Column
  @ManyToMany
  @Cascade(value=CascadeType.SAVE_UPDATE)
  private Set<GeoName> geoNames;

  // ...
}
```

Koda 6.10: Primer dodatnega domenskega objekta WaterLine. Getterji in setterji so izpuščeni.

7 Geokodiranje

Geokodiranje (angl. geocoding, address lookup) je storitev, ki poišče geografske kraje, ki ustrezajo podanemu iskalnemu nizu. Geografski kraji so predstavljeni s prostorsko lokacijo v geografskih koordinatah (zemljepisna dolžina in širina v stopinjah). Geokodiranje pride prav uporabnikom, ki želijo izvedeti, **kje se nahaja nek geografski kraj** (npr. naslov, cesta, zgradba ipd.), pri tem pa vedo samo (celo ali delno) ime kraja. V ustrezno opremljenem odjemalcu vnesejo iskalni niz, orodje jim poišče ustrezne zadetke, grafični odjemalec pa jih prikaže na zemljevidu.

Ustrezna koda se nahaja v javinem paketu net.tinelstudio.gis.geocoding.

7.1 Iskalci

Storitev geokodiranja je zasnovana tako, da odjemalec strežniku poda zahtevo v obliki *iskalca* (angl. locator). Iskalcev je **več vrst** in opravljajo **različne vloge**. Tako lahko uporabnik natančno definira, po kateri vrsti podatkov se naj išče.

Iskalec v bistvu obstaja v dveh oblikah. Tisti na strežniški strani (*LocatorService*) dejansko išče zadetke, tistega na odjemalčevi strani (*Locator*) pa uporabnik uporablja za nastavljanje iskanja. Ko uporabnik izbere iskalca, mu mora podati še:

- *searchString* - **iskalni niz** (geoime) in
- *maxResults* - največje število zadetkov.

Iskalni niz je lahko **cel** (predstavlja celotno geoime, npr. "Cankarjeva ulica") ali le **delen** (npr. "cank ul.").

Storitev pozna štiri vrste iskalcev:

1. Osnovni iskalec, ki išče samo po **glavnem imenu** neke vrste podatka (npr. samo po imenu naslova, ceste ali zgradbe).

    - **AddressLocator** - Išče hkrati po naslovih (geoimena vrste ADDRESS) in po

hišnih številkah. Če v iskalnem nizu ni nobene hišne številke, se iskanje sploh ne prične. To pride prav v kombinaciji z multi iskalci.
- **StreetLocator** - Išče po imenih cest (geoimena vrste STREET).
- **BuildingLocator** - Išče po imenih zgradb (geoimena vrste BUILDING).

2. Osnovni iskalec, ki išče po **drugih imenih** neke vrste podatka (npr. samo po imenu mesta ali države nekega naslova, ceste ali zgradbe).

   Zanimivo pri tej vrsti iskalca je, da vrne samo **en zadetek** na iskalno ime. Če je iskalni niz "Ljubljana", bo, npr., *DistinctAddressByTownLocator* vrnil samo prvi naslov v Ljubljani in ne vseh nekaj tisoč. To je uporabno takrat, kadar nas zanima samo, kje je določeno mesto (bolj na grobo), in ne podrobnosti mesta.

   - **DistinctAddressByNameLocator** - Išče samo po naslovih brez hišnih številk (geoimena vrste ADDRESS) in vrne samo en naslov na ime naslova.
   - **DistinctAddressByTownLocator** - Išče samo po imenih mest (geoimena vrste TOWN) in vrne samo en naslov na mesto.
   - **DistinctAddressByRegionLocator** - Išče samo po imenih regij (geoimena vrste REGION) in vrne samo en naslov na regijo.
   - **DistinctAddressByCountryLocator** - Išče samo po imenih držav (geoimena vrste COUNTRY) in vrne samo en naslov na državo.
   - **DistinctAddressByContinentLocator** - Išče samo po imenih kontinentov (geoimena vrste CONTINENT) in vrne samo en naslov na kontinent.
   - **DistinctStreetByTownLocator** - Išče samo po imenih mest (geoimena vrste TOWN) in vrne samo eno cesto na mesto.
   - **DistinctStreetByRegionLocator** - Išče samo po imenih regij (geoimena vrste REGION) in vrne samo eno cesto na regijo.
   - **DistinctStreetByCountryLocator** - Išče samo po imenih držav (geoimena vrste COUNTRY) in vrne samo eno cesto na državo.
   - **DistinctStreetByContinentLocator** - Išče samo po imenih kontinentov (geoimena vrste CONTINENT) in vrne samo eno cesto na kontinent.
   - **DistinctBuildingByTownLocator** - Išče samo po imenih mest (geoimena vrste TOWN) in vrne samo eno zgradbo na mesto.
   - **DistinctBuildingByRegionLocator** - Išče samo po imenih regij (geoimena vrste REGION) in vrne samo eno zgradbo na regijo.
   - **DistinctBuildingByCountryLocator** - Išče samo po imenih držav (geoimena vrste COUNTRY) in vrne samo eno zgradbo na državo.
   - **DistinctBuildingByContinentLocator** - Išče samo po imenih kontinentov (geoimena vrste CONTINENT) in vrne samo eno zgradbo na kontinent.

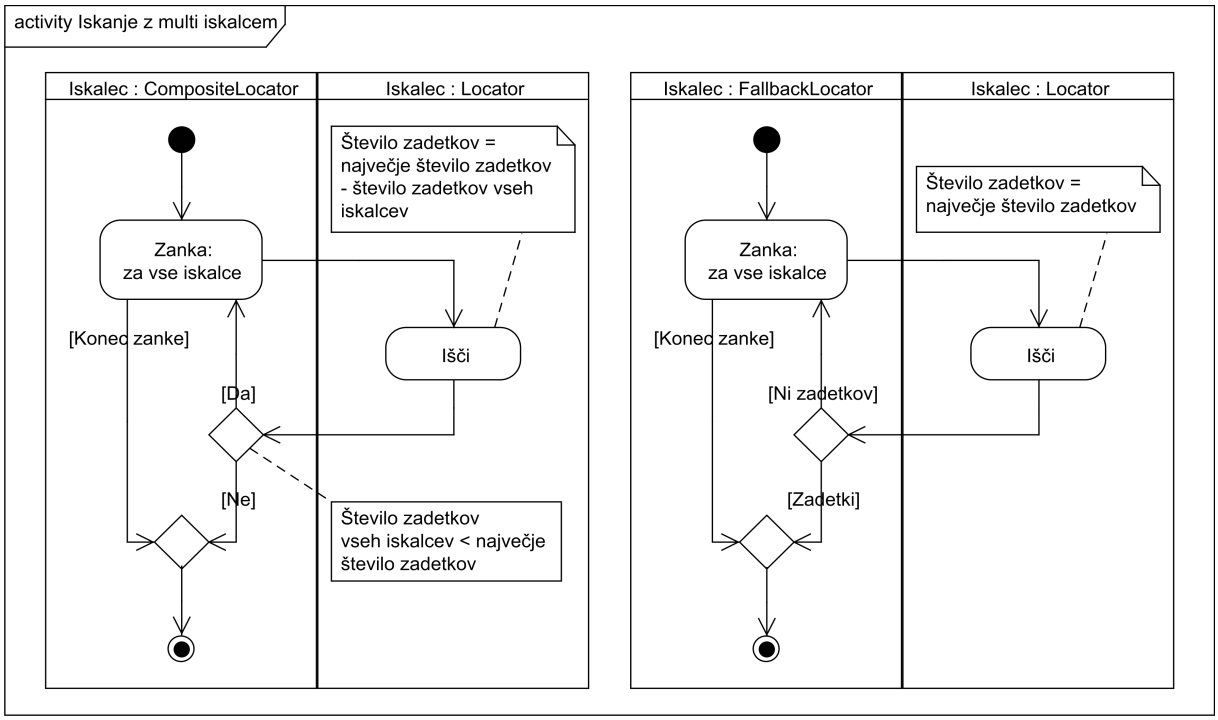
3. Multi iskalec, ki **kombinira več iskalcev**.

   Iskalec najprej delegira iskanje prvemu podanemu iskalcu, nato drugemu, nato tretjemu in tako dalje. Zadetke vseh iskalcev vrne v enem seznamu po vrsti tako, kot so bili najdeni.

   - **CompositeLocator** - Delegira iskanje po vseh navedenih iskalcih.
   - **FullCompositeLocator** - Predpripravljen iskalec, ki delegira iskanje po vseh osnovnih iskalcih.

4. Multi iskalec, ki preide na naslednjega iskalca, če prejšnji ne vrne zadetkov.
   Iskalec najprej delegira iskanje prvemu podanemu iskalcu. Če ta ne vrne nobenega zadetka, se iskanje delegira drugemu, drugače se takoj vrnejo zadetki samo prvega iskalca. Ostali sploh ne pridejo na vrsto.

   - **FallbackLocator** - Delegira iskanje po navedenih iskalcih, dokler eden ne vrne zadetkov.
   - **DefaultFallbackLocator** - Predpripravljen iskalec, ki smiselno delegira iskanje po osnovnih iskalcih, dokler eden ne vrne zadetkov.



Slika 7.1: Diagram aktivnosti iskanje z multi iskalcema CompositeLocator in FallbackLocator.

Z multi iskalci je možna zelo zapletena in fino nastavljiva **kombinacija vseh iskalcev**, tudi več multi iskalcev, npr.:

```
Locator locator=new FallbackLocator(new CompositeLocator(
  new StreetLocator(), new BuildingLocator()),
  new FallbackLocator(new DistinctAddressByTownLocator(),
    new DistinctAddressByCountryLocator()));
```

Koda 7.1: Primer bolj zapletene kombinacije iskalcev in multi iskalcev.

Vsakemu iskalcu lahko nastavimo **največje število zadetkov** (*maxResults*), ki jih vrne. V primeru sestavljenega iskalca (*CompositeLocator*) se iskanje prekine, ko je skupaj najdeno dovolj zadetkov. Drugače pa velja največje število zadetkov posameznega iskalca, če ga nastavimo. V kolikor največjega števila zadetkov za posameznega iskalca ne nastavimo, velja zanj število, ki smo ga nastavili za multi iskalca.

```
Locator locator=new AddressLocator();
locator.setSearchString("cank ul. 3");
locator.setMaxResults(10);
List<? extends Place> places=this.geocodingService.find(locator);
```

Koda 7.2: Primer uporabe iskalca AddressLocator.

Na strežniški strani vsak iskalec naredi svoje delo. Vsak osnovni iskalec poišče zadetke neposredno v podatkovni zbirki. Multi iskalec samo uporabi osnovne iskalce, kar pomeni, da se poizvedba v zbirki požene večkrat, za vsakega osnovnega iskalca posebej.

7.2 Algoritem

Vsak osnovni iskalec najprej primerno **razčleni** (angl. parse) **iskalni niz**. Odstrani vse nečrkovne (angl. non-alphabetic) in neštevilčne (angl. non-digit) znake in jih razkosa v več podnizov.

```
final String regex="[\\s\\p{Punct}]+";
```

Koda 7.3: Regularni izraz (angl. regular expression, regex), ki predstavlja vse nečrkovne in neštevilčne znake. Z njim se iskalni niz razkosa na čiste besede.

Vsak podniz se nato **poišče kot del geoimena** v podatkovni zbirki:

```
geoname.name like '%podniz%'
```

Koda 7.4: Izrezek v SQL-ju, kako se iščejo podnizi v delu geoimena v podatkovni zbirki.

Manjša izjema je *AddressLocator*, ki išče tudi po hišnih številkah, zato pred iskanjem preveri še, kateri podnizi so podobni hišnim številkam (se začnejo s cifro):

```
final String houseNumberRegex="[0-9].*";
```

Koda 7.5. Regularni izraz, ki predstavlja besede, ki se začnejo s cifro - hišne številke.

AddressLocator poleg imena ulic išče še po hišnih številkah:

```
geoname.name like '%ime%' and geoname.type='ADDRESS'
and address.houseNumber like 'hisnaStevilka%'
```

Koda 7.6: Izrezek v SQL-ju, kako se iščejo naslovi v podatkovni zbirki.

Vsak iskalec išče po vseh podnizih tako, da mora geoime vsebovati vse podnize (lahko v različnem vrstnem redu), sicer ni zadetek. Izjema je *AddressLocator*, ki obravnava hišno številko ločeno.

Poizvedbe so definirane na enem mestu, v vmesniku DAO (*FindingDao*). Implementacija ima generator poizvedbe, ki podnize z operatorjem *IN* združi.

```
select s
```

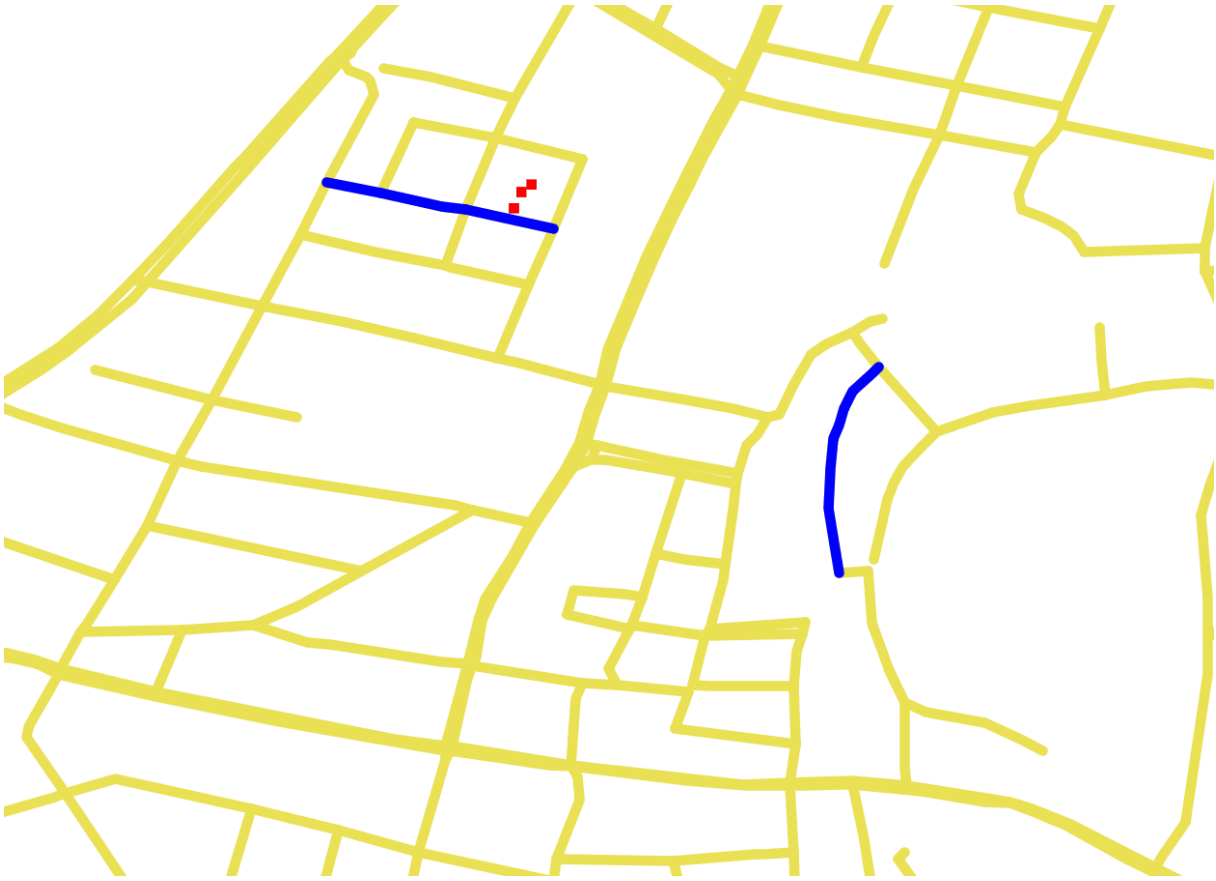
```
from Street as s
join s.geoNames as geoname
where (lower(geoname.name) like '%cank%'
  and lower(geoname.name) like '%ul%')
  and geoname.type='STREET'
```

Koda 7.7: Primer poizvedbe HQL, ki bi se tvorila z iskalcem StreetLocator in iskalnim nizom "Cank ul.".

Pri uporabi iskalca *DistinctStreetByTownLocator* z istim iskalnim nizom, bi bila poizvedba rahlo bolj zapletena in bi vsebovala podpoizvedbo (angl. subselect query), saj ta iskalec vrne le en (prvi) zadetek na mesto:

```
from Street as s
where s in (
  select min(s)
  from Street as s
  join s.geoNames as geoname
  where (lower(geoname.name) like '%cank%' and lower(geoname.name) like
    '%ul%')
  and geoname.type='TOWN'
  group by geoname.name
)
```

Koda 7.8: Primer poizvedbe HQL, ki bi se tvorila z iskalcem DistinctStreetByTownLocator in iskalnim nizom "Cank ul.".



Slika 7.2: Rezultat geokodiranja v središču Ljubljane. Rdeči kvadratki so naslovi, ki jih je našel AddressLocator z iskalnim nizom "cank 10" (Cankarjeva cesta 10, 10a in 10b). Modri črti predstavljata cesti, ki ju je našel StreetLocator z iskalnim nizom "cank" (Cankarjeva cesta in Cankarjevo nabrežje).

Storitev nima vgrajene logike neposrednega prepoznavanja vrste iskanega podatka (ali uporabnik išče naslov, cesto ali zgradbo), to je prepuščeno odjemalcu. Ta lahko z dodatno logiko ugotovi, kaj uporabnik želi, in uporabi ustrezne iskalce. Ali pa uporabniku kar ponudi, da si sam izbere iskalca. Lahko pa uporabi kakšnega od prednastavljenih multi iskalcev, ali sestavi svojega, in s tem že kar dobro izboljša natančnost iskanja.

Da bi bilo iskanje popolno, si prizadeva ekipa, ki izdeluje *PAGC* (Postal Address Geo-Coder). Pomembno pri tem izdelku je, da zna najprej standardizirati iskalni niz, to je ugotoviti, katero vrsto podatka uporabnik išče. Pri tem uporabljajo posebne zbirke podatkov: leksikone in pravila. Nato preiščejo indeksirano podatkovno zbirko po točnih zadetkih in po fonetičnih zadetkih (algoritem soundex), hkrati pa vsak zadetek ovrednotijo.

V tem sistemu je standardizacija iskalnega niza na nek način izvedena s številnimi osnovnimi in multi iskalci. Dodatno vrednost bi iskalci imeli, če bi znali zadetke ovrednotiti in jih sortirati po ustreznosti. Oboje bi pridobili, če bi uporabili eno od orodij za **celobesedilno iskanje**.

7.3 Celobesedilno iskanje

Pri geokodiranju se srečamo z iskanjem in primerjanjem nizov oz. podnizov. Uporabnik posreduje nek niz (naslov ali le nek del naslova, imena ceste, zgradbe ipd.), storitev pa preišče vsa geoimena v podatkovni zbirki in vrne zadetke.

Pri naprednejšem iskanju bi to delo prepustili orodjem, ki omogočajo celobesedilno iskanje (angl. full text search).

Takšna orodja so v bistvu pravi iskalni stroji (angl. search engine), kot jih poznamo iz drugih uspešnih aplikacij, iz, npr., spletnih iskalnikov. Ti iskalni stroji poznajo celo vrsto tako imenovanih **analizatorjev in filtrov**, ki znajo, npr., iskati po korenu besede, predvidijo celo napačen vnos uporabnika (vsebujejo črkovalnik), iščejo nize, ki se izgovarjajo podobno (fonetični algoritmi), in ovrednotijo ustreznost zadetka po primernosti (angl. relevance).

Preizkusil sem celobesedilno iskanje z ogrodjem **Compass**, ki uporablja zelo priljubljen iskalni stroj **Apache Lucene**. V sistemu ga na koncu nisem uporabil.

Nastavitev je razmeroma enostavna (primer je na priloženi CD plošči v paketu tinel-gis-geocoding-compass), saj se Compass lepo zlije s Springom in Hibernateom.

Podobno kot pri tehnologiji ORM, Compass potrebuje preslikavo (angl. mapping), ki opisuje, kako in kateri podatki bodo na voljo za iskanje. To lahko storimo preprosto z dodatnimi Compassovimi oznakami (angl. annotations) neposredno na domenskih objektih ali pa ločeno v preslikovalni datoteki XML. Oboje je na las podobno preslikovanju ORM za Hibernate.

Nazadnje definiramo še nova Springova zrna, ki tvorijo Compassov iskalnik.

Celobesedilno iskanje je tako pripravljeno. Vse, kar moramo še storiti, je, da poženemo prvo indeksiranje in napišemo nekaj iskalnih poizvedb.

Iskalni stroj temelji na dobro premišljenih indeksih, ki jih privzeto hrani v posebnih datotekah na trdem disku. Zaradi tega je zelo podoben sami podatkovni zbirki.

Ob indeksiranju Compass samodejno iz podatkovne zbirke pridobi ustrezne podatke in jih indeksira.

Poizvedbe so minimalistične. Pri iskanju navedemo ime atributa in iskalni niz. Iskalni stroj pozna tudi preprosto in obširno sintakso, s katero lahko kar v poizvedbi vplivamo na iskanje in ovrednotenje zadetkov.

```
String query="GeoName.name:ljub* AND GeoName.type=TOWN";
```

Koda 7.9: Primer poizvedbe pri celobesedilnem iskanju, ki poišče vsa geoimena mest, ki se pričnejo z nizom "ljub".

Ob začetnem navdušenju, da iskanje pravzaprav deluje, sem kasneje prišel do ugotovitve, da za naše potrebe **celobesedilno iskanje ne prinaša dovolj nove vrednosti**:

1. Celobesedilno iskanje se da v celoti nadomestiti samo z bolj zapletenimi poizvedbami po podatkovni zbirki.

2. Celobesedilno iskanje je včasih tudi do dvakrat hitrejše kot poizvedovanje po podatkovni zbirki, vendar vrne le **delne rezultate**. Npr., ob iskanju naslovov ne vrne lokacije, ker ni indeksirana. Po iskanju je tako treba rezultate dopolniti še iz zbirke, kar predstavlja dodatno poizvedbo. Hitrost je v tem primeru nižja.
3. Indeks je treba **posebej vzdrževati** (ob vsaki spremembi v podatkovni zbirki je treba indeks na novo zgraditi).
4. Iskanje po naslovih je povzročalo **težave**, vsaj pri uporabi ogrodja Compass. Naslovi so sestavljeni iz več geoimen, vsak naslov vsebuje tudi eno ime mesta (geoime vrste TOWN). Če iščemo naslove samo po določenem imenu mesta, potem logičen iskalni niz "name:ljubljana AND type:TOWN" ne daje pričakovanih rezultatov. Namesto da bi vrnil vse naslove, ki so v mestu Ljubljana, vrne kar vse naslove, ki vsebujejo geoime vrste TOWN - to so vsi.

Celobesedilno iskanje zadetke **rangira**. V tem sistemu je to zelo omejeno, saj dobro deluje le pri iskanju geoimen, pa še tedaj je rangiranje treba vnaprej določiti s tako imenovano ojačevalno funkcijo (angl. boost). Sama geoimena pa kot zadetki ne prinašajo neke uporabne vrednosti, saj ne vsebujejo prostorskega podatka.

Glede na vse skupaj, sem se odločil opustiti celobesedilno iskanje. Klasično iskanje s pomočjo poizvedb v zbirki skupaj s poljubnimi iskalci trenutno povsem zadostuje.

Edini zelo koristen primer uporabe celobesedilnega iskanja, ki si ga lahko zamislim, je podoben tehnologiji Google Suggest [25] in sorodnim:

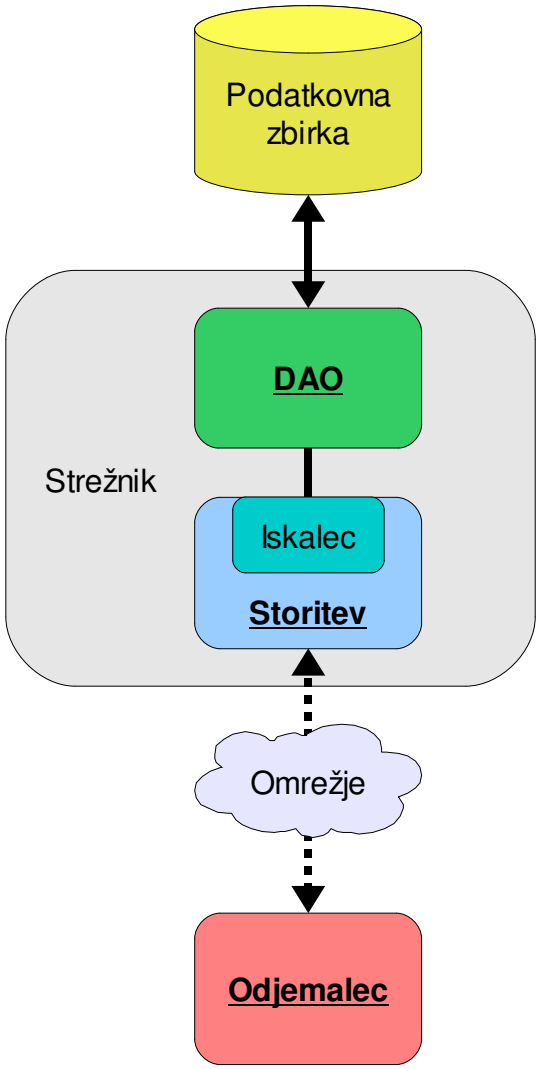
Uporabnik prične v odjemalcu z grafičnim uporabniškim vmesnikom v vnosno polje vpisovati iskalni niz. Ko vpiše, recimo, vsaj tri znake, se sproži celobesedilno iskanje nad geoimeni. Iskanje se dogaja v ozadju, uporabniku pa se zadetki nemudoma izpišejo v spustnem seznamu kot predlogi. Ko uporabnik izbere predlog, se sproži običajna poizvedba v podatkovni zbirki glede na vrsto izbranega geoimena, zadetki pa se narišejo na zemljevid.

7.4 Zmogljivost

Na hitrost poizvedbe v največji meri vpliva število podatkov v podatkovni zbirki. Razlike med enim in drugim iskalcem so predvsem razlike v tem, nad kakšno množico podatkov se iskanje izvaja, poizvedba je namreč pri vseh zelo podobna. Zato kakšne zanimive primerjave ne morem narediti.

Pri testiranju zmogljivosti storitve sem izvedel *trinivojsko* meritev, ki je pokazala hitrostne razlike med **tremi različnimi nivoji** v sami arhitekturi storitve:

1. **DAO** - Neposreden klic metode, ki pripravi poizvedbo in jo pošlje podatkovni zbirki za obdelavo.
2. **Storitev** - Klic storitve z iskalcem. Iskalec pripravi posredovane iskalne podatke in pokliče DAO. Pripravi rezultate.
3. **Odjemalec** - Klic metode storitve z iskalcem prek HTTP Invokerja na zagnan strežnik storitve.



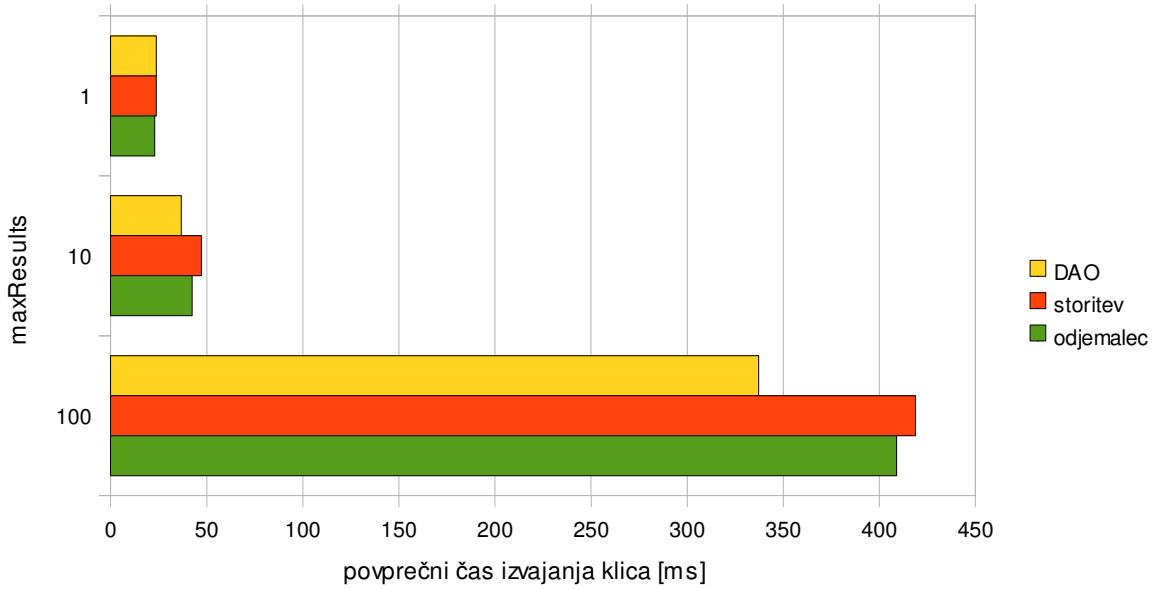
Slika 7.3: Trije nivoji v arhitekturi storitve (DAO, storitev in odjemalec) pomembni pri trinivojski meritvi zmogljivosti.

Vse tri nivoje sem testiral na lokalnem računalniku (testi, strežnik in zbirka so tekli lokalno). Parameter, ki sem ga spreminjal, je največje število zadetkov (*maxResults*), saj ta najbolj vpliva na končno hitrost. Vsak klic sem ponovil stokrat z enim zagonom okolja, vsako meritev pa trikrat.

Iskal sem naslove. Pri meritvi na nivoju DAO je bil iskalni niz "Cank" ter hišna številka "1". Na nivojih storitve in odjemalca je bil iskalni niz "Cank 1" in iskalec *AddressLocator*.

maxResults	Nivo	Povprečni čas izvajanja klica
1	DAO	23,70 ms
	storitev	23,80 ms
	odjemalec	22,97 ms
10	DAO	36,67 ms
	storitev	47,30 ms
	odjemalec	42,34 ms
100	DAO	337,19 ms
	storitev	418,78 ms
	odjemalec	408,90 ms

Tabela 7.1: Rezultati trinivojske meritve zmogljivosti. Manj je bolje.



Graf 7.1: Rezultati trinivojske meritve zmogljivosti. Manj je bolje.

Rezultati kažejo na **hitrost celotne infrastrukture**. Počasnejše izvajanje pri večjem številu zadetkov (*maxResults*) gre pripisati prenašanju (angl. fetch) rezultatov med zbirko in programom. Storitev in odjemalec sta pričakovano nekoliko počasnejša od DAO, kar je posledica dodatnih funkcij, ki jih opravita (uporaba iskalca, pripravljanje rezultatov za odjemalca). Zanimiv pa je ravno odjemalec, saj je malenkost hitrejši od storitve. To je posledica tega, da je strežnik skozi vse meritve tekel in si je tako že napolnil predpomnilnik (angl. cache). Komunikacija med strežnikom in odjemalcem (protokol HTTP) pa očitno ne vpliva toliko, res pa je, da sta bila oba, strežnik in odjemalec, na istem računalniku, tekla sta lokalno.

8 Nasprotno geokodiranje

Nasprotno geokodiranje (angl. reverse geocoding) je storitev, ki poišče geografske kraje, ki se nahajajo v neposredni bližini določene lokacije. Prav pride uporabnikom, ki na nek način pridobijo koordinate neke lokacije (npr. s klikom na karti, iz sprejemnika GPS ipd.) in bi radi izvedeli, kateri je **najbližji geografski kraj** (naslov, cesta, zgradba ipd.). Na primer, lastniku vozila se namesto geografskih koordinat v ustreznem grafičnem odjemalcu prikaže najbližji naslov, kjer stoji njegovo vozilo. Uporabniki si samo s koordinatami le stežka predstavljajo pravo lokacijo.

Nasprotno geokodiranje je, zanimivo, kljub podobnemu imenu tehnično precej drugače izvedeno od geokodiranja.

Ustrezna koda se nahaja v javinem paketu net.tinelstudio.gis.reversegeocoding.

8.1 Iskalci

Storitev nasprotnega geokodiranja je zasnovana tako, da odjemalec strežniku poda zahtevo v obliki *iskalca* (angl. locator). Iskalci imajo zelo podobno vlogo kot iskalci pri storitvi geokodiranja. Uporabnik v odjemalcu nastavi iskalca in ga kot parameter priloži metodi, ko kliče storitev nasprotnega geokodiranja.

Vsak iskalec ima tri parametre, ki jih je potrebno nastaviti:

- *searchLocation* - Geografski koordinati **iskalne lokacije**, podani v stopinjah, v obliki zemljepisne dolžine in širine. Npr. za Ljubljanski grad sta (14.508, 46.049).
- *maxDistanceMeters* - **Iskalni radij**, največja razdalja v metrih med iskalno lokacijo in krajem, ki še pride v poštev. S tem podatkom se omejimo samo na določeno območje okoli iskalne lokacije. Če je radij premajhen oz. v okolici iskalne lokacije ni kraja, iskalec vrne prazen seznam zadetkov. Iskalni radij je po navadi razmeroma majhen, govorimo o številki od nekaj sto metrov do nekaj kilometrov.
- *maxResults* - **Največje število zadetkov**. Če je 1, potem iskalec vrne le najbližji zadetek. Koristna omejitev, če je iskalni radij velik in je v okolici iskalne lokacije ogromno krajev (npr. v mestih).

Z izbiro iskalca uporabnik pove, katere kraje naj storitev išče.

Storitev pozna tri vrste iskalcev:

1. Osnovni iskalec, ki išče samo po **eni vrsti krajev**.

   - **AddressLocator** - Išče samo med naslovi (domena *Address*).
   - **StreetLocator** - Išče samo med cestami (domena *Street*).
   - **BuildingLocator** - Išče samo med zgradbami (domena *Building*).

2. Multi iskalec, ki **kombinira več iskalcev**.
   Iskalec najprej delegira iskanje prvemu podanemu iskalcu, nato drugemu, nato tretjemu in tako dalje. Zadetke vseh iskalcev vrne v enem seznamu po vrsti tako, kot so bili najdeni.

   - **CompositeLocator** - Delegira iskanje po vseh navedenih iskalcih.
   - **FullCompositeLocator** - Predpripravljen iskalec, ki delegira iskanje po vseh osnovnih iskalcih.

3. Multi iskalec, ki preide na naslednjega iskalca, če prejšnji ne vrne zadetkov.
   Iskalec najprej delegira iskanje prvemu podanemu iskalcu. Če ta ne vrne nobenega zadetka, se iskanje delegira drugemu, drugače se takoj vrnejo zadetki samo prvega iskalca. Ostali sploh ne pridejo na vrsto.

   - **FallbackLocator** - Delegira iskanje po navedenih iskalcih, dokler eden ne vrne zadetkov.
   - **DefaultFallbackLocator** - Predpripravljen iskalec, ki smiselno delegira iskanje po osnovnih iskalcih, dokler eden ne vrne zadetkov.

Z multi iskalci je možna zelo zapletena in fino nastavljiva **kombinacija vseh iskalcev**, tudi več multi iskalcev, npr.:

```
Locator locator=new FallbackLocator(new CompositeLocator(
  new AddressLocator(), new BuildingLocator()), new StreetLocator());
```

Koda 8.1: Primer bolj zapletene kombinacije iskalcev in multi iskalcev.

Vsakemu iskalcu lahko nastavimo **največje število zadetkov** (*maxResults*), ki jih vrne. V primeru sestavljenega iskalca (*CompositeLocator*) se iskanje prekine, ko je skupaj najdeno dovolj zadetkov. Drugače pa velja največje število zadetkov posameznega iskalca, če ga nastavimo. V kolikor največjega števila zadetkov za posameznega iskalca ne nastavimo, velja zanj število, ki smo ga nastavili za multi iskalca.

```
Locator locator=new AddressLocator();
locator.setSearchLocation(new Coordinate(15.654, 46.548));
locator.setMaxDistanceMeters(500);
locator.setMaxResults(10);
List<? extends Place> places=this.reverseGeocodingService
```

```
  .findNearest(locator);
```

Koda 8.2: Primer uporabe iskalca AddressLocator.

Na strežniški strani vsak iskalec naredi svoje delo. Vsak osnovni iskalec poišče zadetke neposredno v podatkovni zbirki. Multi iskalec samo uporabi osnovne iskalce, kar pomeni, da se poizvedba v zbirki požene večkrat, za vsakega osnovnega iskalca posebej.

8.2 Algoritem

Vsak osnovni iskalec **vrne seznam najdenih krajev**, ki ustrezajo kriterijem, sortiranih po oddaljenosti od iskalne lokacije tako, da ji je **prvi kraj najbližje**.

Kriteriji, ki so bistveni za algoritem, so trije:

- vrsta iskalca pove, v kateri(h) domeni(ah) se bo iskalo,
- parameter *searchLocation* pove iskalno lokacijo, okoli katere se bo iskalo in
- parameter *maxDistanceMeters* pove največjo oddaljenost od iskalne lokacije, okoli katere se bo iskalo in sortiralo zadetke.

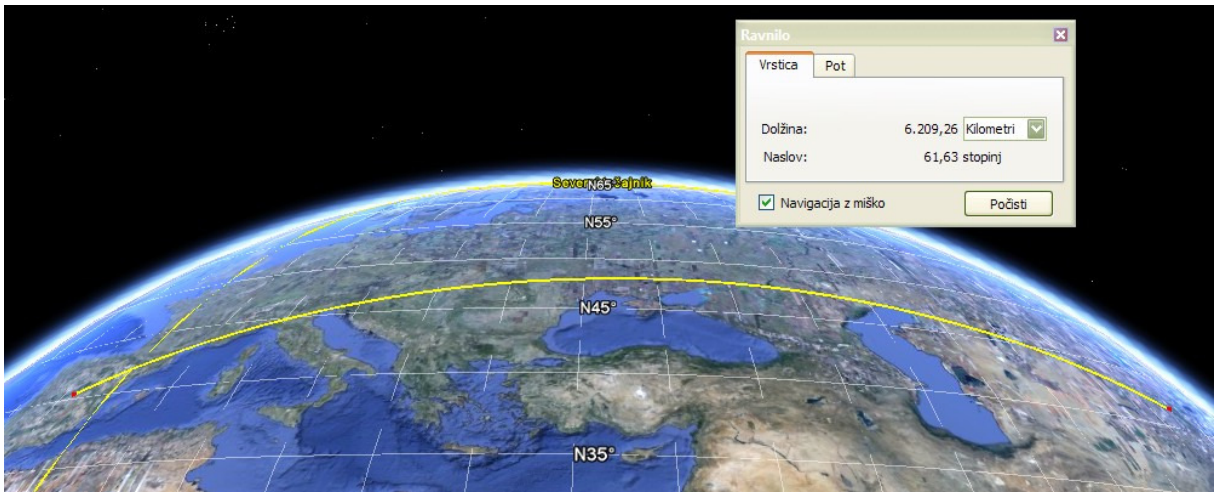
Zadovoljiti vsem tem kriterijem skupaj z največjo hitrostjo in natančnostjo je žal **nemogoče**. Poglejmo, zakaj.

Geografski kraji so prostorski podatki sestavljeni iz točk (angl. point), ki so podane z geografskimi koordinatami v stopinjah. Prva koordinata je zemljepisna dolžina (angl. longitude). Druga je zemljepisna širina (angl. latitude). Pri iskanju najbližjih točk se najprej omejimo na neko **največjo razdaljo med iskalno lokacijo in točkami** (iskalni radij) in nato zadetke **sortiramo glede na oddaljenost** od iskalne lokacije.

Točke so izražene v (kotnih) stopinjah, iskalni radij v metrih. Ti enoti med seboj **nista primerljivi**.

Koordinate označujejo točno lokacijo na površini *objekta*, v našem primeru na površini Zemlje. Pri tem je ključna **oblika** tega objekta. Površina Zemlje je podobna sferi s fiksnim polmerom. Ampak sfera je le, sicer dokaj dober, približek, saj je Zemlja na polih rahlo sploščena, zato je v uporabi vrsta natančnejših elipsoidov. Eden najbolj uporabljanih je WGS 84, saj se uporablja pri tehnologiji GPS. Obliki objekta z referenčnimi točkami strokovno pravimo datum (angl. datum).

Najkrajšo razdaljo (angl. great-circle distance) med dvema lokacijama lahko izračunamo na dva načina. Pri prvem načinu razdalje računamo **neposredno na modelu sfere ali elipsoida**. Za računanje razdalj na sferi uporabimo nekaj kotnih funkcij, potrebujemo pa le podatek o polmeru sfere (Zemlje). Ker površina Zemlje ni sfera, je polmer le dogovorjen približek (npr. 6.371.007 m [19]). Če želimo nekoliko večjo natančnost, lahko razdalje računamo neposredno na elipsoidu. V poštev pridejo nekoliko težje matematične formule (v geodeziji se uporabljajo Vincentyjeve formule [11, 15]). Obe metodi sta natančni in uporabni za računanje razdalj kjerkoli na Zemlji.

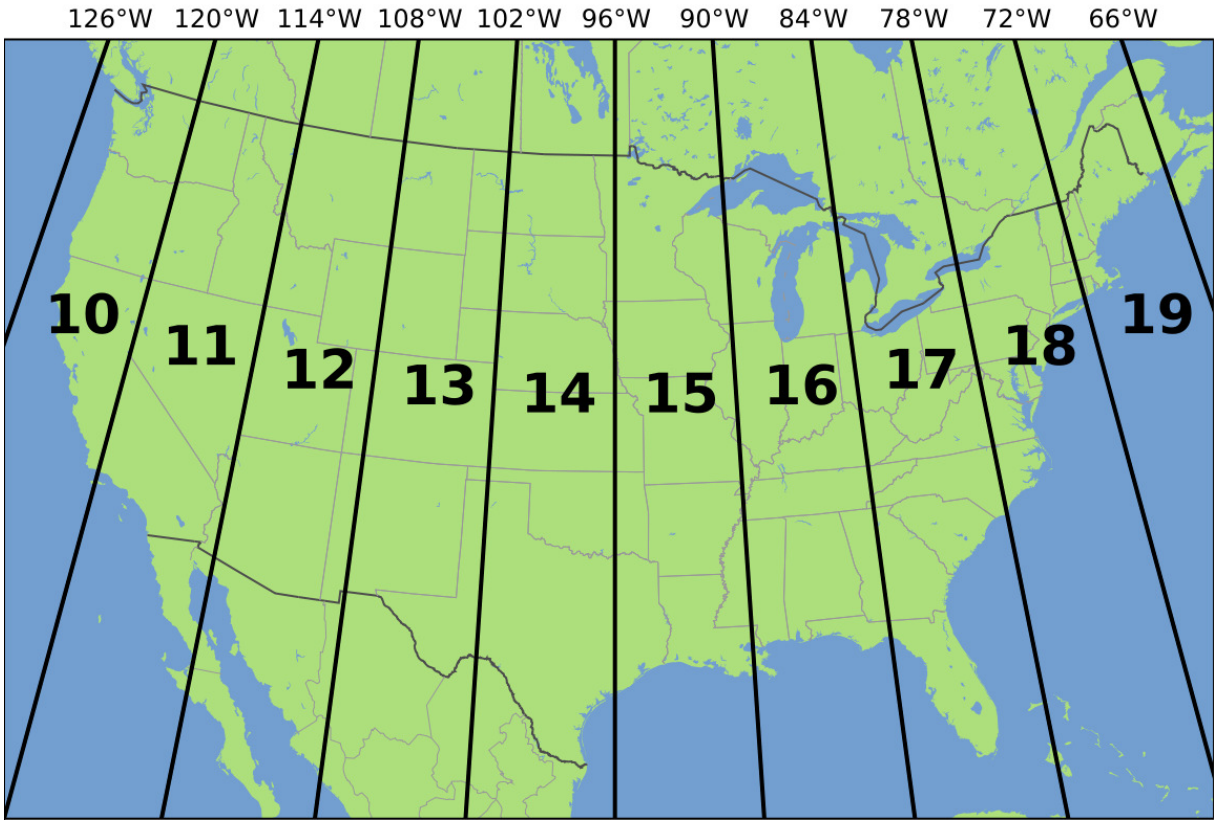


Slika 8.1: Primer najkrajše razdalje med dvema lokacijama na Zemlji. Z rumeno upognjeno črto je označena najkrajša razdalja (6209 km), ki gre po površini Zemlje in ne skozi njo.

Pri drugem načinu lokacijo pretvorimo v metrični (kartezični) koordinatni sistem. Lokacijo s tridimenzionalnega objekta preslikamo v dvodimenzionalno ravnino. To imenujemo **kartografska projekcija** (angl. map projection). Obstaja ogromno projekcij. Ločijo se po različnih načinih projiciranja (valjni, azimutni, stožčni), različnih datumih (sfera ali elipsoid) in različnih enotah (metri, stopinje, milje ipd.). Težava vseh so različna **popačenja**, različna popačenja pa narekujejo različno uporabo.

V našem primeru pridejo v poštev samo projekcije, pri katerih je razdalja v vse smeri pravilna. Ena takšnih je univerzalna prečna Merkatorjeva (angl. Universal Transverse Mercator - UTM) projekcija, ki je pravzaprav sistem več projekcij. Te so razdeljene v posamezne cone v mreži UTM, ki jo sestavlja 60 con za severno poloblo in 60 za južno. Vsaka zajema pas 6° zemljepisne dolžine. Projekcija v vsaki coni je zelo natančna, saj so popačenja na robovih le približno 1 % [1, 10]. Skupaj z univerzalnim polarnim stereografskim (angl. Universal Polar Stereographic - UPS) sistemom projekcij, ki zajema oba pola, imamo Zemljo v celoti pokrito in vsako lokacijo izraženo v metrih.

Težava nastane, kadar želimo izračunati razdaljo med dvema lokacijama, ki padeta v **različne cone**. Vsaka cona ima namreč drugo referenčno točko in lokacije med seboj **niso primerljive**. Lahko bi sicer uporabili eno cono za obe točki, a bi bile napake velike. Če bi uporabili projekcijo, ki pokriva večino Zemlje, npr. svetovno Merkatorjevo (angl. World Mercator), pa bi bile napake nesprejemljivo velike že na zemljepisni širini Slovenije, kaj šele bližje poloma. V kolikor bi se omejili na računanje razdalj samo po Sloveniji, bi bila projekcija UTM za cono 33 N odlična izbira, a kakor bi želeli pokukati izven Slovenije, po Evropi, bi naleteli na več kot 10 con.



Slika 8.2: Cone univerzalnega Merkatorjevega sistema (UTM) na območju ZDA. Razdalje med lokacijami znotraj ene cone so zelo natančne.

Pomembna lastnost sistema je **hitrost iskanja krajev**. Podatkovna zbirka PostgreSQL ima posebno vrsto indeksa primerno za prostorske podatke - GiST. Ker so kraji hranjeni v obliki točk v stopinjah, ki so indeksirane, to pomeni, da se pri iskanju **uporabljajo indeksi** samo takrat, ko iščemo v isti enoti, stopinjah. V našem primeru pa vedno iščemo v metrih. Torej, pri metodah za neposredno računanje na sferi ali elipsoidu se indeksi **ne uporabljajo**, oddaljenost iskalne lokacije se računa z vsakim krajem, kar pa je **počasno**.

Pri drugem načinu, projekciji, obstajata dve možnosti. Lahko sproti **projiciramo kraje**, a pri tem se indeksi ne uporabljajo, saj je treba vsak kraj najprej projicirati, šele nato se izračuna razdalja. Druga možnost je, da bi krajem dodali še en atribut v zbirki, in sicer iste točke, vendar v drugem, že projiciranem metričnem koordinatnem sistemu. Te točke bi indeksirali, iskanje bi potekalo neposredno po indeksih. A težava nastane pri izbiri tega metričnega sistema. Če bi izbrali edini dovolj natančen sistem UTM skupaj z UPS, potem vse točke ne bi bile v isti coni in ne bi bile primerljive med seboj.

Za lažjo primerjavo in preglednost sem nekaj možnih rešitev podal v tabelo 8.1:

Metoda	Opis	Natančnost	Hitrost
A	sprotno projiciranje kraja v cono UTM/UPS od iskalne lokacije	velike napake v primeru, ko je kraj precej oddaljen od iskalne cone; velika natančnost, če je kraj v iskalni coni	nizka hitrost: indeksi se ne uporabljajo, ker se kraji sproti projicirajo
B	sprotno projiciranje kraja v svetovno Merkatorjevo projekcijo	velike napake na zemljepisni širini proti poloma; pola nista podprta	nizka hitrost: indeksi se ne uporabljajo, ker se kraji sproti projicirajo
C	predprojicirane dodatne točke v svetovni Merkatorjevi projekciji	velike napake na zemljepisni širini proti poloma; pola nista podprta	**visoka** hitrost: neposredna uporaba indeksov
Č	računanje neposredno v stopinjah	razdalje v stopinjah niso primerljive z metri: natančnost ob ekvatorju, velike napake proti poloma	**visoka** hitrost: neposredna uporaba indeksov
D	računanje neposredno na sferi ali elipsoidu	**velika** natančnost	nizka hitrost: indeksi se ne uporabljajo, ker se razdalje sproti računajo

Tabela 8.1: Metode za računanje najkrajše razdalje med iskalno lokacijo in geografskimi kraji.

Algoritem temelji na **metodi D**, saj je edina brezpogojno natančna za cel svet.

PostGIS ima za ta namen pripravljeni dve funkciji [24]: *ST_distance_sphere* in *ST_distance_spheroid*. Obe vračata najkrajšo razdaljo v metrih. Prva računa neposredno na sferi s fiksnim polmerom 6.370.986 m, pri drugi pa sami izberemo elipsoid. Druga je natančnejša in zato počasnejša. Obe pa znata računati najkrajšo razdaljo **samo med dvema točkama**, ne pa tudi med linijami ali poligoni.

Osnovna poizvedba (**varianta 1**), ki najde najbližje naslove (naslovi so točke) od iskalne lokacije je naslednja:

```
from Address as a
where ST_distance_sphere(a.point, :point) <= :distance
order by ST_distance_sphere(a.point, :point)
```

Koda 8.3: Poizvedba HQL, varianta 1: iskanje najbližjih naslovov, računanje razdalj v metrih na Zemlji (sferi). ":point" je iskalna lokacija, ":distance" je iskalni radij v metrih.

Poizvedba kliče funkcijo *ST_distance_sphere*, ki ne uporablja indeksov, zato je razmeroma **počasna**.

Zgornjo poizvedbo lahko s trikom precej pohitrimo. Ta najprej iskanje omeji na določeno območje - iskalni pravokotnik (angl. bounding box), nato pa računa razdalje samo še na tem delu (**varianta 2**):

```
from Address as a
where ST_Intersects(a.point, :bbox) = true
  and ST_distance_sphere(a.point, :point) <= :distance
order by ST_distance_sphere(a.point, :point)
```

Koda 8.4: Poizvedba HQL, varianta 2: iskanje najbližjih naslovov, računanje razdalj v metrih na Zemlji (sferi) z dodatnim iskalnim pravokotnikom. ":point" je iskalna lokacija, ":distance" je iskalni radij v metrih, ":bbox" je iskalni pravokotnik v stopinjah.

Pred tem je treba iskalni pravokotnik še izračunati. To storimo tako, da najprej iskalno lokacijo povečamo v smeri zemljepisne dolžine za iskalni radij in nato še v smeri zemljepisne širine. Iskalni radij je podan v metrih, zato si pri pretvorbi pomagamo s kotnimi funkcijami: iz razdalje v metrih izračunamo razdaljo v stopinjah. Ta je seveda različna za različne lokacije na Zemlji.

Takšen iskalni pravokotnik v funkciji PostGIS *ST_Intersects* ignorira vse točke (naslove), ki niso v njem oz. se ga ne dotikajo. Funkcija *ST_Intersects* uporablja indekse, saj se izvaja neposredno na točkah, ki so v stopinjah.

Ostali del poizvedbe je enak prvi varianti, le da se izvaja računanje razdalj med točkami na bistveno manjši množici. Zato je druga varianta **bistveno hitrejša** v primerjavi s prvo, medtem ko se natančnost ne poslabša. Hitrost se začne zmanjševati, če povečujemo iskalni radij, saj tedaj *ST_Intersects* ignorira vse manj točk.

Kot zanimivost sem preizkusil še **tretjo varianto** po metodi A, ki je tudi dovolj natančna, kadar je iskalni radij normalno majhen (tako kot v praksi):

```
from Address as a
where ST_DWithin(ST_Transform(a.point, :utm),
  ST_Transform(:place, :utm), :distance) = true
order by ST_Distance(ST_Transform(a.point, :utm),
  ST_Transform(:place, :utm))
```

Koda 8.5: Poizvedba HQL, varianta 3: iskanje najbližjih naslovov, projekcija UTM. ":place" je iskalna lokacija, ":distance" je iskalni radij v metrih, ":utm" je številka SRID cone UTM.

Ta varianta obe točki, iskalno lokacijo in naslov, najprej projicira v coni UTM (funkcija PostGIS *ST_Transform*), v kateri je iskalna lokacija, in nato med njima izračuna razdaljo. Funkcija *ST_DWithin* ugotovi, ali sta točki preveč oddaljeni druga od druge.

Pri tej poizvedbi se indeksi nikjer ne uporabljajo, saj je potrebno najprej vsako točko (naslov) projicirati, šele nato se lahko izvede primerjava. Ta varianta je zato **počasna**, še nekoliko bolj kot prva.

Slika 8.3: Rezultat poizvedb variante 1, 2 in 3 je enak in natančen. Rdeči kvadratki so najdeni naslovi, iskalni radij je 500 m (tanka črna krožnica), v središču Ljubljane.

Za primerjavo hitrosti sem preizkusil še **četrto varianto** po metodi Č, ki pa ji natančnost že kar pošteno peša, a je najhitrejša:

```
from Address as a
where ST_Intersects(a.point, :bbox) = true
order by ST_Distance(a.point, :place)
```

Koda 8.6: Poizvedba HQL, varianta 4: iskanje najbližjih naslovov, primerjanje razdalj v stopinjah. ":place" je iskalna lokacija, ":bbox" je iskalni pravokotnik v stopinjah.

Ta, enako kot druga varianta, najprej izračuna iskalni pravokotnik (bbox) in pri obeh funkcijah *ST_Intersects* in *ST_Distance* uporablja indekse. Ker se primerjanje razdalj izvaja neposredno na točkah, ki so samo v stopinjah, je poizvedba bliskovito hitra, a nekoliko nenatančna, saj se razmerje med stopinjami in metri z zemljepisno širino spreminja. Poleg te napake vrne tudi vse točke, ki so v iskalnem pravokotniku in na vogalih niso omejeni z iskalnim radijem, kot je lepo vidno na sliki 8.4:

Slika 8.4: Rezultat poizvedbe variante 4 je nenatančen a najhitrejši. Rdeči kvadratki so najdeni naslovi, iskalni radij je 500 m (tanka črna krožnica), v središču Ljubljane. Nekateri najdeni naslovi presegajo iskalni radij.

Funkcija PostGIS *ST_distance_sphere* kot parametra vzame samo dve točki. Če želimo med seboj primerjati **oddaljenost linije** (npr. cesto) od točke, moramo najprej na liniji poiskati najbližjo točko iskalni točki. Z enako težavo se srečamo pri poligonu. Rešitev žal ni vgrajena v PostGIS, ampak jo z malce truda lahko sami razvijemo.

Poizvedba, ki poišče **najbližjo cesto** (linijo) iskalni točki, je naslednja:

```
from Street as s
where ST_Intersects(s.lineString, :bbox) = true
  and ST_distance_sphere(:point,
    ST_line_interpolate_point(s.lineString,
      ST_line_locate_point(s.lineString, :point))) <= :distance
order by ST_distance_sphere(:point,
  ST_line_interpolate_point(s.lineString,
```

```
    ST_line_locate_point(s.lineString, :point)))
```

Koda 8.7: Poizvedba HQL: iskanje najbližjih cest. ":point" je iskalna lokacija, ":distance" je iskalni radij v metrih, ":bbox" je iskalni pravokotnik v stopinjah.

Poizvedba je pravzaprav enaka tisti za naslove (varianta 2), le da je linija dodatno oplemenitena z dvema funkcijama PostGIS. Prva je *ST_line_locate_point*, ki vrne najbližjo točko (na liniji) med linijo in iskalno točko v obliki razdalje od začetne točke linije. Druga je *ST_line_interpolate_point*, ki pa točko na liniji dejansko izračuna iz podane razdalje. Tukaj je treba vedeti, da je točka na liniji interpolirana, ker je največkrat vmes, med definiranimi točkami, ki sicer opisujejo linijo (pravzaprav niz linij).

Ti dve funkciji se sicer zelo hitro izvajata, tako da na koncu celotna poizvedba ne odstopa veliko od tiste za iskanje naslovov.

Zadnja, ki jo storitev potrebuje, je poizvedba, ki poišče **najbližjo zgradbo** (poligon) iskalni točki:

```
from Building as b
where ST_Intersects(b.polygon, :bbox) = true
  and ST_distance_sphere(:point,
    ST_line_interpolate_point(ST_ExteriorRing(b.polygon),
      ST_line_locate_point(ST_ExteriorRing(b.polygon), :point)))
        <= :distance
order by ST_distance_sphere(:point,
  ST_line_interpolate_point(ST_ExteriorRing(b.polygon),
    ST_line_locate_point(ST_ExteriorRing(b.polygon), :point)))
```

Koda 8.8: Poizvedba HQL: iskanje najbližjih zgradb. ":point" je iskalna lokacija, ":distance" je iskalni radij v metrih, ":bbox" je iskalni pravokotnik v stopinjah.

Poizvedba je na las podobna tisti za iskanje cest, a najprej iz poligona s funkcijo *ST_ExteriorRing* pridobi linijo. To je linija (pravzaprav niz linij), ki opisuje zunanji del poligona. Poizvedba deluje tako, da poišče najbližji poligon samo glede na zunanji obod poligona, kar ni nič slabega. Zgradbe so ponavadi majhne glede na iskalni radij, če pa iskalna točka pade točno v notranjost poligona, bo vseeno ta poligon najbližji.

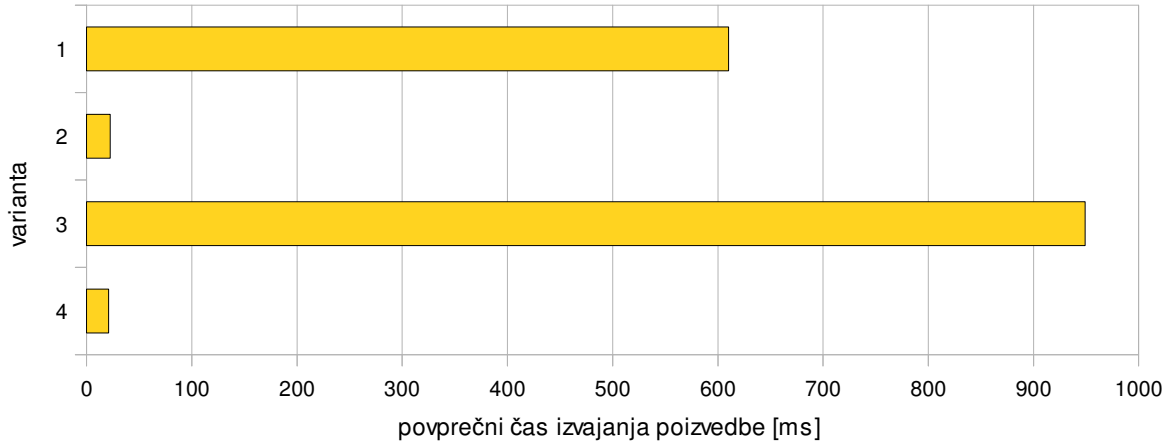
Slika 8.5: Rezultat nasprotnega geokodiranja v središču Ljubljane. Rdeči kvadratki so naslovi, modre črte so ceste in sivi liki so zgradbe, ki jih je našel sestavljeni iskalec FullCompositeLocator z iskalnim radijem 500 m (tanka črna krožnica).

8.3 Zmogljivost

Najprej bom prikazal rezultate meritev hitrosti vseh štirih variant poizvedb za naslove. Parametre sem nastavil tako kot v praksi: iskalni radij na 5 km, zanimalo pa me je prvih deset naslovov. Vsako poizvedbo sem ponovil stokrat z enim zagonom okolja, vsako meritev pa trikrat. Poizvedbe so se vršile v lokalno nameščeni podatkovni zbirki.

Varianta	Opis	Povprečni čas izvajanja poizvedbe	
1	razdalja v metrih na Zemlji	610,00 ms	64,3 %
2	razdalja v metrih na Zemlji omejena z iskalnim pravokotnikom	22,45 ms	2,4 %
3	projekcija UTM	949,22 ms	100,0 %
4	razdalja v stopinjah	20,78 ms	2,2 %

Tabela 8.2: Rezultati meritve hitrosti vseh štirih variant za iskanje najbližjih naslovov. Manj je bolje.



Graf 8.1: Rezultati meritve hitrosti vseh štirih variant za iskanje najbližjih naslovov. Manj je bolje.

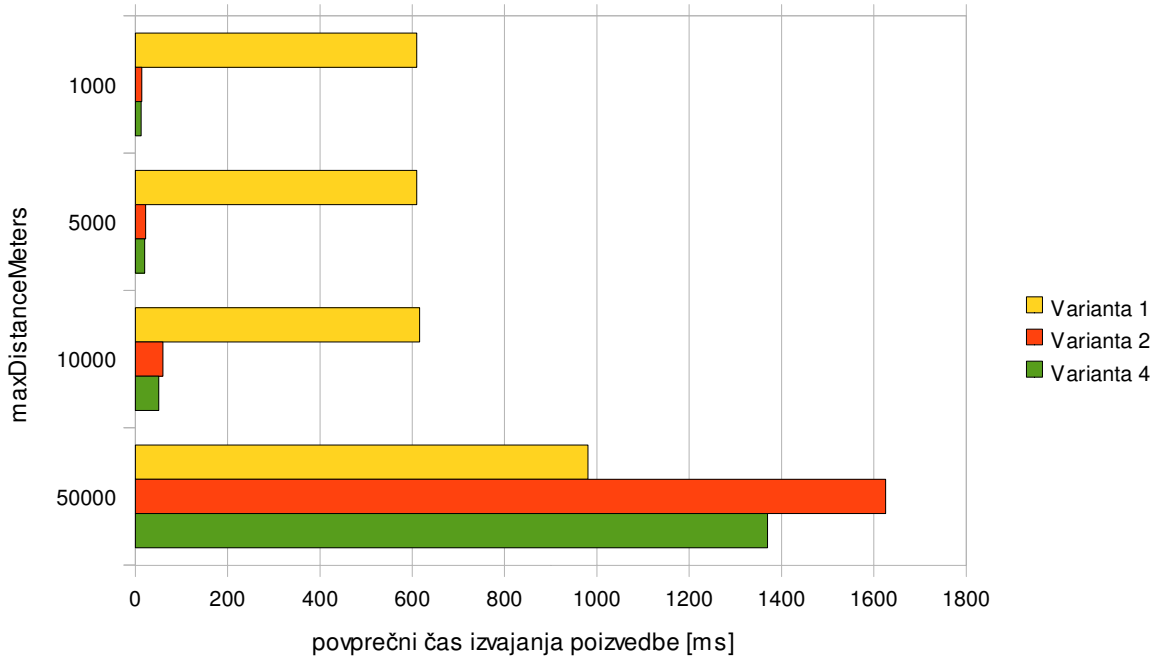
Takoj opazimo občutne razlike v hitrosti izvajanja poizvedb, saj je varianta 1 kar 27-krat počasnejša od variante 2. Meritve so tokrat nazorno pokazale, kakšna je razlika v poizvedbi, ki se vrši z indeksi ali brez: pri variantah 2 in 4 se uporabljajo indeksi. Zanemarljiva, a opazna je razlika med varianto 2 in 4 (v povprečju ~ 2 ms): to lahko pripišemo funkciji *ST_distance_sphere*, ki se uporablja samo pri varianti 2 in ne uporablja indeksov nad sicer zelo zmanjšano množico točk.

Za časovni pribitek pri varianti 3 je vsekakor kriva funkcija *ST_Transform*, ki iskalno lokacijo in naslov najprej projicira.

Sledi meritev hitrosti izvajanja poizvedbe pri različnih iskalnih radijih (*maxDistanceMeters*), vedno pa vrne 10 zadetkov. Meritve sem opravil na variantah 1, 2 in 4.

maxDistanceMeters	Varianta	Povprečni čas izvajanja poizvedbe
1000	1	609,53 ms
	2	14,01 ms
	4	12,97 ms
5000	1	610,00 ms
	2	22,45 ms
	4	20,78 ms
10000	1	615,94 ms
	2	59,89 ms
	4	50,63 ms
50000	1	980,89 ms
	2	1626,00 ms
	4	1369,74 ms

Tabela 8.3: Rezultati meritve hitrosti izvajanja poizvedbe za iskanje najbližjih naslovov pri različnih iskalnih radijih. Manj je bolje.



Graf 8.2: Rezultati meritve hitrosti izvajanja poizvedbe za iskanje najbližjih naslovov pri različnih iskalnih radijih. Manj je bolje.

Rezultati so rahlo **presenetljivi**.

Varianta 1 je najbolj predvidljiva, saj pri njej iskalni radij ne vpliva bistveno - vedno primerja vse naslove. Se pa vseeno malenkost hitrost zmanjša, kar lahko pripišemo sortiranju večje množice zadetkov. Sortiranje tako ali tako vpliva na hitrost pri vseh variantah.

Rezultati meritev pri varianti 2 so nepričakovani. Pričakoval sem zmanjšanje hitrosti z

večanjem iskalnega radija vse tja do hitrosti variante 1, a je hitrost bistveno bolj padla. Odgovor ponuja kar meritev pri varianti 4, ki kaže, da funkcija *ST_Intersects* potrebuje svoj čas. Očitno večji iskalni pravokotnik resno upočasni funkcijo, vsaj na področju, kjer je veliko zadetkov. Varianti 2 in 4 sem pognal še z iskalno lokacijo, ki je bila na področju, kjer ni nobenega naslova daleč naokoli. Obe sta potrebovali tako rekoč nič časa ne glede na iskalni radij, kar je v redu.

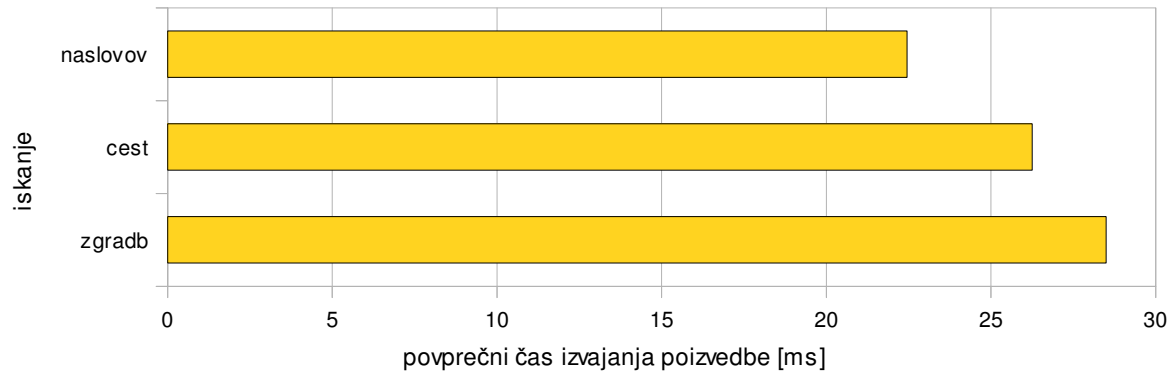
Zanimivo pri vsem tem je, da je varianta 2 počasnejša od variante 1 pri velikem iskalnem radiju, kljub temu, da je bolj napredna. A zanimiva je meja. S poskušanjem sem ugotovil, da je meja pri iskalnem radiju nekje 35 km. Če je iskalni radij manjši, je varianta 2 hitrejša, če je večji, je varianta 1 hitrejša. Naj še poudarim, da to velja samo pri določenem število naslovov, ki so v iskalnem radiju (v mojem primeru jih je pri iskalnem radiju 35 km kar 99.624). Če iščemo najbližje naslove tam, kjer jih ni, potem bo varianta 1 še vedno potrebovala svoj čas, medtem ko se bo varianta 1 takoj izvedla.

Pri snovanju poizvedbe, ki jo uporablja storitev nasprotnega geokodiranja (varianta 2), bi lahko **vgradili dodaten pogoj**, da se funkcija *ST_Intersects* ne uporabi pri velikem iskalnem radiju, a je meja zelo različna in je ne moremo vedeti vnaprej. Poleg tega je meja na področju, kjer je veliko naslovov, razmeroma velika, tako da se v praksi ne uporablja.

V nadaljevanju bom prikazal, kako na hitrost poizvedbe vplivajo dodatne funkcije, ki so potrebne pri iskanju cest in zgradb. Meritve sem izvajal na enak način kot prej.

Poizvedba	Povprečni čas izvajanja poizvedbe	
iskanje naslovov	22,45 ms	78,8 %
iskanje cest	26,25 ms	92,1 %
iskanje zgradb	28,49 ms	100,0 %

Tabela 8.4: Rezultati meritve hitrosti izvajanja poizvedb. Manj je bolje.



Graf 8.3: Rezultati meritve hitrosti izvajanja poizvedb. Manj je bolje.

Vse tri poizvedbe se izvajajo zelo hitro, za 4 ms in še za dodatne 2 ms počasnejši sta poizvedbi za iskanje cest in zgradb, za kar so krivi klici dodatnih funkcij, ki so potrebne pri iskanju najbližjega dela ceste oz. zgradbe iskalni lokaciji.

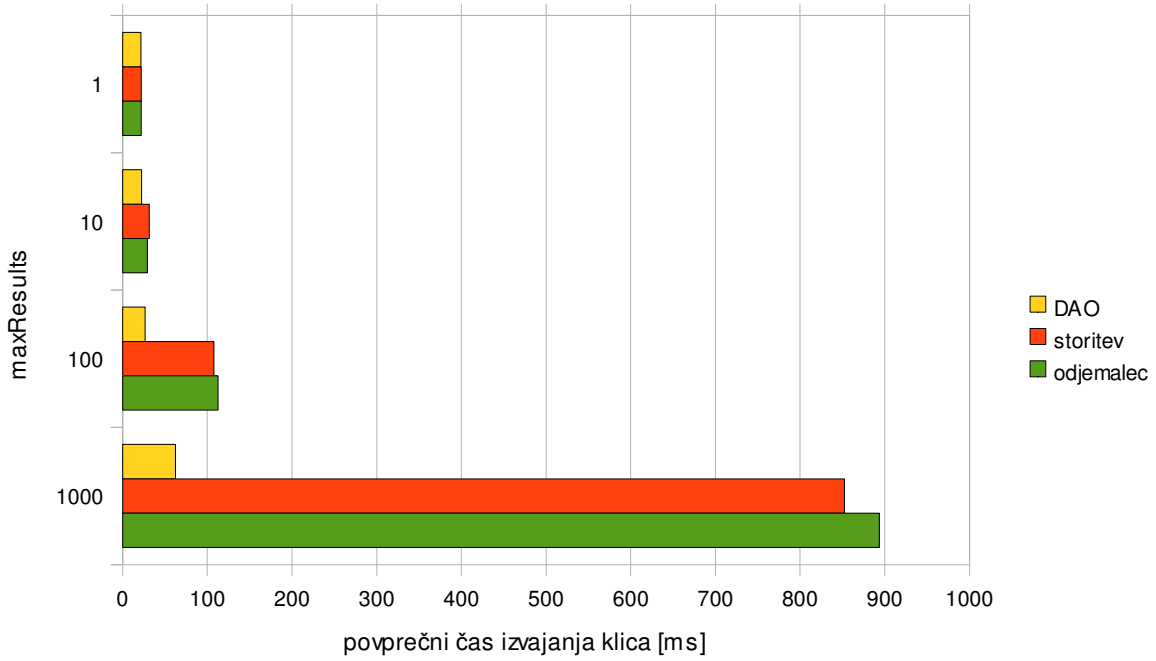
Pri testiranju zmogljivosti storitve sem, enako kot pri storitvi geokodiranja, izvedel *trinivojsko* meritev, ki je pokazala hitrostne razlike med tremi različnimi nivoji v sami arhitekturi storitve: **DAO, storitev, odjemalec**. Nivoji so nazorno prikazani na sliki 7.3.

Vse tri nivoje sem testiral na lokalnem računalniku. Parameter, ki sem ga spreminjal, je največje število zadetkov (*maxResults*). Vsak klic sem ponovil stokrat z enim zagonom okolja, vsako meritev pa trikrat.

Iskal sem naslove (varianta 2). Iskalni radij sem pustil na 5 km. Na nivojih storitve in odjemalca sem uporabil iskalca *AddressLocator*.

maxResults	Nivo	Povprečni čas izvajanja klica
1	DAO	21,67 ms
	storitev	21,93 ms
	odjemalec	21,82 ms
10	DAO	22,45 ms
	storitev	31,41 ms
	odjemalec	29,37 ms
100	DAO	26,46 ms
	storitev	107,81 ms
	odjemalec	112,55 ms
1000	DAO	62,34 ms
	storitev	852,19 ms
	odjemalec	893,60 ms

Tabela 8.5: Rezultati trinivojske meritve zmogljivosti. Manj je bolje.



Graf 8.4: Rezultati trinivojske meritve zmogljivosti. Manj je bolje.

Rezultati slednje meritve kažejo na **hitrost ostale infrastrukture**, kajti sama poizvedba v podatkovni zbirki se je v vseh primerih izvedla enako hitro. Parameter *maxResults* namreč omeji le velikost rezultatov, ne vpliva pa nič na sortiranje v sami poizvedbi. Počasnejše izvajanje torej gre pripisati drugim dejavnikom, kot je prenašanje rezultatov med zbirko in programom. Še posebej se to pozna pri storitvi in odjemalcu, ki sta v primeru, da je največje število 1000, kar 14-krat počasnejša od DAO. To je posledica tega, da Hibernate vsak element posebej pograbi (angl. lazy fetch) iz zbirke šele po tem, ko se ga uporabi (nivo DAO ga ne uporabi). Pri trinivojski meritvi pri storitvi geokodiranja je že sama poizvedba (uporablja se združevanje tabel - join) pograbila (angl. eager fetch) vse podatke, zato so bili rezultati DAO bolj primerljivi s storitvijo in odjemalcem.

9 Usmerjanje

Algoritem za usmerjanje je bolj obširen in bolj zapleten, zato sem izkoristil edinstveno priložnost in se povezal s sošolcem in sodelavcem Ivanom Fućakom, ki je pred kratkim izdelal diplomsko delo na to temo [3]. Njegovo delo mi je pomagalo pri teoretičnem razumevanju, pri iskanju primernih rešitev in pri testiranju.

Usmerjanje je storitev, ki **izračuna najcenejšo pot iz lokacije A v lokacijo B**. Uporabniki lahko to pot primerjajo s svojo trenutno ali pa načrtujejo povsem novo. Pri tem lahko izbirajo, kaj jim pomeni *najcenejša* pot.

Ustrezna koda se nahaja v javinem paketu net.tinelstudio.gis.routing.

9.1 Algoritem

Usmerjanje implementira usmerjevalni **algoritem A*** (angl. izgovorjava "A star"), ki ga ovija v uporabno storitev. Algoritem A* je najbolj priljubljena izbira pri iskanju poti. Spada med algoritme za iskanje najkrajših poti pri grafih (angl. graph search algorithm).

Za usmerjanje je bilo treba rahlo prilagoditi podatkovni model, ki vsebuje nekaj novosti, ki jih uporablja samo usmerjevalni algoritem.

Nov je domenski objekt *StreetNode*, vsebuje pa vsa vozlišča vseh cest, to so začetne in končne točke vsakega odseka ceste.

Domenski objekt *Street* je dobil naslednje nove atribute:

- *lengthMeters* - dolžina odseka ceste v metrih,
- *level* - nivo ceste,
- *oneWay* - enosmernost ceste,
- *startNode* - začetno vozlišče in
- *endNode* - končno vozlišče.

Sam usmerjevalni algoritem A* sem implementiral povsem na novo. Preprost opis algoritma se nahaja na spletni enciklopediji Wikipedia v obliki psevdokode [6] in na spletni strani

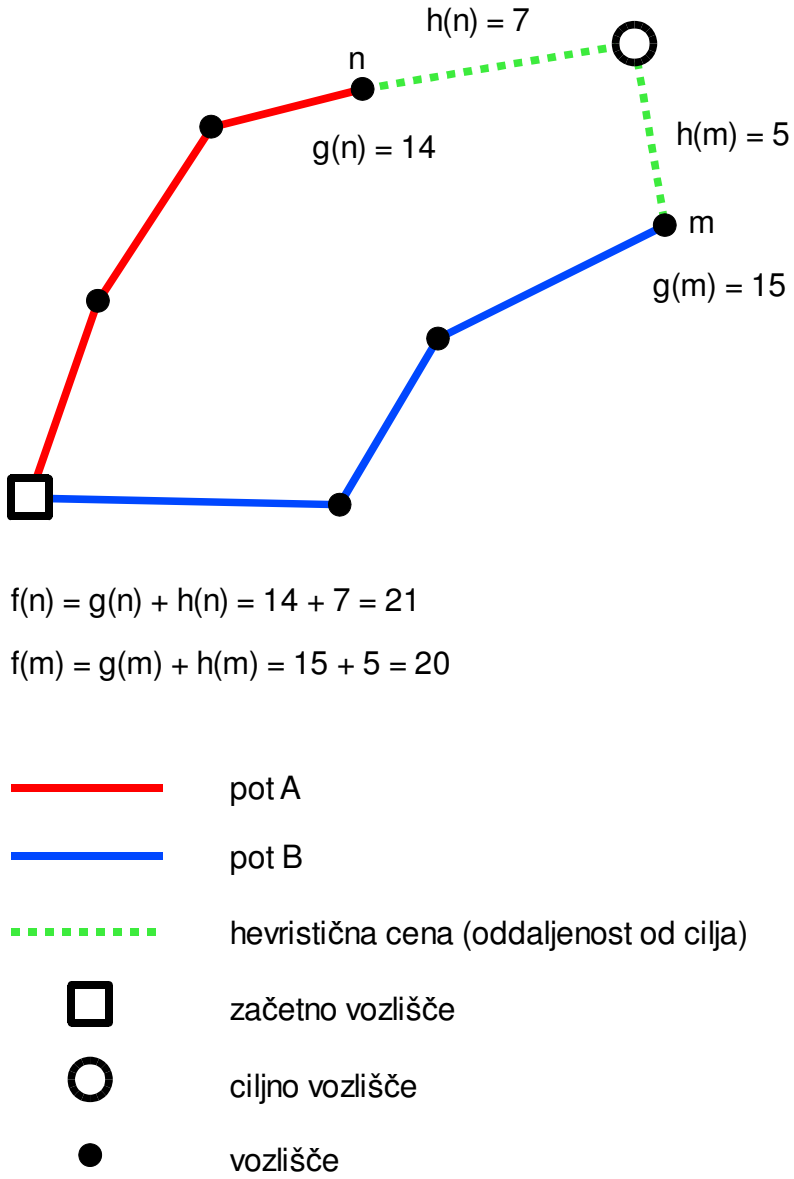
gospoda Amita [5]. Amitov algoritem je za odtenek boljši, saj nudi večji delovni razpon.

Algoritem A* je pravzaprav preprost. Sprehodi se po cestah od začetnega do ciljnega vozlišča. Pri tem za izračun optimalne poti uporablja dve vrsti **ocenjevanja**.

Na eni strani računa ceno že obdelane poti **g(n)**, na drugi strani pa hevristično išče najcenejšo smer proti ciljnemu vozlišču **h(n)**. Če seštejemo oboje skupaj, dobimo ceno celotne poti, ki gre skozi neko vmesno vozlišče n:

$$f(n) = g(n) + h(n) \tag{9.1}$$

V kolikor je cena te poti, recimo ji A, višja od cene poti, recimo ji B, ki gre, npr., skozi drugo vmesno vozlišče m, se za nadaljnjo obravnavo vzame pot B. To traja tako dolgo, dokler algoritem ne pride do ciljnega vozlišča.



Slika 9.1: Algoritem A išče najcenejšo pot. Pot A, ki gre skozi vozlišče n, je dražja (f(n) = 21) od poti B, ki gre skozi vozlišče m (f(m) = 20), zato se za nadaljnjo obravnavo vzame cenejša pot B.*

Od zunaj tako lahko nastavljamo obe pomembni funkciji: ceno že obdelane poti g(n) in hevristično ceno neznane poti do ciljnega vozlišča h(n). S tema dvema funkcijama lahko fino nastavljamo **obnašanje algoritma**. Pot lahko izračuna izredno hitro in razmeroma slabo (ne absolutno najcenejšo), ali pa zelo počasi in zajamčeno najcenejšo (kot algoritem Dijkstra).

Kaj šteje za ceno poti je lahko določeno različno: lahko šteje **le razdalja**, ali pa se tudi upošteva **nivo ceste** (npr. avtocesta je cenejša, ker je v praksi hitrejša in enostavnejša).

Poglejmo, kako te dve funkciji vplivata na obnašanje algoritma A*:

- V eni skrajnosti lahko nastavimo, da hevristična funkcija h(n) vedno vrne 0. Algoritem se spremeni v Dijkstra in **zajamčeno najde najcenejšo pot**. Pri tem je najbolj

počasen.
- Če je h(n) vedno manjša ali enaka od prave cene od vozlišča n do ciljnega vozlišča, potem algoritem še vedno **zajamčeno vrne najcenejšo pot**. Manjši kot je h(n), počasnejši je.
- Če je h(n) vedno enaka od prave cene od vozlišča n do ciljnega vozlišča, potem se algoritem obnaša perfektno in je **zelo hiter**. Vendar je pravo ceno zelo težko vedeti vnaprej (zato raje hevristična ocena). To bi bilo možno samo, če bi, npr., vnaprej izračunali ceno med vsakima dvema vozliščema.
- Če je h(n) kdaj večji od prave cene od vozlišča n do ciljnega vozlišča, potem algoritem ne bo vedno našel najcenejše poti, ampak bo **zelo hiter**.
- V drugi skrajnosti, če je h(n) vedno precej večji od g(n), potem cel algoritem sloni samo na h(n) in se obnaša kot metoda *požrešno najprej najboljši* (angl. greedy best-first search).

9.2 Storitev

Storitev usmerjanja je zasnovana tako, da odjemalec strežniku poda zahtevo s štirimi parametri:
- začetno lokacijo,
- ciljno lokacijo,
- funkcija ocenjevanja že obdelane poti g(n) in
- funkcijo hevurističnega ocenjevanja še neobdelane poti h(n).

Začetna lokacija in ciljna lokacija sta točki, podani v geografskih koordinatah z zemljepisno dolžino in širino v stopinjah.

Storitev pozna dve uporabni implementaciji funkcije g(n):
- **LengthCostFunction** - Vrne ceno, ki je enaka pravi dolžini poti v metrih (iz atributa *lengthMeters* domene *Street*).
- **LevelLengthCostFunction** - Vrne ceno, ki je enaka dolžini poti v metrih pomnoženi z utežjo glede na nivo ceste (iz atributa *level* domene *Street*). Uporabnik mora vnaprej pripraviti uteži za vse nivoje cest.

Za funkcijo h(n) se običajno uporabljajo različne hevristike: razdalja Manhattan, diagonalna razdalja, evklidska razdalja, kvadratna evklidska razdalja in bolj zahtevne. Vsem je skupno to, da so namenjene iskanju poti po mreži. V našem primeru je mreža zemeljska obla (poenostavljeno sfera), ki ima neskončno majhne kvadratke (stopinje zemljepisne dolžine in širine). Ker zemljepisna dolžina in širina nista sorazmerni in je mreža neskončno majhna, je uporaba preprostih funkcij za računanje razdalj nesmiselna. Zato sistem pozna le eno primerno fleksibilno funkcijo h(n):
- **GreatCircleDistanceHeuristicFunction** - Vrne ceno poti glede na razdaljo med dvema vozliščema na sferi (Zemlji). Hkrati omogoča nastavljanje **uteži**, ki se pomnoži s to razdaljo. V primeru, da je utež enaka 1, je cena enaka razdalji.

Ker sta začetni in ciljni lokaciji podani v geografskih koordinatah, mora storitev najprej **poiskati njima najbližja vozlišča**. Način iskanja je isti kot pri iskanju najbližjih naslovov pri storitvi nasprotnega geokodiranja, le da se tokrat namesto naslovov iščejo vozlišča (domena *StreetNode*).

```
from StreetNode as sn
where ST_Intersects(sn.point, :bbox) = true
  and ST_distance_sphere(sn.point, :point) <= :distance
order by ST_distance_sphere(sn.point, :point)
```

Koda 9.1: Poizvedba HQL: iskanje najbližjih vozlišč. ":point" je iskalna lokacija, ":distance" je iskalni radij v metrih, ":bbox" je iskalni pravokotnik v stopinjah.

Storitev kot rezultat vrne celotno pot v obliki seznama odsekov cest (z vsemi atributi). Poleg tega vrne še ceno celotne poti in porabljen čas algoritma v milisekundah.

```
Coordinate startLocation=new Coordinate(14.596, 46.112);
Coordinate goalLocation=new Coordinate(15.100, 46.369);
Cost cost=new LengthCost();
Heuristic heuristic=new GreatCircleDistanceHeuristic(1.2);
Route route=this.routingService.findRoute(startLocation, goalLocation,
  cost, heuristic);
```

Koda 9.2: Primer uporabe storitve usmerjanja.

9.3 Zmogljivost

Pri storitvi usmerjanja sam algoritem ne teče kot ena poizvedba v podatkovni zbirki, pač pa potrebuje cestne odseke. Ko obdela en odsek ceste, iz zbirke zahteva vse odseke, ki se navezujejo na ta odsek (po vozliščih). Poizvedba upošteva tudi enosmernost cest.

```
from Street
where startNode = ? or (oneWay = false and endNode = ?)
```

Koda 9.3: Poizvedba HQL, ki najde vse cestne odseke, ki se začnejo (prvi "?") ali končajo (drugi "?") v podanem vozlišču. Pri tem upošteva morebitno enosmernost odseka.

Meritve, ki sem jih opravil, kažejo ravno na to poizvedbo, saj porabi praktično ves čas algoritma.

Pri meritvah sem spreminjal hevristično funkcijo oz. njeno utež, saj je v vseh primerih nastopala *GreatCircleDistanceHeuristic*:

- Pri uteži 0 (vsa vozlišča so enako oddaljena od cilja), algoritem deluje kot Dijkstra in vedno najde najcenejšo pot na račun počasnosti.
- Pri uteži 1 (vsa vozlišča so od cilja oddaljena toliko, kot znaša zračna razdalja), algoritem še vedno v večini primerov najde najcenejšo pot, vendar svoje delo opravi občutno hitreje.
- Pri uteži 1,2 (vsa vozlišča so od cilja oddaljena 20 % več, kot znaša zračna razdalja),

algoritem najde poceni pot in je pri tem razmeroma hiter. Ta utež se mi zdi najbolj primerna.

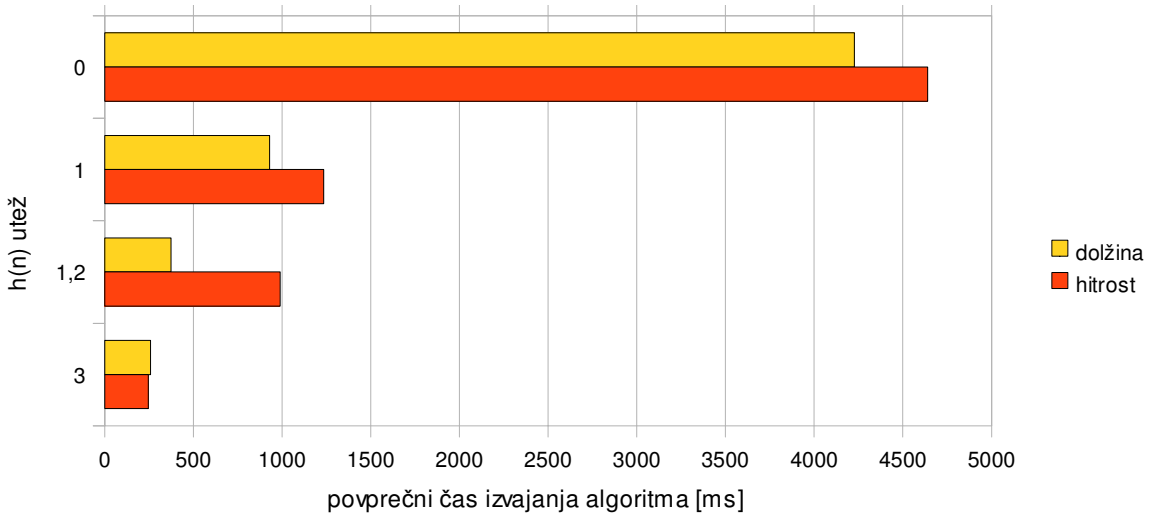
- Pri uteži 3 (vsa vozlišča so trikrat dlje od cilja, kot znaša zračna razdalja), algoritem išče pot vedno v najbolj ravni črti od začetka do cilja. Pri tem je zelo hiter, a redko vrne uporabne rezultate.

Spreminjal sem tudi funkcijo g(n) in sicer sem enkrat za ceno uporabil le pravo dolžino ceste ("dolžina"), funkcijo *LengthCost*, drugič pa je bila cena odvisna tudi od nivoja ceste ("hitrost"), funkcija *LevelLengthCost*. Ceno posameznega nivoja ceste, v podatkovni zbirki sem imel tri, sem nastavil tako, da so bile važnejše ceste najcenejše (ker so običajno najhitrejše), najmanjše pa najdražje (ker so običajno najpočasnejše).

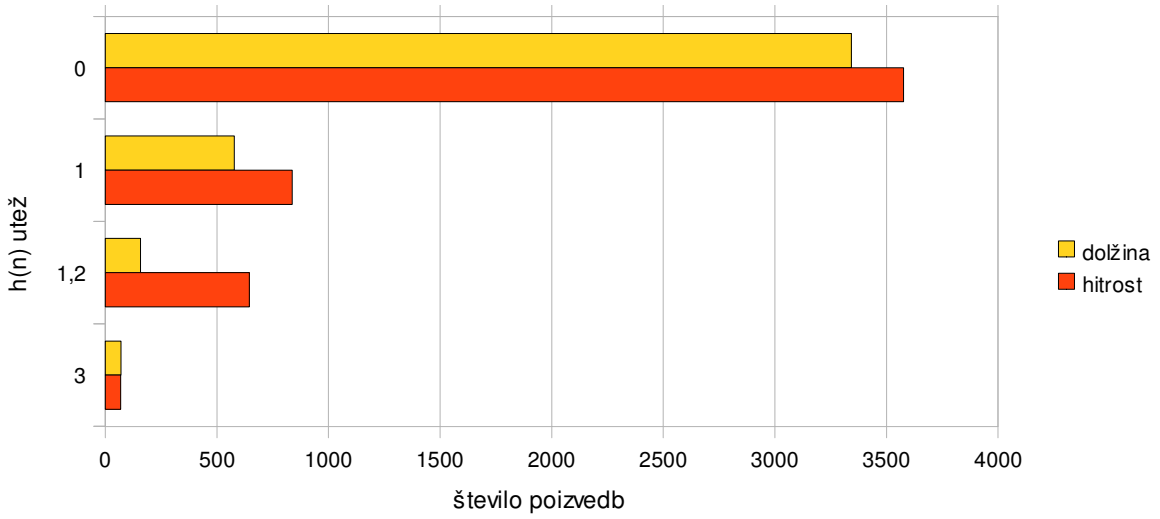
Pri merjenju sem algoritem pognal tridesetkrat z enim zagonom okolja, vsako meritev pa trikrat. Začetna lokacija je bila v Škofljici, ciljna pa v Ljubljana-Dravlje.

h(n) utež	g(n)	Povprečni čas izvajanja algoritma	Število poizvedb
0	dolžina	4227,09 ms	3343
	hitrost	4640,28 ms	3576
1	dolžina	930,21 ms	578
	hitrost	1235,40 ms	837
1,2	dolžina	375,00 ms	158
	hitrost	990,45 ms	645
3	dolžina	258,33 ms	70
	hitrost	246,88 ms	69

Tabela 9.1: Rezultati meritve hitrosti algoritma A in števila poizvedb pri različnih utežeh hevristične funkcije h(n) in dveh različnih funkcijah ocenjevanja že obdelane poti g(n).*



Graf 9.1: Rezultati meritve hitrosti algoritma A pri različnih utežeh hevristične funkcije h(n) in dveh različnih funkcijah ocenjevanja že obdelane poti g(n).*



Graf 9.2: Rezultati meritve števila poizvedb algoritma A pri različnih utežeh hevristične funkcije h(n) in dveh različnih funkcijah ocenjevanja že obdelane poti g(n).*

Oblika grafa 9.1, ki ponazarja hitrost algoritma, je zelo podobna obliki grafa 9.2, ki ponazarja število poizvedb. To pomeni, da je hitrost algoritma odvisna **samo od števila poizvedb**. Možna izboljšava algoritma bi bila, da bi iz baze potegnil odseke cest večjega območja okoli podanega vozlišča, nekakšen **iskalni pravokotnik**. S tem bi sicer upočasnili samo poizvedbo, a bi se število le-teh občutno zmanjšalo. Namesto sedanjih 3343 pri prvem testu, bi za to pot bile dovolj le, recimo, štiri.

Zanimivo je, da algoritem v našem primeru izračuna povsem **enake poti** pri hevrističnih utežeh 0 in 1. A je pri uteži 1 do 4,5-krat hitrejši! Pri uteži 1,2 je algoritem še hitrejši, pri tem pa so razlike v poti minimalne, a pot ni več absolutno najcenejša.

Zanimiva je tudi velika razlika pri hevristični uteži 1,2 med potjo "dolžina" in med potjo "hitrost". Slednja se namreč skoraj trikrat počasneje izračuna. Razlog tiči v rezultatu, ki je lepo viden na sliki 9.2. Pot "hitrost" nas iz Škofljice v Ljubljana-Dravlje pripelje po avtocesti, medtem ko pot "dolžina" vodi skozi center Ljubljane. Pri tem je izračun poti "hitrost" na meji med tem, ali bi tekla po avtocesti, ali skozi center. Namreč, če nastavimo hevristično utež na 1,5, že prevlada pot skozi center.

Slika 9.2: Rezultat usmerjanja iz Škofljice v Ljubljana-Dravlje. Pot "hitrost" (zelena črta) nas pripelje po avtocesti, medtem ko pot "dolžina" (modra črta) vodi skozi center Ljubljane. Na začetku in na koncu se poti prekrivata. Utež h(n) je bila v obeh primerih 1,2. Na sliki so lepo vidni nivoji cest. Ceste najvišjega nivoja so oranžne, srednjega rumene in najnižjega sive. V primeru poti "hitrost" to pomeni, da so oranžne črte najcenejše (najhitrejše), sive pa najdražje (najpočasnejše).

Pri testiranju zmogljivosti storitve sem izvedel *trinivojsko* meritev, ki je pokazala hitrostne razlike med **tremi različnimi nivoji** v sami arhitekturi storitve:

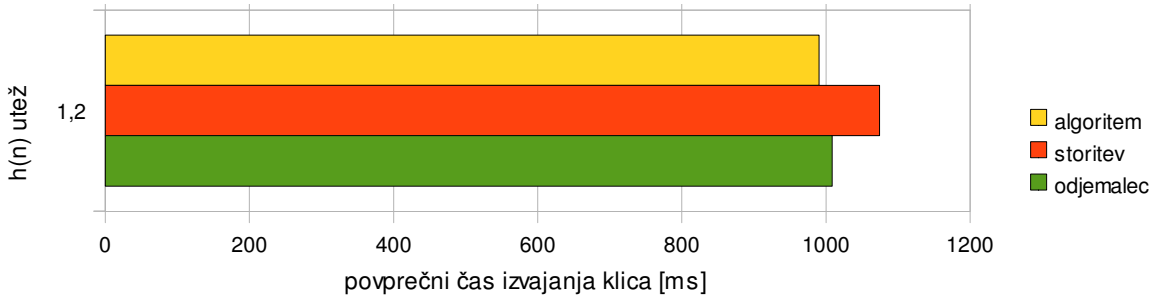
1. **Algoritem** - Neposreden klic algoritma, ki izračuna pot.
2. **Storitev** - Klic storitve, ki pripravi posredovane iskalne podatke (poišče začetno in ciljno vozlišče) in pokliče algoritem.
3. **Odjemalec** - Klic storitve prek HTTP Invokerja na zagnan strežnik storitve.

Vse tri nivoje sem testiral na lokalnem računalniku (testi, strežnik in zbirka so tekli lokalno). Vsak klic sem ponovil tridesetkrat z enim zagonom okolja, vsako meritev pa trikrat.

Začetna lokacija je bila v Škofljici, ciljna pa v Ljubljana-Dravlje. Funkcija h(n) je bila *GreatCircleDistanceHeuristic* z utežjo 1,2. Funkcija g(n) je bila *LevelLengthCost* s primerno utežnimi nivoji: važnejše ceste so najcenejše, manjše so najdražje.

Nivo	Povprečni čas izvajanja klica	
algoritem	990,45 ms	92,19 %
storitev	1074,30 ms	100,00 %
odjemalec	1008,84 ms	93,91 %

Tabela 9.2: Rezultati trinivojske meritve zmogljivosti. Manj je bolje.



Graf 9.3: Rezultati trinivojske meritve zmogljivosti. Manj je bolje.

Rezultati meritve kažejo na **hitrost celotne infrastrukture**. Storitev in odjemalec sta pričakovano nekoliko počasnejša od golega algoritma, kar je posledica dodatnih funkcij, ki jih opravita (predvsem iskanje začetnega in ciljnega vozlišča). Zanimiv pa je odjemalec, saj je hitrejši od storitve. To je posledica tega, da je strežnik skozi vse meritve tekel in si je tako že napolnil predpomnilnik (angl. cache). Komunikacija med strežnikom in odjemalcem (protokol HTTP) pa očitno ne vpliva toliko, kar je posledica tega, da sta strežnik in odjemalec tekla lokalno, na istem računalniku.

10 Izkušnje in nadaljnje delo

Sistem vsebuje nekaj uspešnih rešitev, ki drugače predstavljajo resne omejitve. V podatkovnem modelu se podatki **ne podvajajo** in niso projicirani, zato so neposredno uporabni za analizo **kjerkoli na svetu**. To je velika prednost, še posebej, če to velja tudi pri storitvi nasprotnega geokodiranja. To je redka praksa, saj so GIS-i običajno lokalne narave (npr. za ZDA), kjer pa načeloma ustreza ena projekcija in so zato podatki predprojicirani. Seveda pri svetovni rabi trpi hitrost, saj se indeksi ne uporabljajo vedno, a kakor smo videli pri meritvah zmogljivosti, je pametna uporaba poizvedbe lahko skoraj enako hitra kot tista, ki vedno uporablja indekse. Zanimivost, ki ostane za nadaljnjo obravnavo, je funkcija PostGIS *ST_Intersects*, ki je skrivnostno počasna pri velikem iskalnem pravokotniku. To sicer v praksi ni moteče, ker velja le za zelo velik iskalni radij, a morda bi lahko prispevali kakšno plodno debato razvijalcem PostGIS-a.

Vsak geografski podatek lahko ima **poljubno število imen**. Na prvi pogled sicer ni videti, kakšno prednost prinaša to s seboj. A predstavljajte si ulico, ki ima dve imeni. Ali pa cesto, ki ima dve oznaki. Že v Sloveniji je moč naleteti na takšne primere, npr., avtocesta A1 je na štajerskem koncu tudi E57, na notranjskem koncu pa tudi E61 in E70. Da imajo tudi zgradbe več imen, je odvisno od tega, kako jih poimenujemo. Npr., nakupovalnemu središču lahko dodamo vsa imena trgovin, ki so v njem. Vse to omogoča podatkovni model, le pravilno ga moramo uporabiti. To pa je hkrati tudi slabost, ki se rahlo kaže pri storitvi geokodiranja. Poizvedba vsebuje združevanje tabel (funkcija *join*), ki je značilno po tem, da je počasno. Meritve zmogljivosti so pokazale, kako največje število zadetkov vpliva na samo poizvedbo. V primerjavi z nasprotnim geokodiranjem je nalaganje stotih zadetkov vsaj trikrat počasnejše (nivo storitev). Definitivno je tukaj še veliko možnosti za nadaljnje izboljšave. Poizvedbe z združevanjem tabel bi lahko enostavno nadomestili kar s **celobesedilnim iskanjem**, a je to ravno tukaj odpovedalo - zaradi podatkovnega modela, ki omogoča poljubno število geoimen. Tudi drugače je še veliko odprtih vprašanj, recimo skupna hitrost, saj bi bilo treba po celobesedilnem iskanju iz zbirke potegniti vse rezultate.

Bi pa celobesedilno iskanje lahko močno izboljšalo uporabniško izkušnjo v povezavi z dobrim grafičnim uporabniškim vmesnikom odjemalca. Vanj bi vgradili funkcijo, ki sproti išče, kar uporabnik vpisuje v vnosno polje, in prikazuje rezultate v spustnem seznamu. Ko bi se uporabnik odločil za enega izmed zadetkov, bi se ta zadetek naložil iz zbirke in prikazal na zemljevidu ali uporabil kje drugje.

Ravno nasprotno od storitve geokodiranja, ki je namenjena bolj uporabnikom - ljudem, je storitev nasprotnega geokodiranja uporabna večinoma samo za avtomatske namene. Predstavljamo si lahko odjemalce, ki želijo nekajkrat v sekundi izvedeti, kateri znan geografski kraj je najbližji določeni lokaciji, ki jo vrnejo razne naprave GPS. Pri tako pogosti uporabi bo moral odjemalec pametno uporabljati to storitev. Ena možnost je uporaba **predpomnjenja** (angl. cache, buffer) **rezultatov**, kar pomeni, da si zapomni vse rezultate od vseh iskanj in namesto iskanja uporabi kar shranjene rezultate, če ustrezajo lokaciji. Ali ustrezajo lokaciji in ali bi iskanje vrnilo enake rezultate, kot so ti, iz predpomnilnika, zna biti zelo zahtevno ugotoviti. Pravzaprav so vsi načini uporabe predpomnjenja zelo delikatni in zahtevni. Velikokrat se ravno zaradi njih uporabljajo drugačni protokoli ali dodatni podatki.

Naslednja, bolj obetavna možnost je **kopičenje več zahtev** skupaj (angl. bulk). Namesto da odjemalec nekajkrat na sekundo sproži iskanje, lahko zahtevke nekaj časa kopiči in potem z enim iskanjem poišče vse rezultate. Na primer, enkrat na sekundo, ali ko se nabere največ 100 zahtevkov. Tukaj bi veliko privarčevali pri omrežni komunikaciji, povprečni čas reševanja enega zahtevka bi bil bistveno manjši.

Predpomnjenje rezultatov in kopičenje več zahtev skupaj predstavlja zanimivo možnost za nadaljnje delo, oboje pa bi z nekaj popravki na strežniškem delu izvedli večinoma na odjemalčevem delu.

Storitev usmerjanja je uporabna iz več zornih kotov. Lahko jo uporabljajo uporabniki, ki želijo poceni pot od ene lokacije do druge, lahko jo uporablja stroj za samodejno izdelavo poti, lahko jo uporablja kakšen računalnik za optimizacijo delovnih prevozov. Ne nazadnje jo lahko uporabimo tudi za **navigacijo**. Pri vseh teh načinih uporabe bi bilo smiselno najti optimalne ocenjevalne funkcije, kar zna biti tako zahtevno, kot je zahtevna dejanska cestna mreža.

V prvi fazi bi vsekakor lahko algoritem bistveno pohitrili s tem, da bi občutno **zmanjšali število poizvedb**. To bi storili tako, da bi uporabili predpomnilnik in bi iz podatkovne zbirke naenkrat potegnili vse ceste, ki padejo v večje območje okoli obravnavanega vozlišča. Na primer, pri sedanjem izračunu poti iz Škofljice v Ljubljana-Dravlje pri običajnih parametrih se podatkovna zbirka uporabi 645-krat in praktično ves čas vzamejo ravno te številne poizvedbe in komunikacija z njimi. Če bi iz zbirke vzeli vse ceste v večjem območju okoli obravnavanega vozlišča kot samo neposredno navezujoče ceste, bi lahko število poizvedb zmanjšali na vsega, recimo, štiri.

Za nadaljnjo obravnavo bi, poleg tega predloga, vzeli pod drobnogled tudi projekt **pgRouting**, ki dodaja zmožnost usmerjanja neposredno v PostGIS. Že samo dejstvo, da teče neposredno v zbirki, nas prepričuje, da gre za zelo hitro rešitev.

Storitve so napisane v javini arhitekturi, navzven se predstavljajo kot spletni strežnik, ki pa razume le serializirane javine objekte v obliki, kot jo uporablja Springov HTTP Invoker. Odjemalci morajo tako uporabiti isto tehnologijo. Storitve bi lahko preoblikovali tako, da bi se navzven predstavljale v tehnologiji standardnih **spletnih storitev** (angl. web services). Odjemalci in strežniki v tej tehnologiji komunicirajo le zgolj v obliki XML, zahteve in odgovori so natančno definirani v shemi XSD. Iz izkušenj vem, da bi bila preobrazba obstoječih storitev razmeroma enostavna, če bi uporabili Springov MVC modul. Še več, lahko bi HTTP Invoker obdržali. Ampak iz izkušenj tudi vem, da bi bila takšna storitev nekajkrat počasnejša, saj bi se vsaka zahteva in odgovor najprej po shemi XSD pretvorila v podatke XML (angl. (un)marshall). Po omrežju bi se prenašali večji paketi, ob morebitnih kasnejših razširitvah storitev pa bi morali posebej paziti na posodobitev sheme XSD.

V kolikor nas skrbi za privatnost in varnost podatkov, kar je prav, bi v nadaljevanju lahko komunikacijo med strežniki in odjemalci ovili s kriptografskim protokolom **TLS/SSL**. Javina arhitektura je na to pripravljena.

Del sistema	Izboljšave
geokodiranje	pohitritev združevanje tabel, celobesedilno iskanje
nasprotno geokodiranje	pohitritev funkcije *ST_Intersects*, predpomnjenje rezultatov, kopičenje zahtev
usmerjanje	iz zbirke vzamemo ceste večjega območja, projekt pgRouting
vsi deli	spletne storitve, TLS/SSL komunikacija

Tabela 10.1: Možne izboljšave delov sistema v nadaljnjem delu.

Geografski informacijski sistem bi lahko obogatili tudi tako, da bi mu, poleg novih domen v podatkovnem modelu, dodali še kakšno orodje. Eno izmed njih je, na primer, **storitev interesantnih točk** (angl. Points of Interest - POI), ki bi lahko vsebovala lastne privatne uporabnikove točke ali pa javne zanimivosti (npr. črpalke za gorivo, moteli, restavracije, parki ipd.). Tukaj bi bil še posebej zanimiv del varnosti podatkov pri privatnih točkah, saj bi morali zagotoviti, da so določene interesantne točke na voljo samo določenim uporabnikom.

Uporabnikom bi lahko geografske podatke prikazali v obliki zemljevidov v grafičnem vmesniku (na spletnih straneh ali v namizni aplikaciji). Upodabljanje podatkov bi lahko, na primer, prepustili strežniku **GeoServer**, ki vse to že zna in se pri tem drži lepega nabora standardov.

11 Sklep

Sistem je brez pretiravanja lahko **življenjski projekt**. Sistem takšne velikosti in neomejenih možnosti je mogoče razvijati v nedogled. Ravno zaradi tega sem se moral na neki točki ustaviti in se precej zadržati, da ne bi podaljševal že tako dolgega dela.

Sistem ima nekaj uspešnih rešitev, ki lahko drugače predstavljajo trn v peti. V podatkovnem modelu se podatki ne podvajajo in niso projicirani, zato so neposredno uporabni za analizo kjerkoli na svetu. Vsak geografski podatek lahko ima poljubno število imen. Storitve s pridom uporabljajo podatke tako, da je čas obdelave čim manjši. Tukaj je sicer še mnogo manevrskega prostora. Algoritme in poizvedbe lahko vedno izboljšamo.

Upam si trditi, da je sistem **dobro prilagojen** za potrebe kakšnega podjetja ali organizacije, čeprav ga še nisem imel možnosti preizkusiti. Sem pa ves čas izdelave imel v mislih podjetje Autronic, d.o.o., iz Ljubljane, kjer sem večino študija imel opravka z geografskimi podatki. To podjetje bi ta sistem lahko s pridom uporabilo - morda ga bo.

Meritve zmogljivosti storitev kažejo **solidne rezultate**, še posebej tiste, ki bi se v praksi uporabljale. Ni mišljeno, da bi se storitev geokodiranja uporabljalo avtomatsko, temveč bi jo uporabljali samo uporabniki - ljudje. Zato tukaj milisekunde niso toliko pomembne, kot je pomemben ustrezen rezultat. K izboljšanju rezultata bi dobro vplivala uporaba celobesedilnega iskanja, še posebej v povezavi z dobrim grafičnim uporabniškim vmesnikom.

Ravno nasprotno od storitve geokodiranja, je storitev nasprotnega geokodiranja uporabna večinoma samo za strojne namene. Odjemalci želijo nekajkrat v sekundi izvedeti, kateri znan geografski kraj je najbližji določeni lokaciji, ki jo vrnejo razne naprave GPS. Odjemalec bo moral pametno uporabljati to storitev. Uporaba predpomnjenja rezultatov in kopičenja več zahtev skupaj bo nujna.

Storitev usmerjanja je uporabna iz več zornih kotov. Lahko jo uporabljajo uporabniki, ki želijo poceni pot od točke A do točke B, lahko jo uporablja stroj za samodejno izdelavo poti, lahko jo uporablja kakšen računalnik za optimizacijo delovnih prevozov. Ne nazadnje jo lahko uporabimo tudi za navigacijo. Pri vseh teh načinih uporabe bi bilo smiselno najti optimalne ocenjevalne funkcije, kar zna biti tako zahtevno, kot je zahtevna dejanska cestna mreža. V prvi fazi pa bi vsekakor lahko algoritem bistveno pohitrili s tem, da bi občutno zmanjšali število poizvedb. To bi storili tako, da bi iz podatkovne zbirke naenkrat potegnili vse ceste, ki padejo v večje območje okoli obravnavanega vozlišča.

Storitve so napisane v javini arhitekturi, navzven se predstavljajo kot spletni strežnik, ki pa razume le serializirane javine objekte v obliki, kot jo uporablja Springov HTTP Invoker. Razvijalcem odjemalca tako preostane le eno: da uporabi isto tehnologijo. Če bi želeli vse splošno uporabnost, bi lahko storitve navzven predstavili v tehnologiji pravih spletnih storitev. Odjemalci in strežniki v tej tehnologiji komunicirajo le zgolj v obliki XML, zahteve in odgovori so natančno definirani v shemi XSD.

Sistem bi lahko obogatili tako, da bi mu, poleg novih domen v podatkovnem modelu, dodali še kakšno orodje. Na primer, storitev interesantnih točk in strežnik za upodabljanje zemljevidov GeoServer.

Za konec bi rad izrazil navdušenje nad skriptnim programskim jezikom **Groovy**, ki predstavlja skriptno alternativo ali bolje dopolnilo jeziku Java. V določenih primerih je zelo uporaben, ker skrajša mučno pisanje določene kode. V sistemu bi ga s pridom uporabil pri vseh zagonskih razredih in pri raznih pretvarjanjih podatkov. Vsa zrna bi bila spisana v njem, še posebej domenski objekti. A takšen jezik potrebuje zelo dobro podporo v razvojnem okolju. Groovy je zaenkrat nima, saj je podpora v trenutni različici SpringSource Tool Suite IDE-ja klavrna, prav takšna je podpora tudi pri Mavenu. Končni rezultat je takšen, da je sedaj z groovyjem še celo nekaj več dela. Zato sem se ga izogibal. Ni pa daleč čas, ko bo to urejeno (baje da so že pri komercialnem razvojnem okolju IntelliJ IDEA zelo blizu), tako kot je urejena izvrstna podpora za javo, brez katere si razvoja ne znam več predstavljati.

Dodatek A Navodila za uporabo sistema

Kratka navodila za razvoj, testiranje, pakiranje in poganjanje sistema.

Seznam pomembnih uporabljenih orodij in njihovih različic:

- Pogonsko okolje:
  - Microsoft Windows XP 32-bit Service Pack 3
  - PostgreSQL 8.3.7
  - PostGIS 1.3.6
  - JDK 1.6.0_16
  - Groovy 1.6.3

- Razvojno in testno okolje:
  - SpringSource Tool Suite 2.1.0.SR01 na Eclipseu 3.5 s podporo groovyju
  - Apache Maven 2.2.1
  - JUnit 4.4
  - EasyMock 2.5.1
  - EasyMock Class Extentions 2.4
  - Jetty 6.1.19

- Knjižnice:
  - Apache Commons DBCP 1.2.2
  - Apache Commons Logging 1.1.1
  - GeoTools 2.5.1
  - Hibernate Annotations 3.2.1.ga
  - Hibernate 3.2.7.ga
  - Hibernate Spatial 1.0-M1
  - JTS Topology Suite 1.9
  - Log4j 1.2.14
  - PostgreSQL JDBC 8.3-603.jdbc4
  - Spring Framework 2.5.6.SEC01

- Strojna oprema:
  - osnovna plošča Intel DP35DP
  - CPU Intel Core2 Duo E4400 2 GHz
  - pomnilnik DDR2 4 GB
  - trdi disk Western Digital WD10EADS 1 TB

Sistem je sestavljen iz trinajstih paketov, ki so hkrati projekti za okolje SpringSource Tool Suite (STS):

1. tinel-commons
2. tinel-gis-parent
3. tinel-gis-model
4. tinel-gis-common-client
5. tinel-gis-common-server
6. tinel-gis-geocoding-client
7. tinel-gis-geocoding-compass
8. tinel-gis-geocoding-server
9. tinel-gis-reversegeocoding-client
10. tinel-gis-reversegeocoding-server
11. tinel-gis-routing-client
12. tinel-gis-routing-server
13. tinel-gis-client

Vsi projekti so pripravljeni za uporabo z orodjem Maven (vsi imajo datoteke *pom.xml*). Projekt tinel-gis-parent je poseben. Je glavni projekt, ki ne vsebuje nobene izvorne kode, le definira sestavo vseh ostalih projektov, ki so v tem smislu njegovi podprojekti.

Sistem potrebuje delujočo podatkovno zbirko. Po namestitvi PostgreSQL-ja z dodatkom PostGIS moramo najprej ustvariti novo zbirko. To najlažje storimo s priloženim orodjem pgAdmin. Za ustvarjanje entitet z relacijami je pripravljena skripta groovy *DbSchemaCreator.groovy*, ki se nahaja v projektu tinel-gis-model v direktoriju *src/main/java*. V skripti nastavimo ustrezen naslov do nove zbirke (polje *dbUrl*), uporabniško ime (polje *dbUser*) in geslo (polje *dbPassword*). Nato skripto poženemo v okolju Groovy, pri čemer ne smemo pozabiti na vključitev knjižnice PostgreSQL JDBC. Podatkovni zbirka bi sedaj morala biti pripravljena in čas je, da jo napolnimo s svojimi geografskimi podatki.

Ko uspešno namestimo JDK, Maven in STS, lahko projekte preprosto uvozimo v delovno okolje STS-ja in tam opravimo večino potrebnih nalog. Lahko pa se izognemo STS-ju in sistem zapakiramo lastnoročno, z uporabo Mavena:

V konzoli se postavimo v glavni projekt tinel-gis-parent in preprosto poženemo:

```
mvn install
```

V direktorij *target* od strežnikov tinel-gis-geocoding-server, tinel-gis-reversegeocoding-server in tinel-gis-routing-server bo Maven zapakiral vsak strežnik v obliki WAR. Datoteko WAR samo prenesemo v enega izmed vsebnikov (npr. Tomcat ali Jetty) in sistem bi moral delovati.

Pred pakiranjem (in testiranjem v okolju STS) moramo pregledati še nastavitve. Vsaka

storitev posebej vsebuje datoteko *db.properties* (nahaja se v direktoriju *src/main/resources*), ki definira dostop do podatkovne zbirke (lokacijo, uporabniško ime, geslo), in datoteko za ostale nastavitve (*geocoding.properties*, *reverseGeocoding.properties*, *routing.properties*), ki jo je smiselno pregledati.

Pri aktivnem testiranju, ki ga je smiselno opravljati le v okolju STS, je na voljo precej testov, ki za delovanje nujno potrebujejo podatkovno zbirko. Zato se ti testi ob pakiranju ignorirajo. Namenjeni so samo razvijalcem (sam sem z njimi opravil vse meritve zmogljivosti), med njimi so celo "testi", ki samo poženejo vdelan vsebnik Jetty. Ti testi se ignorirajo tudi ob ročnem zagonu v okolju STS, zato jim moramo najprej odstraniti oznako @*Ignore*.

Seznam slik

Seznam tabel

Seznam grafov

Seznam izrezkov kode

Literatura

[1] S. Dutch. (2008, 1. apr.). The Universal Transverse Mercator System. Dostopno na: http://www.uwgb.edu/DutchS/FieldMethods/UTMSystem.htm

[2] J. Fee. (2006, 18. nov.). ESRI ArcGIS Server Licensing – Be Ready to Get Out Your Checkbook. *James Fee GIS Blog*. Dostopno na: http://www.spatiallyadjusted.com/2006/11/18/esri-arcgis-server-licensing-be-ready-to-get-out-your-checkbook/

[3] I. Fućak, *Usmerjanje v Geografskem informacijskem sistemu*: Diplomsko delo na univerzitetnem študiju, Ljubljana: Fakulteta za računalništvo in informatiko, 2009. Dostopno na: http://eprints.fri.uni-lj.si/822/

[4] X Lopez. (2003). The Future of GIS: Real-time, Mission Critical, Location Services. Predstavljeno na Cambridge Conference 2003. Dostopno na: http://www.cambridgeconference.org/previous_conferences/2003/camconf/papers/8-1.pdf

[5] A. Patel. (2009, 1. jun.). Amit's A* Pages: Implementation notes. Dostopno na: http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html

[6] Wikipedia contributors. (2009, 18. avg.). A* search algorithm. *Wikipedia, The Free Encyclopedia*. Dostopno na: http://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=308643667

[7] Wikipedia contributors. (2009, 2. sept.). Geographic information system. *Wikipedia, The Free Encyclopedia*. Dostopno na: http://en.wikipedia.org/w/index.php?title=Geographic_information_system&oldid=311520260

[8] Wikipedia contributors. (2009, 4. avg.). JavaBean. *Wikipedia, The Free Encyclopedia*. Dostopno na: http://en.wikipedia.org/w/index.php?title=JavaBean&oldid=306021293

[9] Wikipedia contributors. (2009, 18. avg.). Plain Old Java Object. *Wikipedia, The Free Encyclopedia*. Dostopno na: http://en.wikipedia.org/w/index.php?title=Plain_Old_Java_Object&oldid=308689800

[10] Wikipedia contributors. (2009, 18. avg.). Universal Transverse Mercator coordinate system. *Wikipedia, The Free Encyclopedia*. Dostopno na: http://en.wikipedia.org/w/index.php?title=Universal_Transverse_Mercator_coordinate_system&oldid=308610171

[11] Wikipedia contributors. (2009, 18. avg.). Vincenty's formulae. *Wikipedia, The Free Encyclopedia*. Dostopno na: http://en.wikipedia.org/w/index.php?title=Vincenty%27s_formulae&oldid=310022063

[12] (2006, 28. dec.). Address Data Model. *ESRI*. Dostopno na: http://support.esri.com/index.cfm?fa=downloads.dataModels.filteredGateway&dmid=32

[13] (2009). Bing Maps for Enterprise from Microsoft - Integrated Mapping, Imaging, Search and Location Web Service. *Microsoft Corporation*. Dostopno na: http://www.microsoft.com/maps/

[14] (2009). FreeGIS Database. Dostopno na: http://www.freegis.org/database/?cat=0

[15] (2009). Geodesics on the Ellipsoid. Dostopno na: http://charles.karney.info/geographic/geodesic.html

[16] (2009). Google Latitude. *Google*. Dostopno na: http://www.google.com/mobile/products/latitude.html

[17] (2009, 27. maj). Google Maps/Google Earth APIs Terms of Service. *Google*. Dostopno na: http://code.google.com/apis/maps/terms.html

[18] (2009). Google Maps API Reference. *Google*. Dostopno na: http://code.google.com/apis/maps/documentation/reference.html Pogl. enum GGeoAddressAccuracy

[19] (2009). GRS 1980 Authalic Sphere. EPSG Geodetic Parameter Registry. Dostopno na: http://www.epsg-registry.org/ EPSG: 7048

[20] (2009). Lokus. *Mobitel d.d.* Dostopno na: http://www.mobitel.si/storitve/lokus.aspx

[21] (2009, 30. maj). Most Popular Programming Languages. *Tiwebb Ltd*. Dostopno na: http://www.devtopics.com/most-popular-programming-languages/

[22] (2008, 9. sept.). OpenGIS Location Services (OpenLS): Core Services. *Open Geospatial Consortium Inc*. Dostopno na: http://portal.opengeospatial.org/files/?artifact_id=22122

[23] (2009). The IoC container. *SpringSource*. Dostopno na: http://static.springsource.org/spring/docs/2.5.x/reference/beans.html Pogl. 3.3.1

[24] (2009). Using PostGIS. Dostopno na: http://postgis.refractions.net/documentation/manual-1.3/ch04.html

[25] (2009). Web Search Help: Features: Query suggestions. *Google*. Dostopno na: http://www.google.com/support/websearch/bin/answer.py?hl=en&answer=106230