SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1554

Razvoj programske podrške za označavanje objekata u slijedovima slika

Bojan Popović

Zagreb, srpanj 2010.

SADRŽAJ

1. Uvod **1**

2. Marker **3**

2.1. Najvažnije značajke aplikacije 3

2.1.1. Čitanje slijedova slika iz komprimiranih datotečnih formata . 3

2.1.2. Označavanje znakova 4

2.1.3. Spremanje oznaka 4

2.2. Arhitektura Model-Pogled-Upravljač 5

2.3. Organizacija izvornog koda po paketima 5

2.4. Rad sa znakovnim konstantama 7

3. Rad s vanjskim komponentama **8**

3.1. Hijerarhija komponenata 9

4. Povezivanje vanjskih komponenti i bazne aplikacije **15**

4.1. Korisnički alati 15

4.2. Ugradnja klasifikatora 15

4.2.1. Vizualne komponente 15

4.2.2. Podrška za tipkovnicu 17

4.2.3. Fokus vizualnih komponenata 20

4.2.4. Ugradnja detektora 22

5. Zaključak **25**

Literatura **27**

1. Uvod

Digitalna arhiva znakova pokazala se korisnim pomagalom za održavanje samih znakova, izgradnju interaktivnih prometnih karata, povećanje razine sigurnosti u prometu, te unaprijeđenje kvalitete prometnih puteva. Anotacija lokacije i klase prometnih znakova na nekom širem području, primjerice u jednoj županiji ili državi predstavlja težak problem ukoliko se pristup svede na standardne kartografske postupke. Takva arhiva nudi i početnu točku za inovativne modele. Primjerice, sustav ugrađen u automobil koji bi bio sposoban automatski razlučivati prometnu signalizaciju te pružati sugestije vozaču prilikom vožnje bio bi revolucionaran. Međutim, implementacija takvog sustava vjerojatno bi se oslanjala na moderne metode strojnog učenja koje pak zahtijevaju bazu znanja za treniranje klasifikatora kako bi sustav bio sposoban razaznati pojedine znakove s visokom razinom točnosti.

Moderna kartografija oslanja se na slike iz zračne perspektive (satelitske snimke) što u ovom slučaju nije odgovarajuće. Prometne znakove je iz zraka teško razlučiti, a još teže klasificirati, uzevši u obzir da nisu zaklonjeni. Alternativa je vožnja prometnicama specijaliziranim vozilom koje tokom vožnje bilježi videosnimku ili slijed fotografija i GPS koordinate u određenom trenutku. U svrhu anotacije znakova, na ZEMRISu je u okviru međunarodnog projekta MASTIF razvijena aplikacija Marker koja u svojoj prvoj verziji nudi mogućnost reprodukcije više multimedijskih formata, alate za ručno označavanje i klasifikaciju znakova, te spremanje oznaka u standardnom formatu. Manualno označavanje znakova je monoton i dugotrajan postupak jer podrazumijeva da ljudski korisnik mora reproducirati svaki video, zaustaviti ga dok ugleda prometni znak, izabrati točnu klasu znaka iz izbornika svih mogućih klasa, te na kraju spremiti rezultate. Područje računalnog vida nudi efikasna rješenja za automatizaciju ovakvog posla. Radom na predmetu "Projekt 2009/2010" razvijene su komponente za detekciju, klasifikaciju i praćenje prometnih znakova.

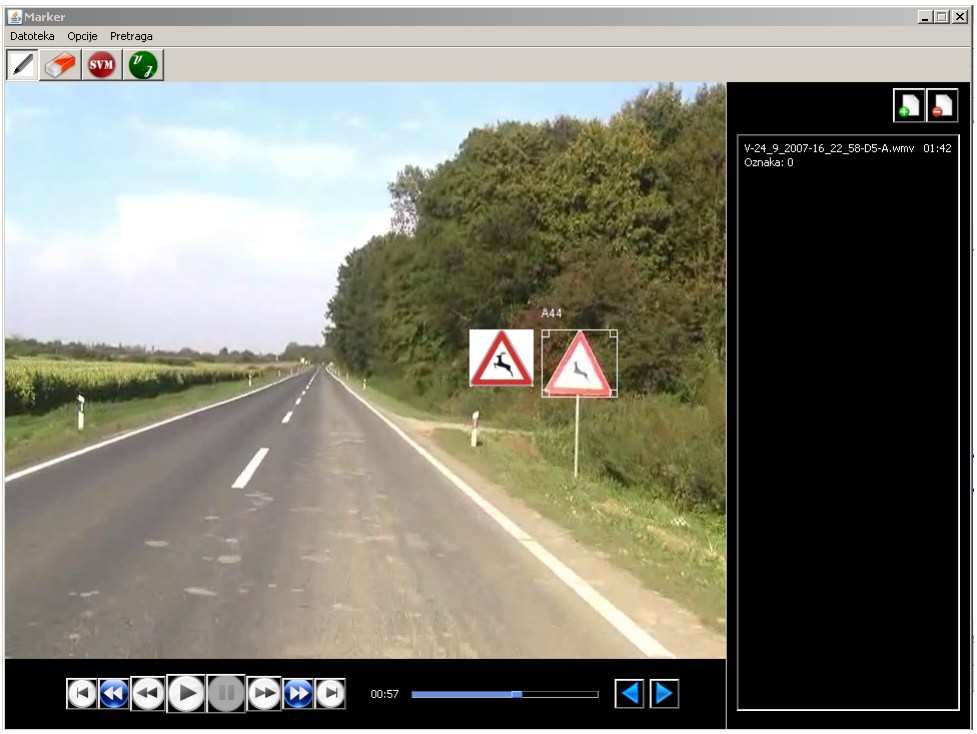
Ovaj se završni rad bavi integracijom tih komponenti u aplikaciju Marker, nadogradnjom korisničkog sučelja aplikacije kako bi se komponente mogle što efikasnije koristiti, te prekrajanjem izvornog modela Markera radi postizanja modularnosti, brzine, nadogradivosti i lakog održavanja. Rad se zasniva na programskoj implementaciji dijelova aplikacije Marker, ostvarenoj u programskom jeziku Java. Velik dio posla obavljen je korištenjem biblioteka Swing i AWT, specijaliziranih za izradu grafičkih sučelja. Također, popravljene su manje pogreške u nativnoj biblioteci koja sadrži algoritme za detekciju i praćenje prometnih znakova.

Rad počinje uvodom u arhitekturu aplikacije i analizom njenih bitnih dijelova. Ovaj se dio rada bazira na prvoj verziji Markera, prije ugradnje komponenti koje omogućavaju automatizaciju njegovog rada. Zatim se, u trećem poglavlju, opisuje model po kojem je napravljen paket za vanjske komponente, te mogućnosti koje pruža - dinamičko dodavanje komponenti u sustav, njihovu konfiguraciju, te učitavanje resursa koje koriste pri radu. Četvrto se poglavlje bavi ugradnjom detektora i klasifikatora u sustav. Razmatra se način dodavanja novih komponenti u prezentacijskom sloju aplikacije, te problemi fokusa grafičkih komponenata i mapiranja korisničkih akcija na tipkovnicu. Obrađuje se uvodni ekran aplikacije i napredni razredi biblioteke Swing programskog jezika Java koji su korišteni pri integraciji komponenata u Marker.

2. Marker

2.1. Najvažnije značajke aplikacije

Marker je aplikacija pisana u programskom jeziku Java koja služi za anotaciju prometnih znakova. Prvu verziju razvila je dipl.ing. Karla Brkić. Slika 2.1 predstavlja korisničko sučelje Markera. Osnovnu je funkcionalnost aplikacije moguće svrstati u tri skupine zadataka:



Slika 2.1: Korisničko sučelje Markera

2.1.1. Čitanje slijedova slika iz komprimiranih datotečnih formata

Kako bi korisnik mogao označavati znakove, aplikacija mora biti sposobna učitati slijedove slika iz izvornog medija gdje su znakovi snimljeni. Marker nudi učitavanje i

reprodukciju više multimedijskih formata:

Video *.wmv*, *.avi*

Slike *.jpeg*, *.jpg*, *.bmp*, *.gif*, *.png*

Za reprodukciju videa u stvarnom vremenu korištena je nativna biblioteka *libvs*. Alati za navigaciju videom nude mogućnost postupnog pomicanja videa na željeno mjesto (1 ili 10 sekundi naprijed ili natrag) ili pomak na određeni dio navigacijskom trakom. Radi bržeg korištenja, funkcije navigacije mapirane su na tipkovnicu.

2.1.2. Označavanje znakova

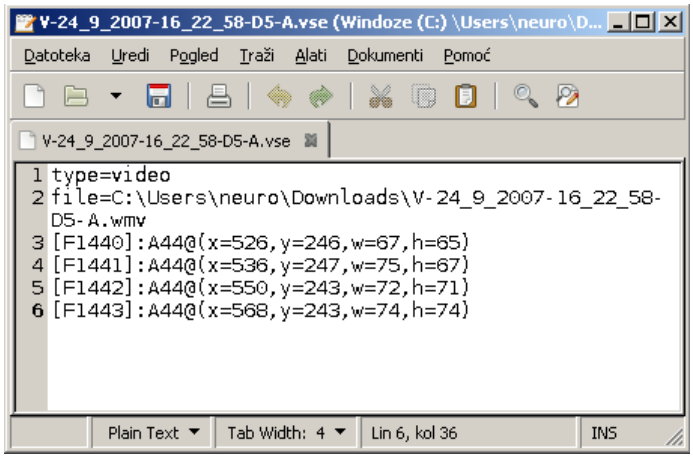
Marker je u svojoj prvoj verziji nudio manualni način označavanja prometnih znakova koji se svodi na sljedećih 6 koraka:

1. zaustavljanje videa
2. odabir alata za označavanje znakova
3. ucrtavanje pravokutnika mišem oko željenog znaka
4. odabir odgovarajuće klase znaka navigacijom kroz izbornik
5. ako je došlo do pogreške, odabir alata za brisanje, te klikom na oznaku njeno brisanje
6. ponovo pokretanje videa

Ove je korake bilo potrebno provesti za svaki znak u videu. Napredni algoritami za strojnu klasifikaciju znakova zahtijevaju označavanje znakova u svim slikama slijeda. Stoga je znakove potrebno *gusto* označavati. To znači da je u videu znak trebalo precizno označiti na svakoj slici gdje se nalazi, što je zadatak anotacije učinilo još težim i dugotrajnijim [Shotton et al. (2008)].

2.1.3. Spremanje oznaka

Opcija spremanja označenih znakova sprema oznake na tvrdi disk, iz videa u formatu *.vse*, a iz slijeda slika u formatu *.seq*. Sprema se točan *frame* u kojem je znak označen, te njegova klasa i koordinate pravokutnika koji ga okružuje. Slika 2.2 ilustrira sadržaj datoteke sa spremljenim anotacijama iz videa.



Slika 2.2: Datoteka sa spremljenim oznakama

2.2. Arhitektura Model-Pogled-Upravljač

Model-Pogled-Upravljač (*engl. Model View Controller*), u daljnjem tekstu MVC, trodijelni je model izgradnje aplikacija kojima je težište na interaktivnom korisničkom sučelju [Eckstein (2007b)]. MVC je osmislio Trygve Reenskaug 1979. Ilustracija MVC arhitekture prikazana je na slici 2.3. Prednost tog rješenja jest razdjeljenost glavnih elemenata sustava i obrazaca po kojima ti elementi međusobno surađuju:

Model predstavlja podatke i pravila za osvježavanje i postavljanje podataka.

Pogled čine klase grafičkog sučelja koje korisnik vidi i nad kojima može raditi akcije koje mijenjaju stanje modela.

Upravljač je dvosmjerna poveznica između modela i pogleda, on prevodi korisničke akcije u promjene stanja modela, i osvježava pogled sukladno s promjenama podataka u modelu.

2.3. Organizacija izvornog koda po paketima

U aplikacijama pisanim u programskom jeziku Java, standardna je praksa odvajanja srodne funkcionalnosti u *pakete*. Slijedi popis važnijih paketa u aplikaciji Marker zajedno s odgovarajućim opisima namjene i podpaketa.

`akcije` su zapravo međusloj koji povezuje korisnikove radnje u korisničkom sučelju i odgovarajuće metode pokretane u Kontroleru aplikacije. Radi modularnosti i nadogradivosti, sve one naslijeđuju apstraktnu klasu AkcijaSucelja koja definira standardno ponašanje akcija. Podpaketi `kontrole` i `alati` odnose

se na specifične dijelove sustava, gumbe za navigaciju medijem i alate povezane s komponentama koje automatiziraju posao.

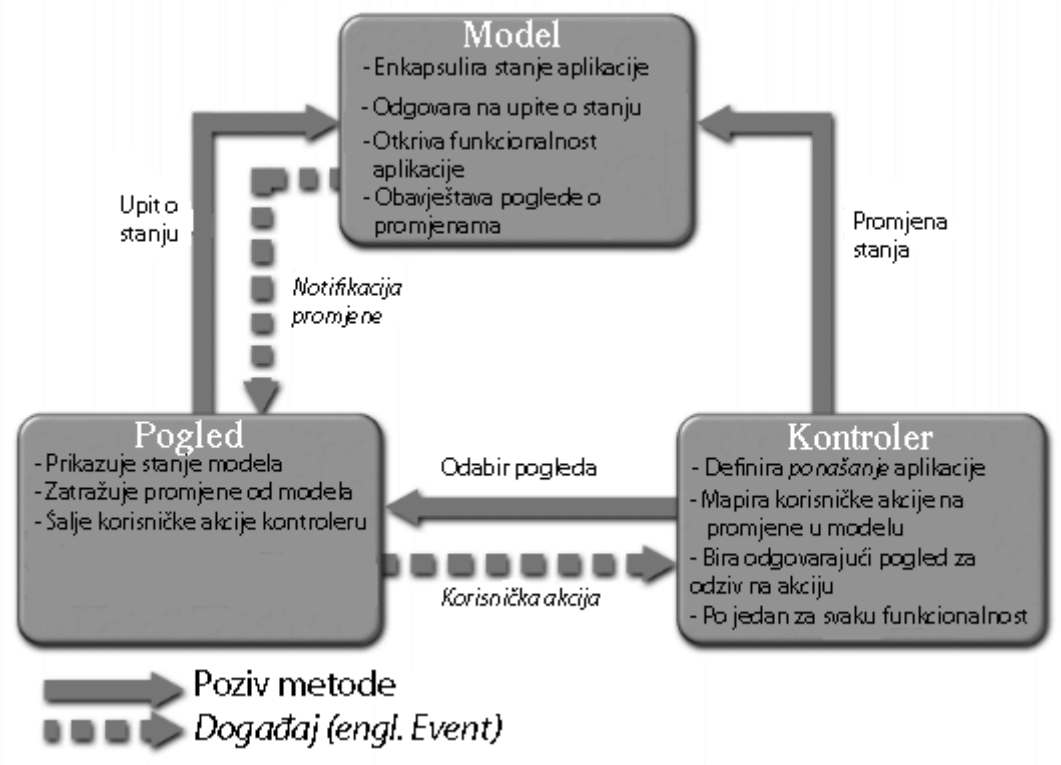
`gui` paket koji sadrži implementacije nadograđenih vizualnih komponenti biblioteke Swing. Ovdje se nalaze i podpaketi specifični za prikaz *playliste*, uvodnog ekrana i grafičkih klasa koje koriste komponente.

`izvori` sadrži klase za rukovanje medijima koje anotiramo. Ovdje se nalazi i sučelje prema nativnoj biblioteci `libvs.dll` koja reproducira video.

`komponente` predstavlja skup paketa u kojima su implementirane vanjske komponente Markera. Ovim ćemo se paketom baviti u idućim poglavljima.

`opcije` sadrži logiku izbornika i opcija koje nudi Marker. Opcije su postepeno nadograđivane uvođenjem vanjskih komponenti pa će i klase ovog paketa biti detaljno objašnjene kasnije.

`pravilnik`, `pretraga`, `util` sadrže klase za pomoć pri radu vizualnih komponenti, pretraživanje, te klase koje se nisu mogle svrstati u druge pakete.



Slika 2.3: Arhitektura Model-Pogled-Upravljač

2.4. Rad sa znakovnim konstantama

Znakovne konstante (*eng. String*) u većim aplikacijama nije poželjno *hardkodirati* tamo gdje se koriste. Bolja je praksa čuvanje konstanti u posebnim datotekama odakle ih je lako dobaviti ili promjeniti. To se posebno odnosi na nizove koji predstavljaju labele grafičkih objekata. Programski jezik Java podržava ovakav način rada s konstantama te pruža potrebnu podršku u vidu razreda `Properties` koji je namijenjen pamćenju korisničkih opcija i razreda `ResourceBundle` koji služi za pamćenje labela i ostalih konstanti. Obje klase rade s datotekama tipa *.properties*. Podaci su organizirani kao parovi *ključ-vrijednost*, te se dobavljaju i mijenjaju pomoću ključa [Deitel i Deitel (2004)]. Marker sadrži tri datoteke za rad sa znakovnim konstantama:

`Messages` je razred koji čuva nizove znakova kao što su poruke korisniku, labele (naslovi grafičkih komponenti). Nizove se pomoću ključa dohvaća statičkom metodom *`getString(String key)`* i postavlja metodom *`setString(String key, String value)`*.

`Options` funkcionira na sličan način kao i `Messages`, a namjena joj je pamćenje opcija navedenih u meniju *Opcije* u korisničkom sučelju Markera. Podaci se čuvaju u datoteci *options.properties*

`Properties` sadrži globalne postavke rada programa. Ovdje se pamti zadnje korištena kaskada detektora, zadnje otvoreni direktorij kod izbora izvornog medija (izbornik *File->Open*), mapiranje tipki na tipkovnici. Datoteka *global.properties* sadrži ove podatke.

3. Rad s vanjskim komponentama

Komponente su nadogradnje Markera koje ubrzavaju, poboljšavaju i olakšavaju njegov rad. Unaprijed nije poznato koliko će komponenata aplikacija sadržavati, hoće li biti više specijaliziranih komponenti sa jednakim područjem rada. Primjerice, jedan od mogućih scenarija jest ugradnja dva klasifikatora, po jedan za trokutaste i kružne znakove koji moraju funkcionirati zasebno, ali sustav mora s njima raditi na unificiran način [Bučar et al. (2010a)]. Podrazumijeva se da je komponente moguće dinamički priključivati i isključivati iz sustava. Gotovo sve komponente sadrže i elemente grafičkog sučelja pomoću kojih se njima upravlja. Potrebno je omogućiti učitavanje resursa pojedinih komponenti, te konfiguracija njihovog rada pomoću opcija tijekom rada programa. Poželjno je na početku rada programa, tokom učitavanja resursa koji su potrebni za rad komponenti, korisniku prikazati koliko se resursa učitalo, te koliko ih je još preostalo za učitavanje. Radi bržeg učitavanja, poželjno je pokrenuti novu dretvu posebno za učitavanje resursa, kako bi se paralelno moglo inicijalizirati korisničko sučelje glavnog programa. Sve akcije rada s komponentama, osim odabira njihovih specifičnih opcija potrebno je zbog brzine i lakoće korištenja mapirati na miš i tipkovnicu računala.

Napredni i modularni programski sustavi pisani u višim objektno orijentiranim jezicima poput Jave osobine nadogradivosti i modularnosti ostvaruju implementacijama koje omogućavaju ponovno korištenje gotovih komponenti te fleksibilna sučelja za ugradnju novih komponenti. Glavni mehanizmi zaslužni za te osobine su polimorfizam i naslijeđivanje. Izgradnja modela za vanjske komponente išla je u smjeru maksimalnog iskorištenja tih mehanizama.

Programski jezik Java nudi dva osnovna jezična konstrukta za ostvarenje koncepta nadogradnje bez promjene:

sučelja prestavljaju potpuno apstraktne razrede s prototipovima metoda koje se implementiraju tek u razredima koji ih nasljeđuju i koriste. Sučelja omogućavaju rukovanje različitim objektima koji ga implementiraju na jednak način (poli-

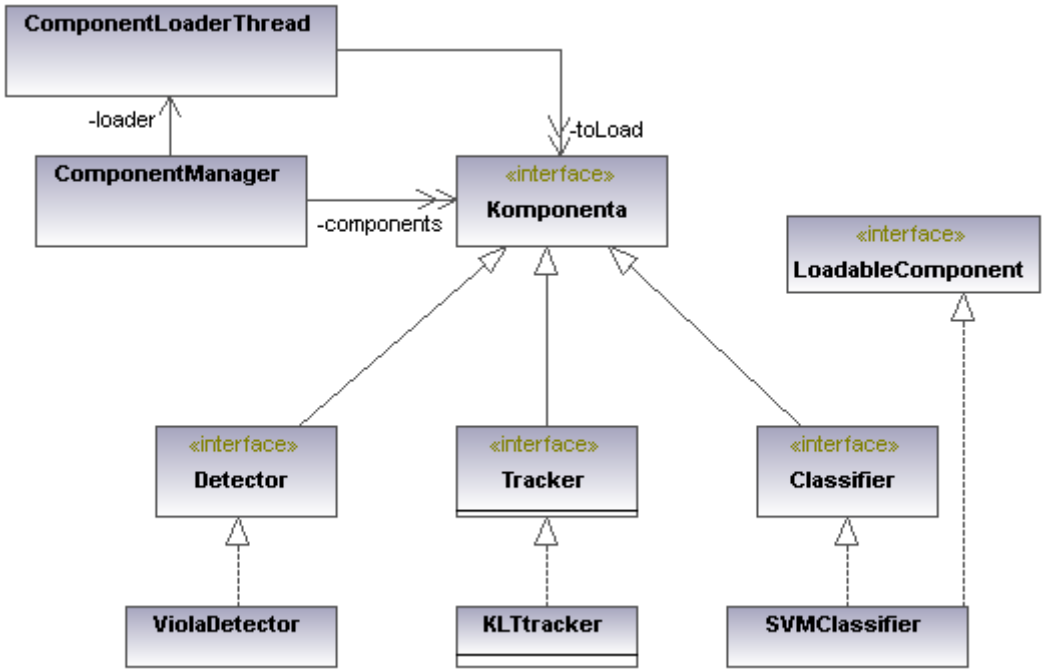
morfizam). Kako bi razred implementirao neko sučelje mora nadopuniti sve njegove metode.

apstraktni razredi su jednu razinu apstrakcije ispod sučelja. Oni pružaju standardnu implementaciju odabranih metoda (neke mogu ostati bez implementacije, ali im se mora dodati ključna riječ `abstract`). Kao i za sučelja, vrijedi da se takve klase ne mogu instancirati, već je potrebno naslijediti apstraktni razred te nadopuniti implementaciju njegovih metoda.

3.1. Hijerarhija komponenata

Hijerarhijska struktura objekata koji dijele funkcionalnost dobar je način izgradnje sustava čiji se dijelovi mogu ponovno koristiti. Postiže se fleksibilnost rukovanja objektima, te na višim razinama apstrakcije omogućava da drugi dio sustava gleda na objekte koji su dijelovi hijerarhije na manje specifičan način (npr. da se na specifični `Detector` gleda kao na objekt tipa `Komponenta`).

UML dijagram cijele hijerarhije prikazan je na slici 3.1.



Slika 3.1: Hijerarhija komponenti

Vrh hijerarhije čini sučelje `Komponenta` koje nema nikakvih metoda. Svrha takvog *nazivnog*, praznog sučelja jest upravo mogućnost baratanja svim komponentama

korištenjem polimorfizma [Deitel i Deitel (2004)]. Tri su osnovne vrste komponenti koje je potrebno ugraditi u sustav, te one nasljeđuju razred `Komponenta`. One su također modelirane kao sučelja.

`Detector` služi za detekciju znakova na slici. Nakon što pokrenemo ovu komponentu, ona vraća koordinate pravokutnika koji opisuju detektirane znakove na slici.

```
public interface Detector extends Komponenta{

    public ArrayList<Detection> detectInFrame (BufferedImage frame, LibMastifInterface lm);
    public void reconfigure(String cascadeName) throws IllegalAccessException;

}
```

Slika 3.2: Izvorni kod sučelja `Detector`

`Classifier` raspoznaje klasu prometnog znaka te vraća skupinu najvjerojatnijih kandidata za točnu klasu, koju korisnik mora odabrati.

```
public interface Classifier extends Komponenta{

    public ArrayList<Sign> classifySingle(BufferedImage buffImage);
    public HashMap<BufferedImage, String> classifyBulk(ArrayList<BufferedImage> images);

}
```

Slika 3.3: Izvorni kod sučelja `Classifier`

`Tracker` je komponenta za praćenje detektiranog prometnog znaka kroz više slijednih slika ili *frame*ova u videu. Ona se inicijalizira nad jednim znakom te vraća koordinate pravokutnika koji ga opisuje u narednim slikama.

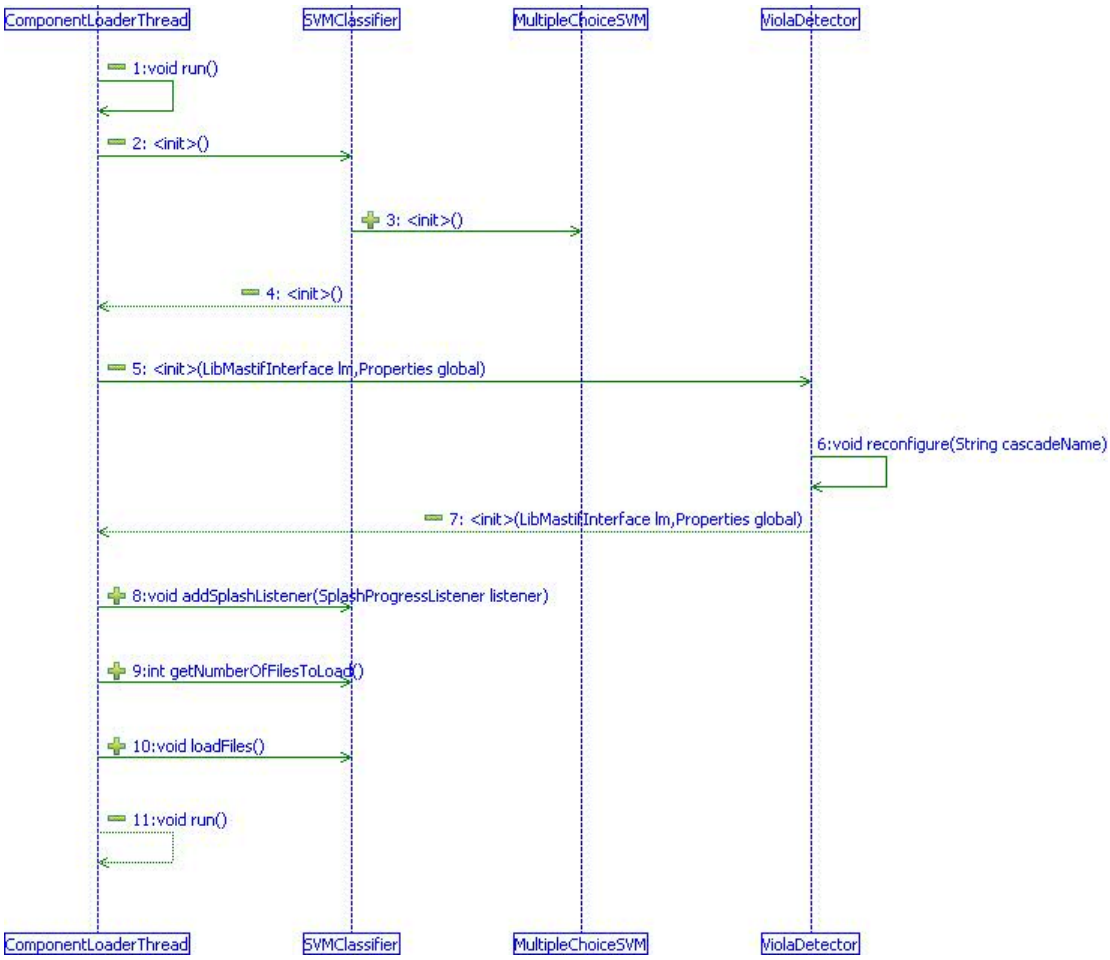
```
public interface Tracker extends Komponenta{

    public void initialize(LibMastifInterface lmi,
            BufferedImage initialFrame, ArrayList<Point> pointsToTrack );

    public void addTrackRefs(ArrayList<Point> references);

    public ArrayList<Detection> track(BufferedImage nextFrame,
            LibMastifInterface lmi);
}
```

Slika 3.4: Izvorni kod sučelja `Tracker`

Rukovanje komponentama

Tijekom rada, Marker mora na raspolaganju imati više inicijaliziranih komponenata [Bučar et al. (2010b)]. Poželjno je da komponente učitaju sve što im treba za rad

kako se o tome ne bi morao brinuti kod koji ih pokreće. Također, potrebno je voditi računa o tome da se komponente stvore samo jednom, jer je učitavanje njihovih podataka vremenski i prostorno skupa operacija. Kao rješenje se nameće razred koji će služiti kao centralna točka za rukovanje komponentama, `ComponentManager`. U vidu programske realizacije tog razreda, najelegantnija mogućnost jest korištenje oblikovnog obrasca *jedinstveni objekt* kako bi se spriječilo višestruko učitavanje istih komponenti. Korištenjem sučelja komponenti stvaramo pristupne točke pojedinim vrstama komponenti na način da rukovoditelj ne zna koju točno implementaciju pojedine komponente čuva. Ako se doda nova vrsta klasifikatora, rukovoditelja neće biti potrebno mijenjati. `ComponentManager` prilikom inicijalizacije pokreće učitavanje komponenti u posebnoj dretvi `ComponentLoaderThread`. Ilustraciju rada dretve u obliku sekvencijskog dijagrama prikazuje slika 3.5.



Slika 3.5: Sekvencijski dijagram dretve `ComponentLoaderThread`

Dretva za učitavanje resursa radi paralelno s Markerovom glavnom dretvom koja učitava korisničko sučelje. `ComponentLoaderThread` stvara uvodni ekran aplikacije te po redu inicijalizira komponente. Ako neka od komponenti implementira sučelje `LoadableComponent`, to znači da učitava dodatne podatke, te je dretva povezuje s uvodnim ekranom. Korisnik tokom pokretanja aplikacije na uvodnom ekranu vidi koja komponenta trenutno učitava podatke te koliko je datoteka preostalo do kraja. Uvodni ekran dobiva informaciju o broju preostalih komponenti u stvarnom vremenu. Ta je funkcionalnost ostvarena preko oblikovnog obrasca *Preglednik* tako da komponenta koja implementira sučelje `LoadableComponent` mora na sebe povezati `SplashScreenListener` koji na njoj čeka događaje i osvježava uvodni ekran [Cooper (1998)]. Odnos razreda koji rukovodi komponentama, dretve za učitavanje komponenti, te razreda za enumeraciju komponenti kojima su potrebni dodatni podaci vidljiv je na slici 3.1.

Konkretne klase komponenti implementiraju odgovarajuće sučelje. Slijedi detaljniji prikaz načina njihovog rada.

ViolaDetector

Razred `ViolaDetector` implementira sučelje `Detector`, te koristi nativnu biblioteku *libmastif.dll* kako bi mogao pozivati algoritam za detekciju znakova. Algoritam radi po principu *kaskade boostanih Haarovih klasifikatora*, a osmislili su ga Paul Viola i Michael Jones. Viola-Jones algoritam koristi okno kojim više puta prolazi kroz cijelu sliku. Na svakoj se poziciji okna pokreće klasifikator, a ako on vrati pozitvan odgovor, prijavljuje se da je traženi objekt detektiran. *Kaskada* predstavlja lanac klasifikatora gdje pojedini klasifikator istražuje manji skup značajki na oknu. Ovaj pristup je bolji od korištenja jednog klasifikatora s velikim brojem značajki, jer lanac djeluje kao serijski spoj, te vrlo brzo eliminira dijelove slike gdje nema prometnih znakova. Marker u tekućoj verziji nudi tri skupa kaskada dobivenih strojnim učenjem klasifikatora. Kaskade su specijalizirane za detekciju klase trokutastih prometnih znakova.

Rad razreda `ViolaDetector` svodi se na tri zadatka:

inicijalizacija detektora - Razred u konstruktoru prima objekt `LibMastifInterface`, kao poveznicu s nativnom bibliotekom, te objekt `Properties` iz kojeg se dobavlja kaskada korištena u zadnjoj sesiji rada s programom. Algoritam se inicijalizira metodom

`libMastifCreate(Messages.`

`getString("Detector.algorithm"));`

sučelja `LibMastifInterface`. Nakon inicijalizacije učitavaju se kaskade spremljene u direktoriju *komponente/kaskade*. Učitavanje obavlja posebna statička metoda `Util.ucitajKaskade()`. Na kraju se algoritam konfigurira za rad s kaskadom korištenom u zadnjoj sesiji.

rekonfiguracija kaskade - metoda

```
public void reconfigure(String cascadeName)
throws IllegalAccessException
```

konfigurira algoritam za rad s kaskadom naziva `cascadeName`. Ako takva kaskada ne postoji ili je algoritam ne može učitati, metoda baca iznimku.

pokretanje detekcije - metoda `detectInFrame` kao argument očekuje objekt tipa `LibMastifInterface` nad kojim pokreće algoritam te sliku sadržanu u objektu `BufferedImage` u kojoj je potrebno detektirati prometne znakove. Ako algoritam detektira objekte na slici, vraća listu koordinata pravokutnika koji predstavljaju detekcije, kako bi ih komponente korisničkog sučelja mogle iscrtati. Rad ovih komponenti bit će detaljnije analiziran u narednom poglavlju.

SVMClassifier

Klasifikator na temelju značajki prikupljenih treningom na skupovima pozitivnih i negativnih primjera za učenje svrstava objekte u klase. Konkretna implementacija klasifikatora napisana je u programskom jeziku Java, te uključena u Marker (paket `komponente.svm`), odakle se direktno koristi pri radu programa. Bazira se na konceptu *stroja s potpornim vektorima*, modificiranom da radi s više klasa istovremeno. Pri raspoznavanju prometnih znakova binarna klasifikacija u standardnom obliku predstavlja loš izbor, jer bi se sve klase trebale uspoređivati međusobno. Što je više klasa uključeno u bazu znanja, to je više kombinacija potrebno isprobati. Bolji pristup je stvaranje klase *ALL* s kojom se uspoređuju ostale klase. *ALL* sadrži značajke svih klasa znakova iz baze znanja. Na ovaj se način i dalje koristi binarna klasifikacija, ali se broj usporedbi znatno smanjio.

Konkrentni klasifikator `SVMClassifier` nasljeđuje sučelje `Classifier` te implementira metodu `ArrayList<Sign> classifySingle(BufferedImage image)`. Članski objekt koji pokreće sam algoritam je tipa `MultipleChoiceSVM`, te klasifikator služi kao omotač oko njega. Metoda vraća listu znakova rangiranih

po kvaliteti klasifikacije. Znakovi se prezentiraju korisniku te on odabire točan znak. `SVMClassifier` implementira i sučelje `LoadableComponent`, jer pri inicijalizaciji programa učitava naučene značajke za klasifikaciju.

KLTtracker

Algoritam za praćenje dio je nativne biblioteke *libmastif.dll*. Razred `KLTTracker` inicijalizira sučelje za praćenje `LibMastifInterface` predajući mu inicijalnu sliku. Potom se u metodi `addTrackRefs` dodaju referentne točke objekata koje će algoritam pratiti. Kako Marker reproducira video, komponenti šalje daljnje slike metodom `track(BufferedImage nextFrame)` te `KLTTracker` vraća koordinate praćenih objekata. Marker po tim koordinatama iscrtava pravokutnike oko praćenih objekata u videu. Ako se pojavi potreba dodavanja novih referentnih točaka u toku praćenja jednog objekta, dozvoljeno je koristiti metodu `addTrackRefs` kako bi se paralelno pratilo još objekata.

4. Povezivanje vanjskih komponenti i bazne aplikacije

4.1. Korisnički alati

Pokretanje komponenti odvija se pomoću skupa korisničkih alata, vidljivih iznad podprozora s videom u aplikaciji Marker. Alati su integrirani u arhitekturu *Model-pogled-upravljač* u svim slojevima. Marker pamti trenutno izabrani alat *stanjem*. Ako se dogodi akcija na prozoru gdje se reproduciraju alati, pokreće se akcija tekućeg alata. Ovo je izvedeno kombinacijom oblikovnih obrazaca *Promatrač* i *Naredba*. Alati se inicijaliziraju pomoću obrasca *Metoda tvornica* [Cooper (1998)]. Korisnik bira alat na alatnoj traci, model pamti koji je alat odabran, a upravljač interpretira korisnikove radnje s alatima u odgovarauće akcije nad modelom. Postoje sučelja `Alat` i `Akcija`, kako bi sustav bio nadogradiv bez promjene starog koda. U prvoj verziji Marker je nudio dva korisnička alata, alat za označavanje kojim se crta pravokutnik oko znaka koji želimo označiti, te alat za brisanje koji briše pravokutnike zajedno s njihovim oznakama.

Prilikom ugradnje novih komponenti cilj je maksimalno iskoristiti postojeće nadogradive dijelove koda. Dodani su alati za detekciju i klasifikaciju, te su im dodane akcije kako bi mogli mijenjati stanje modela, a time i prezentacijski sloj aplikacije.

4.2. Ugradnja klasifikatora

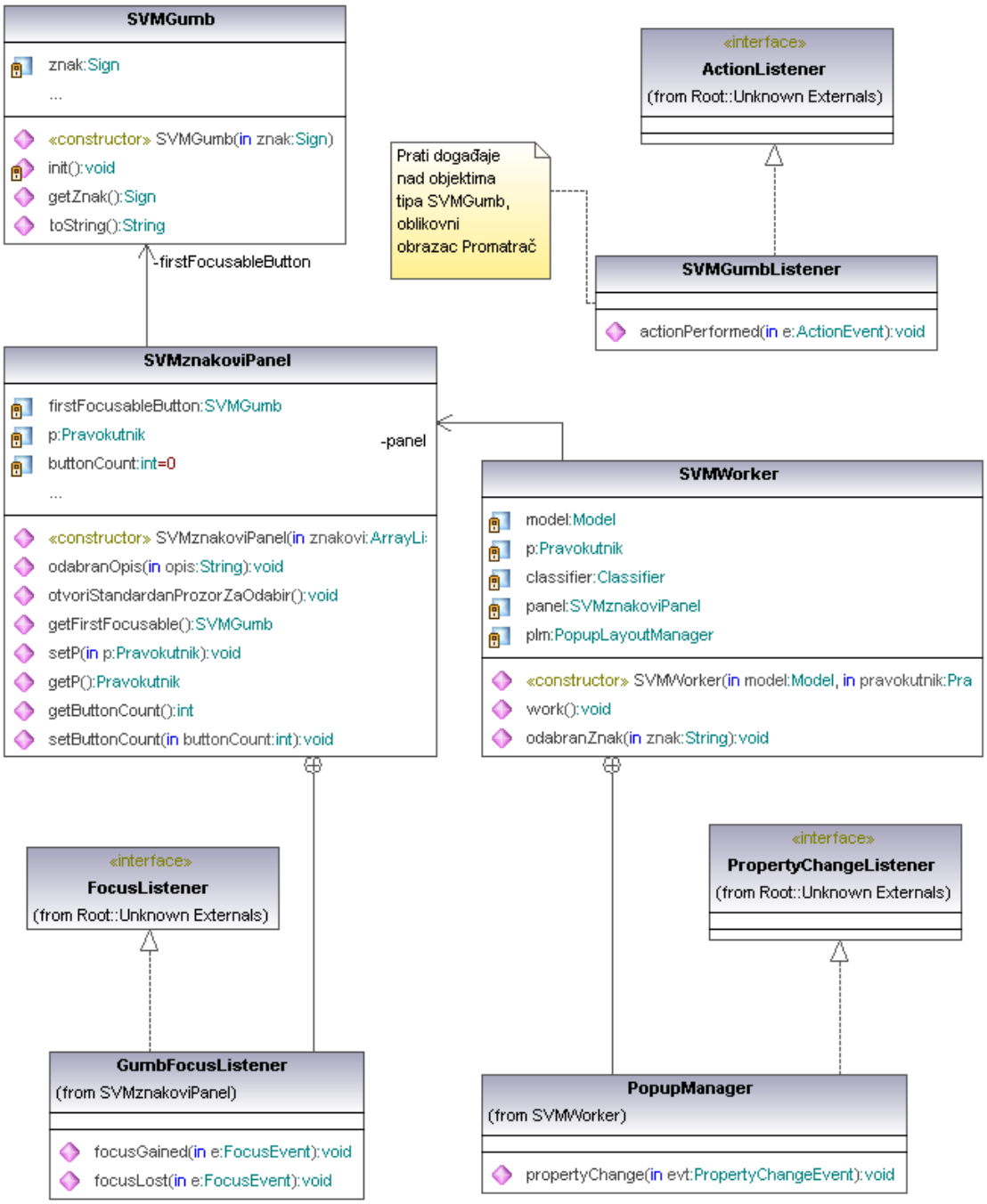
4.2.1. Vizualne komponente

Alat `AlatSVM` pokreće proces raspoznavanja na zadanom objektu. Programski kod koji radi s vizualnim komponentama iz grafičke biblioteke Swing smješten je u razred `SVMWorker`. Objekt `SVMWorker` prvo *izreže* (*eng. crop*) regiju nad kojom će se

provesti klasifikacija korištenjem posebnog razreda `ImageCropper`, te poziva klasifikator predajući mu tu regiju kao objekt tipa `BufferedImage`. Klasifikator pokreće algoritam preko nativne biblioteke LibMastif te vraća listu prepoznatih znakova koje je algoritam pronašao, poredanih po preciznosti klasifikacije. Korisniku je potrebno ponuditi opciju izbora odgovarajućeg prometnog znaka među onima koje pronašao klasifikator. Ostvarivanje funkcionalnosti robusnog, jednostavnog izbornika koji se pojavljuje na određenom dijelu prozora (uz odabranu regiju) pokazao se relativno teškim problemom. Pokušaj rješenja svih navedenih zahtijeva prikazan je na slici 4.1.

Korištenje grafičke biblioteke *Swing* programskog jezika Java svodi se na instanciranje gotovih komponenata koje se pozicioniraju u prozoru jednostavnim korištenjem podržanih *layout managera*, koji pružaju primitivno postavljanje vizualnih komponenti, obično ih pozicionirajući relativno u odnosu na položaj drugih komponenti. *Apsolutno* pozicioniranje znači da se komponente smještaju na određene koordinate, što je nerješivo korištenjem layout managera. Problem se još više komplicira činjenicom da se izbornici znakova moraju pojavljivati iznad drugih komponenata (moguće i postojećih izbornika). Umetanje izbornika u samu scenu loš je pristup jer se izbornici dinamički stvaraju i brišu. Također, potrebno je omogućiti korisniku rukovanje izbornicima pomoću tipkovnice, kako bi se ubrzao rad. Sva ova funkcionalnost ostvarena je korištenjem nekoliko specijaliziranih grafičkih razreda.

Razred `SVMZnakoviPanel` predstavlja izbornik u kojem se nalaze gumbi sa znakovima koje je klasifikator vratio. Korisnik može odabrati neki od ponuđenih znakova ili opciju da mu se pokaže izbornik svih postojećih znakova u slučaju da klasifikator potpuno promaši raspoznavanje. Za svaki gumb veže se akcija koja povezuje gumb sa razredom `PopupManager`, privatnim razredom klase `SVMWorker`. Mehanizam je izveden korištenjem oblikovnog obrasca *Promatrač*. Za svaki gumb veže se zasebni objekt tipa `SVMGumbListener`. Standardna biblioteka Jave sadrži sučelje `ActionListener` kako bi se Promatrač mogao lakše implementirati, te ga `SVMGumbListener` nasljeđuje i implementira metodu `public void actionPerformed()` kojom definira akcije koje je potrebno poduzeti kada korisnik pritisne gumb. Kada se to dogodi, `SVMWorker` postavlja oznaku odabranog znaka na sliku te u strukturu oznaka vezanih uz medij koji anotiramo dodaje novu oznaku. Postavljanje oznake vidljivo je na slici **??**.



Slika 4.1: Dijagram logičkih i vizualnih razreda komponente klasifikatora

4.2.2. Podrška za tipkovnicu

Gusto označavanje prometnih znakova podrazumijeva anotiranje znakova u svakom dijelu videa ili slijedu slika gdje je znak vidljiv. Korisničke radnje pokretanja detektora i klasifikatora, te potvrda klasificiranog znaka ili odabir točnog znaka traju relativno dugo ako je korisnik osuđen na rad mišem. Mapiranje samih akcija *detekcije* i *klasi-*

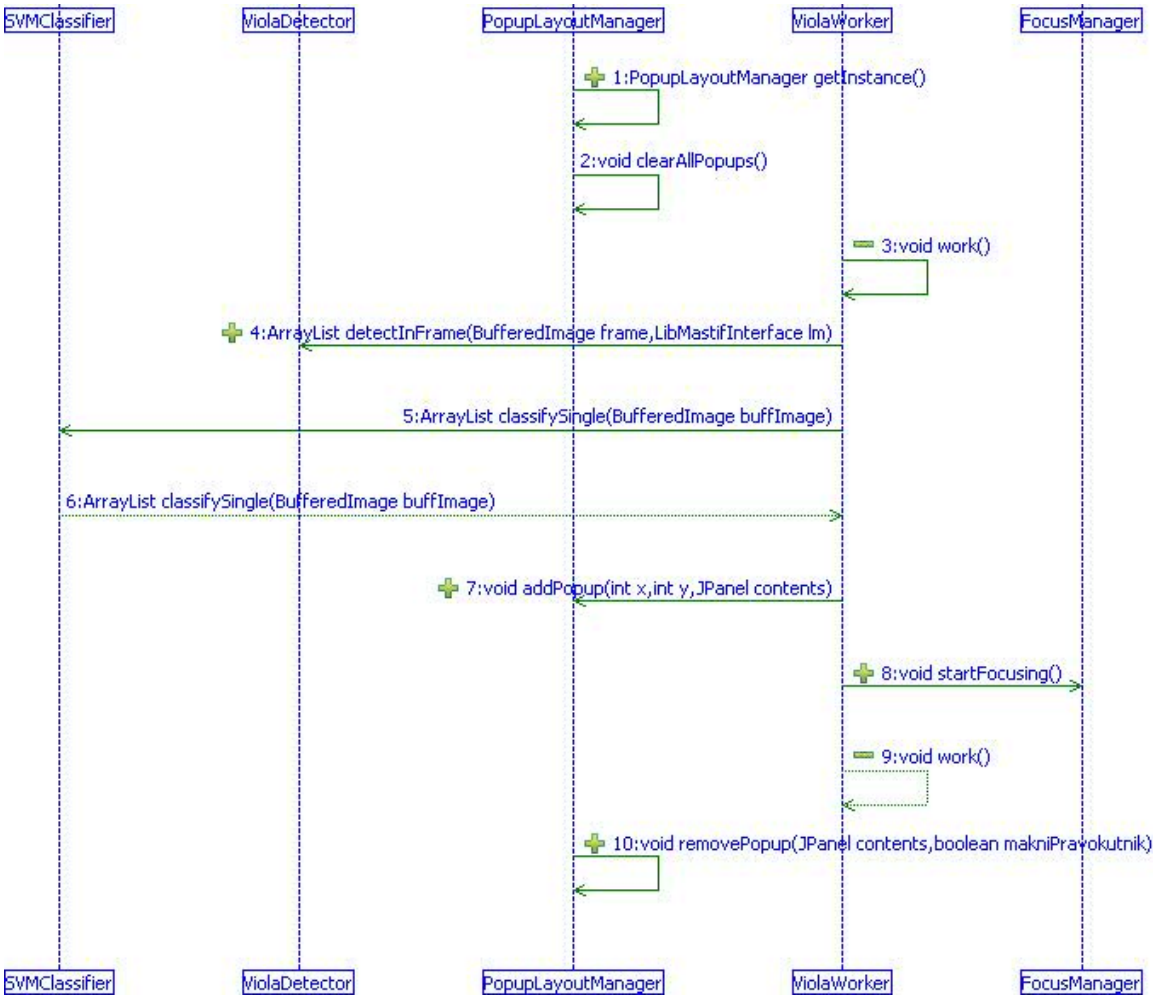
Slika 4.2: Postavljanje oznake na klasificirani znak

fikacije na tipkovnicu relativno je jednostavan zadatak [Eckstein (2007a)]. Podrška za rad s tipkovnicom uvodi nove probleme pri realizaciji izbornika za odabir klasificiranih znakova.

Komponenta za detekciju može u jednoj slici vratiti nula, jednu ili više regija gdje bi mogao biti prometni znak. Ako postoji više regija, moguće je da se one nalaze prostorno blizu, te će se njihovi izbornici preklapati. Uvodimo razred `PopupLayoutManager` koja se brine o dodavanju izbornika u scenu, te postavljanja svakog izbornika u poseban sloj kako bi se mogli pravilno iscrtati. Budući da je u cijeloj aplikaciji dovoljan jedan ovakav objekt, implementiran je kao oblikovni obrazac *Jedinstveni objekt*, u slučaju da nove komponente u budućnosti zatrebaju raslojene izbornike [Cooper (1998)]. Kada bi postojale dvije instance, ne bi mogle voditi zajedničku evidenciju slojeva na kojima se nalaze izbornici, te bi moglo doći do situacije da se izbornici crtaju u istom sloju. Time bi se dogodila pogreška, a u najgorem slučaju i rušenje cijele aplikacije. `ViolaWorker`, dio komponente za detekciju, delegira dodavanje i micanje izbornika `PopupLayoutManageru` koji zna koji su slojevi slobodni te raspoređuje izbornike. Ovaj je mehanizam prikazan sekvencijskim dijagramom na slici 4.3.

Kao što je spomenuto, izbornik čini objekt tipa `SVMZnakoviPanel`, podrazred `JPanel` iz biblioteke Swing. Roditeljski razred `JPanel`-a je `Component`. Kako bi se unutar izbornika radilo s tipkovnicom, elementi izbornika (objekti tipa `SVMGumb`) moraju biti fokusirani. Podrazredi razreda `Component` implicitno nasljeđuju sučelje `Focusable` te prema tome mogu raditi s fokusom. Željeni tijek rada s izbornikom je sljedeći

1. Kada se izvrši detekcija, izbornici moraju biti poredani po slojevima tako da zadnje detektirani objekt bude u najvišem sloju (neprekriven drugim izbornicima)



Slika 4.3: Sekvencijski dijagram rada s izbornicima

2. Izbornik u najvišem sloju traži fokus kako bi se u njemu moglo raditi tipkovnicom

3. Tipkom `TAB` moguće je kružiti izbornikom, odnosno njegovim gumbima, tako da se fokus seli s gumba na gumb, ali pod uvjetom da fokus ne prijeđe na idući izbornik

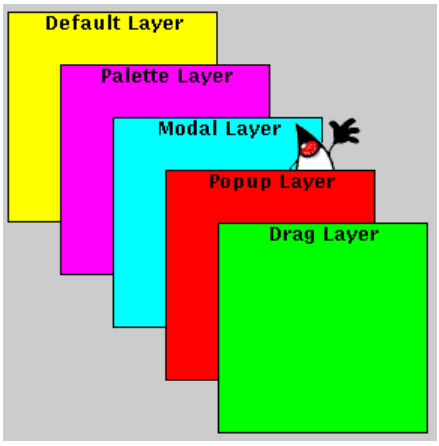
4. Tipkom `SPACE` moguće je pritisnuti gumb s jednakim efektom kao da na njega kliknemo mišem, nakon čega se izbornik miče sa scene

5. Tipkom `ESC` moguće je maknuti izbornik bez odabira ijednog ponuđenog znaka

6. Ako se trenutni izbornik miče, te ako postoji neki ispod njega, on dobiva fokus

7. Ciklus se ponavlja od koraka 2 sve dok postoje izbornici u sceni

Slika 4.4 prikazuje model razreda `LayeredPane`, te općenito kako izgleda *slojevita* grafička komponenta. Svaki `LayeredPane` objekt podijeljen je u više slojeva, a slojevi pripadaju kategorijama. Najniži sloj je onaj koji su uvijek vidi, tj. onaj na koji se komponente postavljaju bez dodatnih postavki. Kako bi se nešto postavilo u viši sloj, potrebno je pri dodavanju komponente u onu koja podržava slojeve navesti broj sloja. Marker izbornike postavlja u slojeve kategorije PopupLayer, interno vodeći računa o tome koji su slojevi zauzeti.



Slika 4.4: Pregled mogućih učinaka razreda `LayeredPane` iz biblioteke Swing

Slika 4.5 prikazuje više izbornika koji su preklopljeni (jedan se nalazi u sloju iznad drugog). Na slikama 4.6 i 4.7 prikazano je kako gumb dobiva fokus, te kako fokus prelazi na daljnje gumbe. Fokusirani gumb okružen je crvenim obrubom.

Slika 4.5: Preklapanje izbornika

4.2.3. Fokus vizualnih komponenata

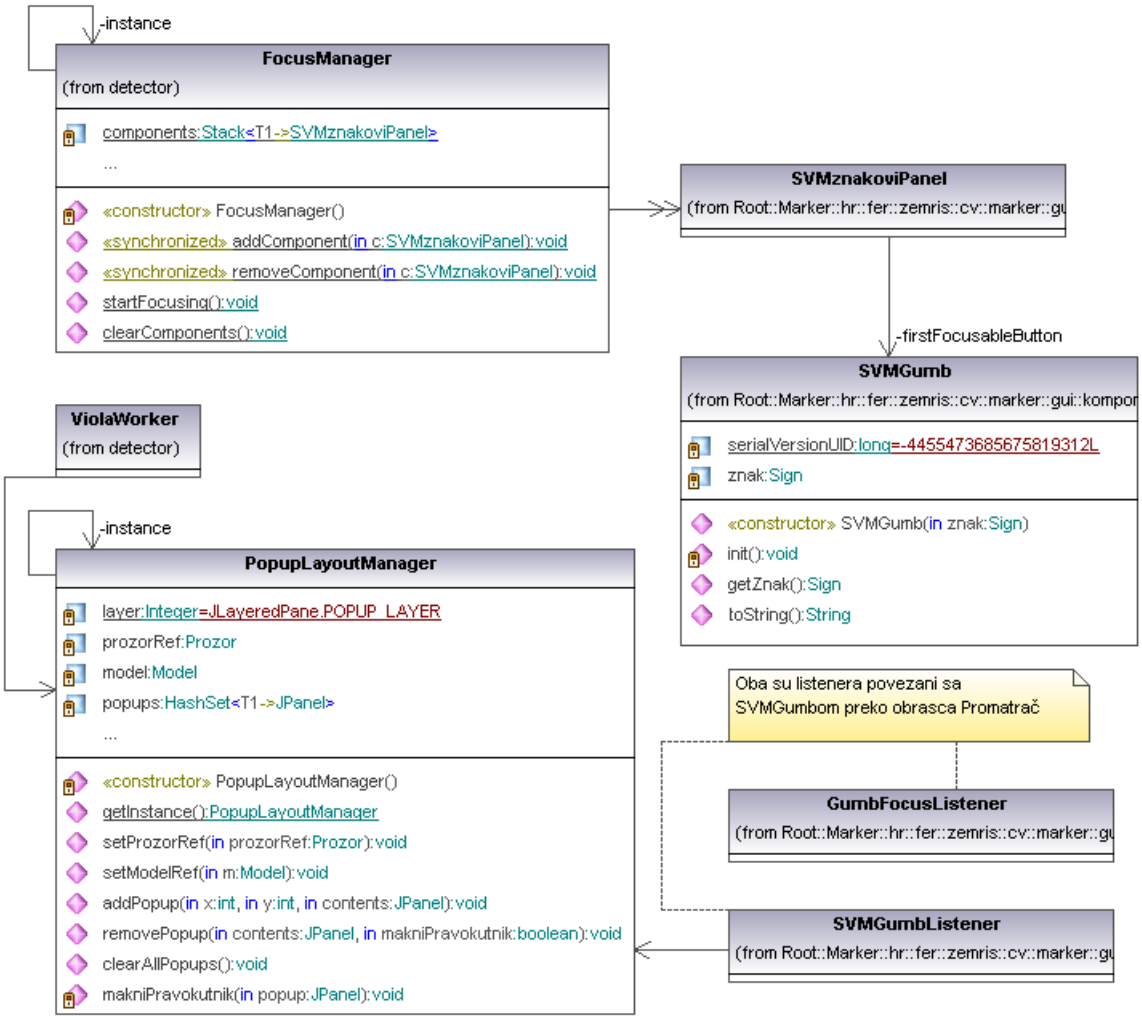
Dio aplikacije direktno povezan s rješavanjem problema fokusa izbornika i njihovih elemenata prikazan je na slici 4.8. U okviru biblioteke Swing sustav za baratanje fokusom grafičkih razreda napravljen je po *modelu upravljanom događajima* (*eng. event*

Slika 4.6: Prikaz fokusiranog gumba u izborniku

Slika 4.7: Promjena fokusa gumba tipkovnicom

driven model). Model se bazira na sposobnosti odašiljanja događaja kada su ispunjeni određeni uvjeti. Tako komponente šalju `FocusEvent` kada im virtualni stroj dodijeli ili oduzme fokus. U samom objektu `FocusEvent` sadržani su detalji vezani uz događaj - objekt koji ga je odaslao i prezican tip događaja (npr. fokus dobiven, fokus izgubljen) predstavljaju najvažnije detalje. Ako želimo da naš razred reagira na događaje tipa `FocusEvent`, on mora implementirati sučelje `FocusEventListener` te se dodati kao slušatelja na ciljanu komponentu. Većina grafičkih komponenata biblioteke Swing pruža mogućnost slušanja događaja vezanih uz fokus.

Razred `FocusManager` brine se o fokusu među izbornicima, spremajući ih na interni stog te fokusirajući onu komponentu koja se nalazi na vrhu stoga. Kako bi se fokusirale samo valjane komponente (izbornici), one implementiraju sučelje `Focusable` [Adamson i Marinacci (2005)]. `PopupLayoutManager` pri svakom postupku dodavanju i micanju izbornika sa scene dodaje ili miče taj izbornik iz `FocusManagera`. Svaki se izbornik interno brine o fokusu svojih elemenata (objekata tipa SVMGumb), djelujući kao korijen fokusnog ciklusa (*focus cycle root*). Korijen dopušta alterniranje fokusa samo među komponentama koje sadrži. Ovaj mehanizam sprječava *bježanje* fokusa iz jednog izbornika u drugi dok korisnik pritiskom tipke `TAB` želi prijeći sa zadnjeg gumba u izborniku natrag na prvi, odnosno, omogućava kruženje gumbima sve dok se ne dođe do onog kojeg se želi potvrditi.



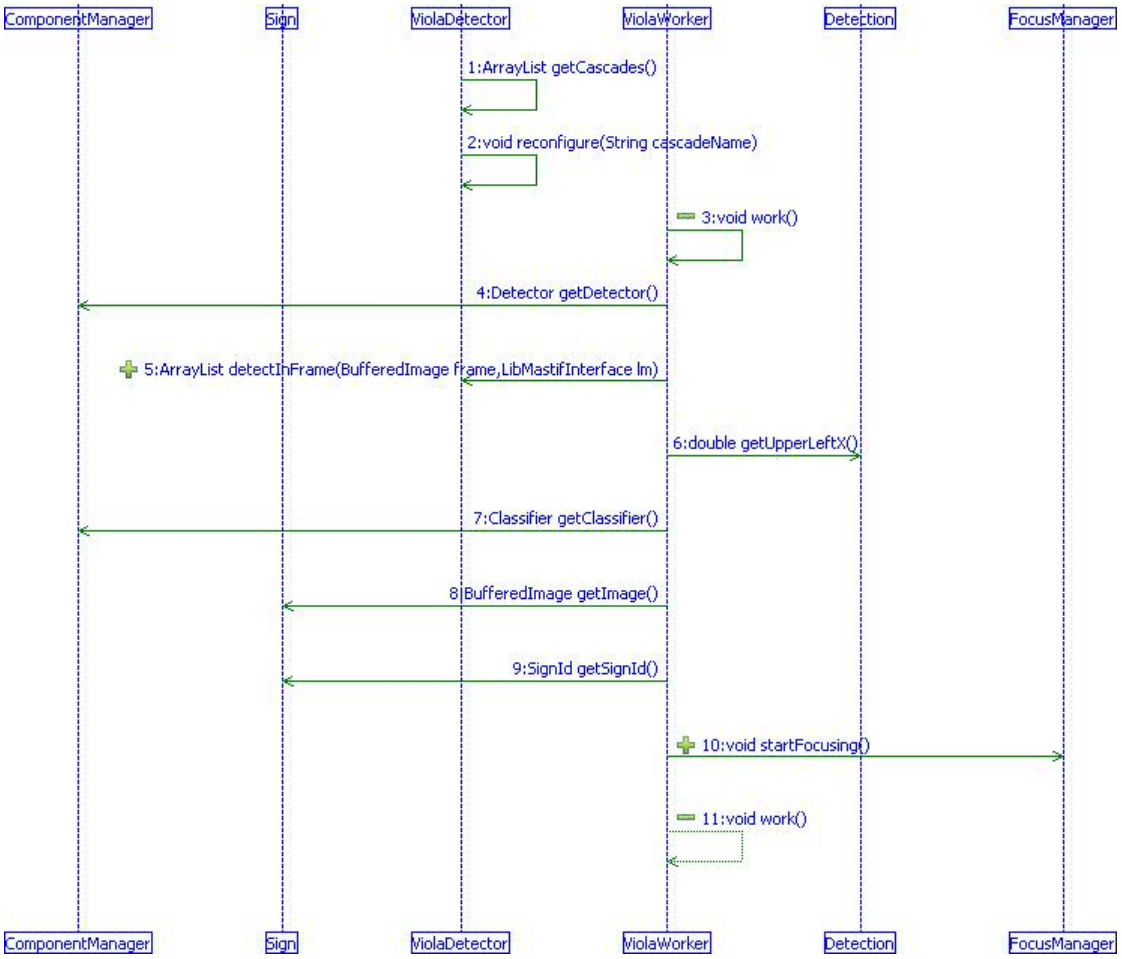
Slika 4.8: Razredi povezani s dodjelom fokusa izbornicima

4.2.4. Ugradnja detektora

Detektor je bio mnogo jednostavniji za ugradnju u sustav od klasifikatora. Budući da vanjske komponente usko surađuju na prezentacijskoj razini, velik dio ugradnje već je objašnjen. U okviru hijerarhije vanjskih komponenti koje pokreću algoritme, vidjeli smo da kod detekcije najvažniju ulogu ima razred `ViolaDetector`. Na prezentacijskoj razini, razred kojim `ViolaDetector` najviše surađuje jest `ViolaWorker`. Sekvencijski dijagram na slici 4.9 prikazuje njegov rad.

`ViolaWorker` ima nekoliko važnih zadaća:

pokretanje detektora - odnosno, pozivanje komponente `ViolaDetector`, te dohvat liste detekcija koju komponenta vrati



Slika 4.9: Sekvencijski dijagram toka detekcije na prezentacijskoj razini

iscrtavanje pravokutnika - za svaku detekciju, potrebno je izračunati koordinate regije gdje je detektiran znak na slici te okružiti tu regiju pravokutnikom

pozivanje klasifikatora - ako je uključena opcija za automatski poziv klasifikatora kad se stvori nova regija, `ViolaWorker` to čini

obavještavanje pogleda - ako se dogodila promjena na prezentacijskoj razini (dodan je novi pravokutnik), model se treba promijeniti te prijaviti promjenu svim svojim pogledima kako bi se oni iscrtali s novim podacima

početak fokusiranja - nakon svega, potrebno je započeti fokusiranje novih grafičkih komponenata kako bi se moglo s njima raditi pomoću tipkovnice

Odsječak koda na slici prikazuje razred `ViolaWorker`

U odsječku koda vidi se korištenje razreda `ComponentManager` za pokretanje algoritma detekcije. Razreda `Pravokutnik` opisuje pravokutnik na slici. Za to je

```
public void work(){

    detectedObjects = model.getComponentManager()
        .getDetector().detectInFrame(frame,
                model.getComponentManager().getLibMastif());
    for (Detection d  : detectedObjects) {
        System.out.println(d);
        Pravokutnik p = new Pravokutnik(
                (int)d.getUpperLeftX(),
                    frame.getHeight()-1-(int)d.getLowerRightY(),
                (int)Math.abs(d.getLowerRightX()-d.getUpperLeftX()),
                (int)Math.abs(d.getLowerRightY()-d.getUpperLeftY()));

        model.getOznakeZaPisanje().dodajPravokutnik(p);

        if (Options.getOption("SVM.modeAutomatic")){
            SVMWorker svmworker = new SVMWorker(model, p);
            svmworker.work();
        }
    }
    FocusManager.startFocusing();
    model.obavijestiPoglede();
}
```

Slika 4.10: Izvorni kod razreda `ViolaWorker`

potrebna koordinata njegovog gornjeg lijevog ugla, te širina i visina. Kada iz detekcije izračunamo koordinate položaja te instanciramo `Pravokutnik`, on se dodaje u model. Model u svakom trenu zna koji se pravokutnici nalaze na slikama, tj. pamti njihove oznake.

Razred `Opcije` spomenut je u kontekstu spremanja znakovnih konstanti, te u izvornom kodu razreda `ViolaWorker` na slici 4.10 vidimo kako on radi u praksi. *Mode automatic* znači da se klasifikator pokreće na svaku pojavu novog pravokutnika te ga pokušava klasificirati. Za pokretanje klasifikatora brine se razred `SVMWorker`, kao što je spomenuto ranije.

Kada su sve detekcije iscrtane na ekranu u obliku pravokutnika, potrebno je još samo dati fokus jednoj od njih (bira se zadnje učitana detekcija koju je algoritam vratio), te obavjestiti model da su se pogledi promjenili, što je vidljivo u zadnje dvije linije koda.

5. Zaključak

Serija nadogradnji aplikacije Marker vanjskim komponentama za klasifikaciju i detekciju prometnih znakova višestruko je skratila vrijeme koje korisnik mora utrošiti kako bi označio prometne znakove u slijedu slika. Podsustav koji sadrži i pokreće vanjske komponente izveden je na način kako bi bio maksimalno nadogradiv i modularan te olakšava ugradnju budućih komponenti u aplikaciju.

Postavljeni su temelji za ugradnju komponente za praćenje znakova u slijedovima slika, dodavanjem potrebnih sučelja u hijerarhiju vanjskih komponenti. Nativna biblioteka LibMastif u nekoliko je zahvata prilagođena za rad s komponentom za praćenje, te je ubrzan njen rad.

Velik dio koda izvorne aplikacije prekrojen je kako bi bio što fleksibilniji i moderan, pogotovo u prezentacijskom sloju koji obuhvaća grafičke komponente. Učinjeni su zahvati koji omogućavaju ugodan rad s tipkovnicom, gdje za praktički svaku akciju postoji odgovarajuća kratica.

Marker još uvijek nije dosegao željenu razinu zrelosti, iako se razvija već više od godinu dana. Vizija projekta jest razviti aplikaciju koja je sposobna iz danog izvora slijeda slika, konfigurirana odgovarajućim vanjskim komponentama (klasifikatori, detektori, komponente za praćenje), pronaći i označiti sve prometne znakove na način da korisnik potroši što manje vremena. Idealno, to bi značilo da korisnik može otvoriti video, aplikaciji zadati zadatak označavanja, pričekati tokom obrade, te na kraju smo potvrditi ili popraviti oznake.

Prvi korak ka automatskom načinu rada je potpuna ugradnja algoritma za praćenje kako bi se olakšalo gusto označavanje. Nadalje, tokom označavanja znakova, u svakom trenutku mora se odrediti hoće li se pokrenuti neki algoritam, ima li prometnih znakova na trenutnoj slici, je li potrebno pratiti znak koji je detektiran na prošloj slici. Stoga,

važno je ostvariti podsustav koji bi vodio brigu o interakciji vanjskih komponenti. Također, postoji mnogo varijacija opcija koje bi aplikacija trebala pružiti korisniku. Te bi opcije bile usko vezane uz rad podsustava za interakciju vanjskih komponenata, te odredile granicu kompromisa između kvalitete obrade slijeda slika i brzine rada. Na koncu, vizualni segment aplikacije može uvijek napredovati, kao i ugodnost pri korištenju.

LITERATURA

Chris Adamson i Joshua Marinacci. *Swing Hacks*. O'Reilly, 2005.

D. Bučar, A. Bulović, S. Gržičić, J. Hucaljuk, B. Kovačić, P. Palašek, B. Popović, i A. Sambol. *Optimiranje komponenata programa Marker*. FER, ZEMRIS, 2010a. URL `http://www.zemris.fer.hr/~ssegvic/project/0910grupa22/docs.pdf`.

D. Bučar, A. Bulović, P. Palašek, B. Popović, A. Trbojević, L. Zadrija, I. Kusalić, S. Šegvić, i K. Brkić. *Streamlining collection of training samples for object detection and classification in video*. FER, ZEMRIS, 2010b.

James W. Cooper. *Desing Patterns in Java*. 1998.

Paul Deitel i Harvey Deitel. *Advanced Java 2: How to Program*. Deitel inc., 2004.

Robert Eckstein. *Keyboard Bindings in Swing*. Sun inc., 2007a. URL `http://java.sun.com/products/jfc/tsc/special_report/kestrel/keybindings.html`.

Robert Eckstein. *Java SE Application Design With MVC*. Sun inc., 2007b. URL `http://java.sun.com/developer/technicalArticles/javase/mvc/`.

J. Shotton, G. Brostow1, J. Fauqueur, i R. Cipolla. *Segmentation and Recognition using Structure from Motion Point Clouds*. University College London and ETH Zurich, 2008. URL `http://jamie.shotton.org/work/publications/eccv08.pdf`.

Razvoj programske podrške za označavanje objekata u slijedovima slika

Sažetak

Cilj ovog rada je nadogradnja aplikacije za označavanje prometnih znakova u slijedu slika vanjskim komponentama za klasifikaciju, detekciju i praćenje prometnih znakova. Prikazan je postupak modeliranja hijerarhije vanjskih komponenata, te njihova ugradnja u postojeći sustav. Osim osnovne funkcionalnosti komponenata, učinjeni su potrebni zahvati za integraciju tih komponenti u prezentacijski sloj aplikacije. Uložen je napor u rješavanju problema fokusa grafičkih izbornika u aplikaciji te mapiranju korisničkih akcija na tipkovnicu. Detaljno je opisana suradnja izvornog koda aplikacije, pisanog u programskom jeziku Java, sa nativnom bibliotekom LibMastif koja sadrži korištene algoritme računalnog vida.

Ključne riječi: računalni vid, detekcija objekata, klasifikacija objekata, prometni znakovi, ugradnja vanjskih komponenti u aplikaciju srednje veličine

Developing application support for object annotation in image sequences

Abstract

The goal of this work is upgrading an annotating software with components for classification, detection and tracking of traffic signs. The process of modeling a hierarchy of external components is shown in detail. In addition to the basic functionality of the components, necessary steps were taken to achieve integration on the presentation level of the main application. Significant effort has been made in solving problems involving focus of graphical menus and keyboard mappings of user actions. The work also provides a detailed description of the interworking of the main application code, written in the Java programming language, with the native LibMastif library which contains the employed computer vision algorithms.

Keywords: computer vision, object detection, object classification, traffic signs, integrating components into a medium size application