ZAVOD ZA ELEKTRONIKU, MIKROELEKTRONIKU, RAČUNALNE I INTELIGENTNE SUSTAVE
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
SVEUČILIŠTE U ZAGREBU

INTEGRACIJA DODATNIH MOGUĆNOSTI U PROGRAMSKI SUSTAV MARKER

Damir Bučar, Ana Bulović, Sandro Gržičić, Josip Hucaljuk, Bruno Kovačić, Petar Palašek, Bojan Popović, Alan Sambol

Zagreb, 2009.

Sadržaj

1. Uvod

Marker je programski sustav, pisan u Javi, čija je svrha označavanje prometnih znakova u videu ili na slici. Predviđena upotreba Markera je bila pronalaženje i prepoznavanje prometnih znakova u videu ili nizu slika ručnim označavanjem.

Svrha takve upotrebe je prikupljanje pozitivnih i negativnih primjera koji se koriste za izgradnju i evaluaciju postupaka detekcije i klasifikacije prometnih znakova koje je naknadno moguće u njega integrirati. Marker također može poslužiti kao okolina u kojoj se može ispitati uspješnost spomenutih postupaka.

Prethodna funkcionalnost Markera je obuhvaćala:

- učitavanje filmova ili nizova statičnih slika
- označavanje znaka, brisanje oznake te odabir tipa znaka
- spremanje podataka o oznakama u tekstualnu datoteku.

Cilj ovog projekta je bio proširiti funkcionalnosti Markera u srvhu automatizacije njegovog rada.

Metodu raspoznavanja potpornim vektorima, implementiranu u prošlogodišnjem završnom radu [1], je bilo najjednostavnije integrirati zbog činjenice da je urađena u programskom jeziku Java. Iz gore spomenutog koda je trebalo izdvojiti potrebnu funkcionalnost te je prilagoditi za potrebe projekta.

Za pronalazak i praćenje prometnih znakova u slijedu okvira potrebno je integrirati algoritme za pronalazak i praćanje objekta koji su napisani u programskom jeziku C++. Budući da je Marker napisan u Javi, potrebno je na neki način premostiti ovu razliku. To je ostvareno korištenjem JNI-ja (Java Native Interface), programskog okvira koji omogućava da Java kod (koji se izvršava pomoću Java virtualnog stroja) poziva nativne biblioteke. To su aplikacije koje su specifične za pojedini operacijski sustav ili arhitekturu računala, poput aplikacija u C-u, C++-u ili asembleru. Za povezivanje koda u Javi i C++-u napravljeno je sučelje biblioteke libmastif preko kojeg je moguće pozivati algoritme računalnog vida.

Nakon što je omogućen prijenos parametara između algoritama u C++-u i Markera, bilo je potrebno napraviti grafičko sučelje za svaku od novih funkcionalnosti – poluautomatsko pronalaženje i prepoznavanje znakova. Za to je potrebno poznavati oblikovni obrazac MVC po kojem je napisan Marker.

U prvom poglavlju će biti riječ ukratko o metodi raspoznavanja potpornim vektorima te izmjenama koje je bilo potrebno učiniti u kodu [1] da bi postao upotrebljiv u svrhe projekta. Bit će navedene

nove funkcionalnosti koje je trebalo ostvariti pomoću raspoloživog koda te će biti prikazani rezultati ispitivanja uspješnosti metoda u ovisnosti o skupu za učenje.

U drugom poglavlju će biti pojašnjena ideja JNI-ja te način na koji je on primijenjen u ovom projektu. Ideja ovog poglavlja je da se nakon čitanja stekne općenit dojam kako koristiti ovaj programski okvir.

U trećem poglavlju će se sumirati rezultati prvog i drugog poglavlja – bit će objašenjena integracija prethodnih dviju komponenti u Marker.

2. Integracija metode raspoznavanja potpornim vektorima u Marker

Označavanje prometnog znaka moguće je napraviti ručno ili korištenjem Viola – Jones detektora.

Da bi Marker mogao obavljati poluautomatsko ili automatsko prepoznavanje prometnih znakova, potrebno je u njega integrirati algoritam za prepoznavanje objekata zasnovan na potpornim vektorima – SVM (Support Vector Machines). Algoritam je bio implementiran kao zasebna aplikacija sa grafičkim sučeljem. Da bi integracija bila moguća, potrebno je odvojiti logiku od pogleda. To je obavljeno metodama refactoringa koje će detaljnije biti pojašnjene u nastavku teksta. Potom je funkcionalnost upotpunjena za potrebe Markera. Dotada je bilo omogućeno prepoznavanje znaka iz dvije klase. Takvo prepoznavanje ne bi bilo moguće iskoristiti u Markeru jer je algoritam sam po sebi upotrebljiv samo za binarne usporedbe (npr. na slici se nalazi / ne nalazi znak za sretanje ulijevo). Nakon što je algoritam postao upotrebljiv, trebalo je ispitati njegovu uspješnost s različitim skupovima za učenje.

2.1. Refactoring aplikacije SVM

Nekada kod s kojim je potrebno raditi nije u obliku u kojem se lako dodaju nove funkcionalnosti. Ponekad je lakše na početku uložiti određeno vrijeme mijenjajući kod tako da postane razumljiviji nego ga odmah krenuti proširivati. Što je lošije kod strukturiran, veća je šansa da će ga se pogrešno razumjeti i da će proširenja imati greške koje će potom biti teško otkriti, opet iz istih razloga. Postoje neke standardne metode, od kojih će neke (posebno korisne i česte) biti spomenute, pomoću kojih se kod jednostavno, učinkovito i bez grešaka može dovesti u oblik koji je čitljiv svakom programeru.

Da bi se algoritamski dio koda mogao urediti, bilo ga je potrebno odvojiti od pogleda. Dio logike je bio implementiran u akcijama gumba, a neke funkcije koje nipošto nisu imale veze s pogledom su se nalazile u klasi zaduženoj za njega. Primjerice, u klasi `ImageGUI` zaduženoj za grafički prikaz se nalazila metoda `getPointsFromFile` koja se bavi računanjem vektora značajki za slike na kojima se obavlja učenje. Pozivom metode `getPoint` se računa vektor značajki jedne slike.

Neintuitivni nazivi, neki od kojih nisu ni točni, odnemažu razumijevanju koda. Jednostavan postupak kao što je preimenovanje metoda dosta pomaže razumijevanju onoga što one rade, posebice u matematički zahtjevnoj aplikaciji poput ove. Ako se metoda često poziva, posebice u

različitim klasama, potrebno je ime promijeniti na svim mjestima korištenja. Jednostavan način da se to učini je alat za refactoring unutar razvojne okoline Eclipse, koji pruža mogućnost jedinstvenog preimenovanja metode koji potom novo ime dodijeli na svim mjestima korištenja. Promjena imena metode iz `getPoint` u `getFeacureVector` će svakome tko ima osnovno znanje o metodi raspoznavanja potpornim vektorima reći točno što koji posao da metoda obavlja. Metoda `getPointsFromFile` je preimenovana u `calcAllFeatureVecs` jer računa vektore značajki za sve slike iz direktorija koji se predaje kao parametar.

Funkcionalnost učitavanja slika, vađenja vektora značajki, učenje SVM-a te nakon provjere pripadnosti slike određenoj klasi bilo je potrebno negdje premjestiti pa je napravljena klasa `SVMTools`. Kako je prvi cilj bio premjestiti metodu za računanje vektora značajki, potrebno je provjeriti koje sve metode ili varijable klase ona koristi i razmisliti treba li i njih premjestiti. U slučaju metode `getFeatureVector`, jasno je da je potrebno i nju premjestiti. Takve operacija seljenja metodi se najlakše radi korištenjem Refactor – Move alata. Nakon seljenja je potrebno provjeriti je li se desila kakva promjena funkcionalnosti i rada čitave aplikacije. To se radi pisanjem testova koji obuhvaćaju upravo izmijenjeni dio koda.

U klasi `ImageGUI` gumb koji pokreće učenje SVM-a ima akciju koja poziva metodu `solve`. Ta metoda prvo prepoznaje koji je kernel u pitanju, pokreće učenje te na kraju na grafičkoj komponenti ispisuje uspješnost operacije. Da bi se metoda uopće mogla koristiti u drugoj klasi, potrebno je odijeliti sve nepovezane stvari koje obavlja. Nakon premještanja je uklonjen ispis na grafičkoj komponenti, a korišteni kernel je predan kao parametar metode. Prepoznavanje o kojem je kernelu riječ je prebačeno u novu metodu – `determineKernelType` koja kao parametar prima KernelEnum vrijednost koju je korisnik odabrao. U ovisnosti o tome koristi li se kernel ili ne, parametar *w* se računa na različite načine. Matematički izračun koji se nalazio u if – else bloku je premješten u novu metodu `calculateW`. Pomicanje dijela koda iz jedne metode u novu zasebnu metodu može se obaviti koristeći već gotov alat razvojne okoline Eclipe: Refactor – Extract Method. U izborniku koji se pojavi moguće je odabrati ime nove metode, odabrati vidljivost metode (public, protected, default, private), ime povratnog parametra (ako takav postoji), automatski generirati komentar te naznačiti baca li metoda kakve iznimke (Exception).

Pritiskom na jedan od gumba bilo je moguće otvoriti sliku te pokrenuti prepoznavanje koje bi odredilo kojeg je tipa slika. Prvo se otvarao izbornik za odabir slike za prepoznavanje, potom se iz nje vadio vektor značajki, pa se pretvarala u oblik prikladan za prikaz na grafičkom sučelju, umetala se u njega te se konačno ispisivala poruka o tipu slike. To je previše posla za jednu metodu. Prvo je

bilo potrebno izvući u jedinstvenu metodu samu logiku – vađenje vektora značajki i određivanje tipa. Nova metoda je nazvana `checkImageType`. Provjera tipa se obavlja pozivom metode *g*, čije ime ne odaje dovoljno informacija o tome što radi, pa je i njeno ime promijenjeno u `getImageType`.

2.2. Dodavanje novih funkcionalnosti

Nakon što je sva dostupna funkcionalnost odijeljena od pogleda i preoblikovana na nešto intuitivniji način, bilo je potrebno omogućiti nekoliko stvari. Da se proces učenja ne bi morao raditi svaki put kod pokretanja programa, trebalo je omogućiti da se naučeni stroj s potpornim vektorima (SVM) može spremiti i pročitati iz datoteke. To je nadalje omogućilo spremanje veće količini naučenih SVM-ova te njihove upotrebe u višeklasnom prepoznavanju koje izvorni algoritam ne podržava.

2.2.1. Spremanje i čitanje SVM-a iz datoteke

Budući da je algoritam potpornih vektora dosta kompliciran i tim koji je dalje radio na proširenju njegove funkcionalnosti ga nije u potpunosti razumio, trebalo je odlučiti koje parametre je potrebno spremiti da bi se jednom naučeni SVM mogao koristiti pri bilo kojem pokretanju aplikacije. Kako proces učenja obuhvaća obradu slike, može biti prilično spor pa je bilo potrebno omogućiti brži i efikasniji način njegove upotrebe u smislu da je za određeni skup slika učenje potrebno provesti samo jednom, dok se svaki sljedeći put SVM može učitati iz datoteke. Proučavajući kod koji se koristio za provjeru pripadnosti slike određenoj klasi, napravljen je popis svih iskorištenih parametara. Osim samih parametara SVM-a, za potrebe Markera je potrebno znati koje su klase znakova iskorištene pri pojedinom procesu učenja. Upravo zbog toga je dodan enum koji sadrži popis svih vrsta znakova zapisanih njihovim službenim kodovima. Da bi se mogao održati nekakav standard što se tiče zapisa tipova slika, svi ulazni direktoriji moraju imati određene karakteristike. Ulazni direktorij je direktorij u kom se nalaze 2 poddirektorija, c1 i c-1 u kojima se nalaze slike za učenje. Brojevi uz c govore koja klasa (1 ili -1) se pridružuje slikama iz tog direktorija. Primjer imena direktorija je A01_A11 gdje je A01 tip znaka u direktoriju c-1, a A11 tip znaka u direktoriju c1. U klasu `SVM` su dodane privatne varijable `typeFirst` i `typeSecond`, a u klasi `SVMTools` je dodana metoda koja parsiranjem zna odrediti te tipove. Uz klase znakova, potrebno je zapisati i kernel koji se koristio prilikom učenja. To je ostvareno na način da je u sučelje *Kernel* dodana metoda `toStream` koju onda svaki kernel zasebno implementira. Prvo što svaki kernel zapisuje je prvo slovo imena, pa će tako *NO_KERNEL* zapisati N, *GAUSSIAN_KERNEL* G i tako dalje. Osim imena kernela, svaki će posebno zapisati svoje parametre koji se koriste pri izračunu. Ako svaki kernel zna u `DataOutputStream` zapisati svoje podatke, onda ih mora znati i pročitati. U sučelje

`Kernel` je dodana i metoda `fromStream`. Njena implementacija je prilično jednostavna – svaki kernel čita svoje parametre iz `DataInputStream`-a. Dakle, zapis naučenog SVM-a koji nudi klasa `SVMTools` delegira tu odgovornost klasi `SVM`. Čitanje naučenog SVM-a iz datoteke se obavlja pozivom metode `loadSVMFromFile` kojoj se predaje datoteka u koju je prethodno zapisan naučeni SVM. Ekstenzija datoteke s klasifikatorom je .svm, te nema zadane konvencije što se tiče imenovanja budući da se u nju svakako zapisuju tipovi slika, ali se preporuča zbog jasnoće da se imenuju jednako kao i direktoriji sa slikama za učenje.

Nakon što je omogućeno spremanje i čitanje SVM-a, moguće ih je naučiti više i upotrijebiti u raspoznavanju prometnih znakova, i to ne samo binarno kako osnovni algoritam dopušta, već i višeklasno.

2.2.2. Implementacija višestrukog izbora korištenjem binarnog algoritma potpornih vektora

Koristeći samo binarno prepoznavanje koje pruža osnovni algoritam potpornih vektora, ne može se pokrenuti prepoznavanje prometnog znaka u Markeru. Postoje dva načina na koja se može implementirati višestruki izbor. U oba je osnovna ideja da se jedan višeklasni problem podijeli na više binarnih problema. Svaki binarni problem posjeduje binarni klasifikator s funkcijom koja za pozitivno prepoznatu sliku tipa 1 vraća pozitivne vrijednosti, a za klasu -1 negativne vrijednosti. Ako se ne koristi jezgra, apsolutna vrijednost povratne vrijednosti može biti mjerilo uspješnosti prepoznavanja. To ne vrijedi za slučajeve u kojima se koristi jezgra. Kako se u našem slučaju ne koristi kernel, povratna vrijednost nam ipak može reći nešto o uspješnosti prepoznavanja. Što je veća apsolutna vrijednost rezultata poziva funkcije, to je veća vjerojatnost uspjeha prepoznavanja.

"Jedan nasuprot jednog" (*one versus one*) strategija uspoređuje svaka dva tipa iz cijelog skupa tipova na raspolaganju i kao uspješno prepoznat znak se uzima onaj koji se najviše puta pojavio kao rezultat.

"Jedan nasuprot svih" (*one versus all*) strategija koristi SVM-ove naučene na nešto drukčiji način. Kao jedan tip pri učenju se uzimaju slike jednog znaka, a kao drugi tip se koriste slike svih ostalih znakova. Kao slike drugog tipa mogu se upotrijebiti i slike pozadine bez znakova.

Proučavanjem nije ustanovljena značajna razlika u uspješnosti između ove dvije metode. Za primjenu u Markeru je odabrana potonja metoda. Razlog odabira je jednostavnija implementacija te manje naučenih SVM-ova što znači i kraće vrijeme učitavanja pri pokretanju aplikacije. Još jedna prednost korištenja metode jedan nasuprot svih je činjenica da budući da se pri učenju nije koristio kernel moguće je koristiti direktan rezultat funkcije uspoređivanja, a ne samo vrijednosti 1 ili -1

koje se moraju koristiti pri uporabi kernela. To pomaže pri klasifikaciji rezultata u slučaju da više od jednog SVM-a vrati pozitivan rezultat. Kako je broj značajki vektora po slici jednak 24x24 (piksela)x3(RGB vrijednost) = 1728 veći od broja slika kojima se raspolaže, nekorištenje jezgre je opravdano. Kako će podaci biti mapirani u prostor dimenzija 1728, a maksimalan broj slika u skupu za učenje s kojim se raspolagalo je bio nešto veći od 300, moguće ih je dobro razdvojiti bez povećavanja broja dimenzija. Ako se u budućnosti broj slika popne iznad tog broja, trebalo bi koristiti jezgru jer inače nije moguće dobro mapirati podatke zbog manjeg broja dimenzija prostora u kog se podaci mapiraju od broja slika u skupu za učenje.

Da bi se jednostavnije moglo koristiti podatke o odabiru i prepoznavanju znakova, u projekt SVM je dodana nova klasa `Sign` koja sadržava sve što bi se trebalo znati o jednom znaku. To je njegova jedinstvena oznaka, opis te slika. Klasa sadrži metodu `loadImageFromFile` kojom se učitava slika znaka. Za potrebe prepoznavanja znaku se može postaviti i vjerojatnost `p` da je on ispravno prepoznat. Da bi se moglo jednostavno koristiti kolekciju znakova bez repetitivnog učitavanja napravljena je klasa `Signs` koja sadrži hash–mapu u kojoj se po jedinstvenoj oznaci mapiraju svi znakovi. Zbog toga što se u vrijeme implementacije ove funkcije raspolagalo samo sa slikama znakova oblika trokuta, popis je napravljen samo tih znakova. Njihovi kodovi su od A1 do A50 i njihov popis se nalazi u projektu Marker, direktoriju komponente u datoteci *signs.txt*. Uz kodove prometnih znakova nalaze se i kratki opisi sa stranice Hrvatskog autokluba. Iz te datoteke se prvo učitavaju kodovi i opisi čime se popunjava mapa te se naknadno učitavaju slike koje se u projektu nalaze u direktoriju slike.

Za korištenje višestrukog izbora nudi se klasa *MultipleChoiceSVM*. Objekt te klase se instancira na način da mu se preda direktorij u kojem se nalaze svi naučeni SVM-ovi koji se žele koristiti u višeklasnom prepoznavanju. SVM-ovi se učitaju i spreme u listu. Klasa nudi metodu za prepoznavanje slike *recognizeImage*. Ona sliku koja joj se predaje kao parametar skalira na veličinu 24x24, pronađe njen vektor značajki te za svaki od raspoloživih SVM-ova provjerava vraća li određeni tip znaka ili ne (to jest vraća li tip ALL). Ako za neki SVM vraća naznačeni tip znaka se taj znak onda sprema u listu prepoznatih znakova. Nakon što je taj proces završen, prepoznati znakovi se sortiraju po vjerojatnosti ispravnog prepoznavanja.

2.3. Ovisnost uspjeha prepoznavanja o skupu za učenje

Da bi se dobila nekakva ideja o tome koliko uspješnost prepoznavanja ovisi o skupu za učenje. Napisana je klasa `ImageChoiceTest` koja svaki put koristi drukčiji skup za učenje na način da

slijedno bira 15% slika te ih ukloni iz skupa za učenje. Te slike se naknadno koriste za testiranje. U tablici 2.1 su prikazani rezultati svakog pojedinačnog naučenog SVM-a kao broj slika za testiranje tipa 1 i -1 te ukupan broj ispravno prepoznatih slika. Korišteni direktorij je input/ALL_A11.

Testne slike tipa ALL	Uspješno prepoznate slike tipa All	Testne slike tipa A11	Uspješno prepoznate slike tipa A11	Postoci uspješnosti
17	16	33	22	0.94 : 0.67
17	16	33	23	0.94 : 0.70
17	15	33	27	0.88 : 0.82
17	16	33	20	0.94 : 0.61
17	16	33	24	0.94 : 0.73
17	17	55	36	1 : 0.65

Očigledno skup za učenje ima velik utjecaj na uspješnost prepoznavanja. Ručnim pregledavanjem skupa za učenje je postignuta 90%-tna uspješnost kod prepoznavanja.

Klasa nije na raspolaganju u Markeru jer je nastala samo iz eksperimentalnih razloga, ali se i dalje može naći na repozitoriju na adresi https://garo.zemris.fer.hr/svn/SVM.

2.4. Korištenje gotovih rješenja

Kao alternativa ovom rješenju može se koristiti LIBSVM, raspoloživ u programskim jezicima Java, Python i C. Podržava među ostalim i višeklasnu klasifikaciju. Za testiranje korištenih parametara kod klasificiranja može se koristiti web aplikacija na njihovim stranicama (http://www.csie.ntu.edu.tw/~cjlin/libsvm/). Parametri koje je moguće mijenjati su među ostalim tip SVM-a, tip jezgre, stupanj te gamma vrijednost u funkciji jezgre, epsilon (tolerancija na odstupanja).

Za potrebe testiranja je može se koristiti i originalna aplikacija upotrebljena u ovom radu koja se nalazi na repozitoriju na adresi https://garo.zemris.fer.hr/svn/SVM.

2.5. Daljnji rad na aplikaciji

Kako je postupak pronalaska znaka različit za razne oblike znakova, tako je moguće na neki način signalizirati o kojem se obliku znaka radi. Prva ideja je bila da se između pronalaska i prepoznavanja znaka postavi jedan dodatni sloj prepoznavanja koji bi služio samo za određivanje oblika znaka. SVM-ovi bi se učili redom: trokuti – krugovi, trokuti – kvadrati te krugovi – kvadrati. Kako ovako naučeni SVM-ovi nisu pokazivali velik postotak uspješno prepoznatih znakova, ta ideja je odbačena. Uzme li se u obzir da bi se prepoznalo 80% oblika znakova ispravno, te ako se uz taj dosta nizak postotak uzme u obzir da je prosječan postotak uspješno prepoznatih trokutastih znakova oko 90%, postotak uspješno prepoznatih znakova postupkom prepoznavanja oblika te potom samog znaka se smanjuje. Bolji način je da algoritam za pronalaženje znakova ostavi neku

vrstu potpisa o kojem se obliku radi. Time se ne smanjuje uspješnost prepoznavanja te se smanjuje vrijeme potrebno za učitavanje SVM-ova (dodatna 3) te vrijeme potrebno za prepoznavanje jer nema sloja između pronalaska i prepoznavanja znaka.

Zbog nedostatka slika pojedinih znakova došlo se na ideju da se pojedini znakovi koje se ne susreće često grupiraju s nekim vizualno sličnim znakovima velikog broja pojavljivanja. Od toga se odustalo zbog tog što metoda potpornih vektora ne radi na način koji podržao grupiranje objekata i time bi se njegova učinkovitost znatno smanjila. Zasad nije moguće prepoznati znakove koji se rijetko susreću (iako je moguć slučajni pozitivan rezultat).

Kako će broj znakova s kojim se raspolaže rasti, eventualno će biti potrebno prilagoditi korištenje višeklasnog prepoznavanja za upotrebu kernela. To znači da će biti bolje koristiti jedan nasuprot jednog strategiju. Moguće je dodati metodu u klasu `MultipleChoiceSVM` koja će to raditi.

3. Java Native Interface

U sklopu ovog projekta javila se potreba za korištenjem algoritama koji već postoje napisani u C++-u, pa je bilo praktično iskoristiti već gotove implementacije. Uz to, neosporiva je nadmoć C++-a naspram Jave u pogledu brzine, koja je vrlo bitna u dijelovima ovog projekta. Java pruža podršku povezivanju s nativnim (C, C++ i sl.) kodom. Međutim, izvedba toga i nije sasvim jednostavna.

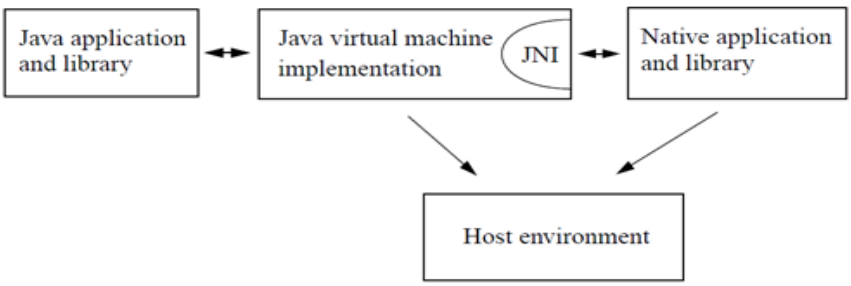
Za povezivanje s Javom, potrebno je imati instaliranu Javu, Eclipse (ili neki drugi IDE) i neki C++ prevodioc (Microsoft Visual Studio 2008 u našem slučaju).

Za ostvarenje tražene funkcionalnosti kreiran je jedan C++ projekt koji može komunicirati s Javom i libmastif-om te na taj način posreduje u komunikaciji tih dvaju dijelova.

Biblioteka libMastif omogućava pozivanje algoritama ljuske cvsh (komponenata koje enkapsuliraju postupke računalnog vida) preko poziva sučeljnih funkcija dinamičke biblioteke. Time se postiže interoperabilnost komponenti koje su razvijene i istestirane u okviru ljuske cvsh s klijentima koji mogu biti pisani u različitim programskim jezicima. Konkretno, libMastif se uspješno kor isti iz glavnih programa koji su pisani u Visual basicu i Javi.

Biblioteka libMastif_jni je odgovorna za omogućavanje pozivanja libMastifa iz Jave. Java ne može pozivati proizvoljne dinamičke biblioteke, nego samo one biblioteke koje poštuju specifikaciju JNI. Da bi se libmastif mogao koristiti, veza između komponenata ljuske cvsh i Jave je ostvarena tako da je kreirana novu biblioteku libMastif_jni koja ostvaruje funkcionalnost libMastifa preko JNI-ja.

Prema svojoj funkciji, libMastif i libMastif_jni mogli bi se svrstati negdje između prilagodnika i middlewarea: prilagodnik obično ostaje u okviru jednog programskog jezika i ne pruža mogućnost odabira konkretnog objekta prema simboličkom imenu (naming service), dok middleware obično prenosi podatke preko računalne mreže.



Slika 3.1. Uloga JNI-ja

3.1. Prednosti i mane JNI-ja

Bitno je zapamtiti da korištenjem JNI-a gubimo dvije prednosti Jave.

Prvo, Java aplikacije koje koriste JNI, ne mogu se više pokretati na raznim operacijskim sustavima. Iako je dio koda napisan u Javi prenosiv na sve operacijske sustave, nativni dio koda će trebati posebno prevesti za svaku platformu na kojoj bi koristili aplikaciju.

Drugo, dok je Java *type-safe* i siguran jezik, nativni jezici poput C-a i C++-a nisu. Kao rezultat toga, treba povećati oprez dok razvijamo takve aplikacije. Metoda u nativnom dijelu koda koja ne radi kako bi trebala, može srušiti cijelu aplikaciju.

Kao neko pravilo, bilo bi dobro koristiti JNI u najmanjem mogućem broju klasa i čim više izolirati nativni kod od ostatka aplikacije.

3.2. Kada koristiti JNI

Prije nego odlučimo koristiti JNI u projektu, potrebno je razmotriti alternativne solucije, koje bi mogle biti primjerenije. Kao što je već spomenuto, aplikacije koje koriste JNI nasljeđuju i gubitke naspram aplikacija koje su napisane samo u Javi.

Postoje brojne alternative, neke od njih su:

- Java aplikacija može komunicirati s nativnom aplikacijom preko TCP/IP protokola ili pomoću drugih među-procesnih komunikacijskih (IPC) mehanizama
- Java aplikacija se može spojiti na neke zastarjele baze podataka pomoću JDBC™ API-ja
- Java aplikacija može iskoristiti prednosti distribuiranih objektnih tehnologija kao što je Java JDL

Svim tim alternativama je zajedničko da se nativni kod i Java aplikacija izvršavaju u različitim procesima (u nekim slučajevima čak i na različitim računalima). Odvojeni procesi nam pružaju veliku prednost. Zaštita adresnog prostora podržana od procesa daje nam visoki stupanj zaštite od pogreški u nativnom djelu koda. Rušenje aplikacije napisane u nativnom kodu ne izaziva nužno rušenje Java aplikacije koja komunicira s njom preko TCP/IP-a.

Međutim, nekada je nužno povezati Java aplikaciju s nativnim kodom koji se izvršava u istom procesu. To je slučaj kada JNI postaje koristan. Neki od takvih scenarija su:

- Java API često ne podržava sve mogućnosti hardwarea na kojem se izvršava. U takvim slučajevima ako aplikacija želi koristiti neke specijalne operacije, efikasno je to napraviti u native kodu i povezati ga da s Javom radi u istom procesu
- Postojanje nativnog koda koji nam se ne isplati ponovo programirati u Javi, a puno je efikasnije da se izvodi u istom procesu

Uglavnom, treba koristiti JNI ukoliko postoji potreba da se Java aplikacija i aplikacija pisana u nativnom kodu izvršavaju u istom procesu.

3.3. Priprema za povezivanje u Java projektu

Prvi korak u povezivanju jest kreiranje klase s definicijama metoda u Javi. Deklaracije preuzmemo iz nativnog koda na koji se povezujemo (libmastif u ovom projektu) i samo dodamo ključnu riječ ***native***.

Primjer deklaracije funkcije `libMastifCreate`:

```
public native int libMastifCreate (String alg);
```

Nakon što napišemo sve definicije funkcija, treba iz komandne linije pozvat alat za kreiranje zaglavlja.

To činimo naredbom:

```
Javah –jni imeKlase
```

gdje je *imeKlase* naziv klase u kojoj su zapisane definicije. Rezultat ove naredbe je C/C++ header datoteka s definicijama svih metoda koje treba implementirati u C/C++-u. Datoteka se smjesti u trenutni direktorij s nazivom *imeKlase.h*.

Ovdje je bitno napomenuti da je, ukoliko klasa koja sadrži nativne metode pripada nekom paketu, kreiranje headera nužno pokrenuti naredbom `Javah` uz potpuno kvalificirano ime klase, kako bi se funkcije mogle ispravno pozivati iz bilo koje klase projekta. U našem slučaju, u sklopu projekta *Marker*, klasa pripada paketu *hr\fer\zemris\cv\marker\komponente\libmastif*, stoga je u komandnu liniju nužno upisati sljedeće:

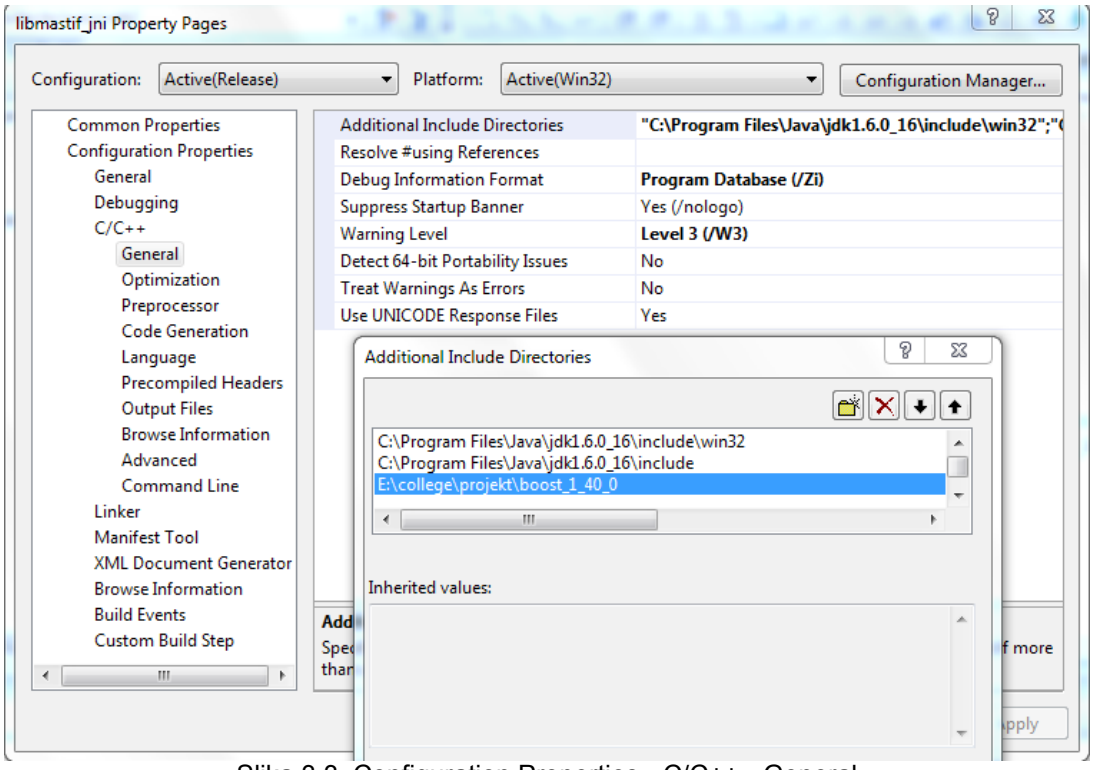
```
Javah –jni hr.fer.zemris.cv.marker.komponente.libmastif.LibMastifInterface
```

3.4. Priprema C++ projekta

Potrebno je kreirani novi C++ Class Library projekt u Visual Studiju i u njega uključiti header koji smo dobili u prethodnom koraku. U postavkama projekta je potrebno postaviti puteve do biblioteka Boosti sučelja biblioteke JNI na lokalnom računalu. Potrebni putevi do JNI header-a su:

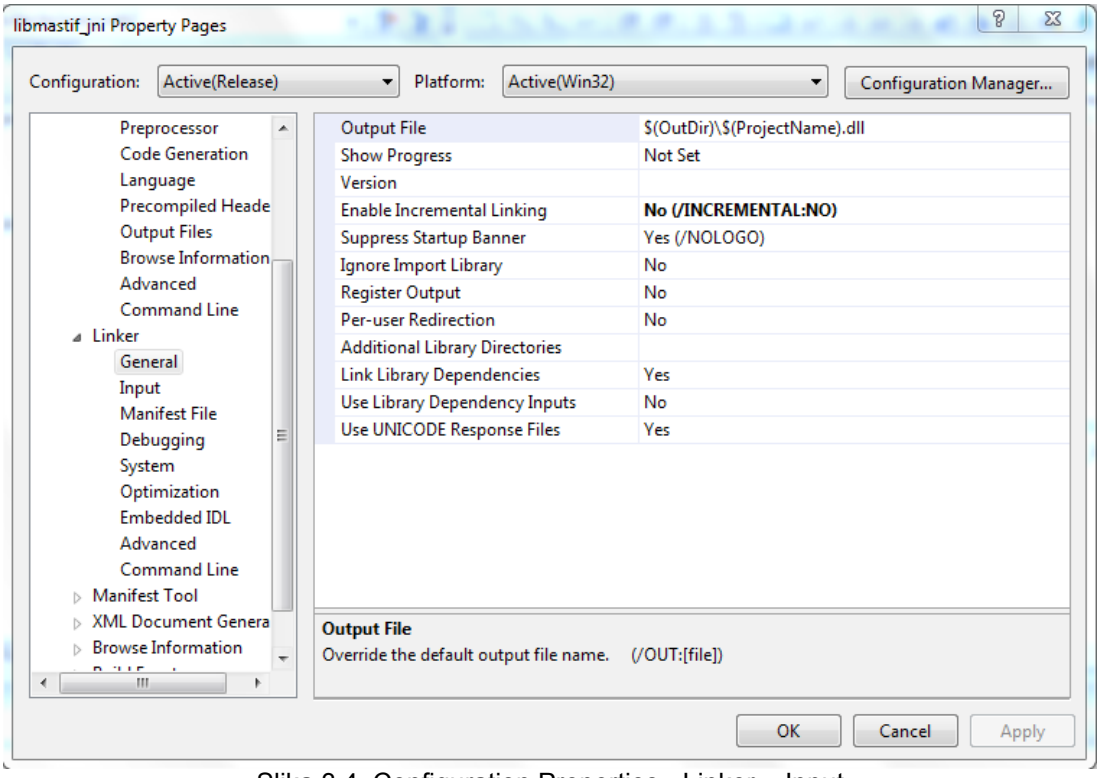
<JDK-Version-Path>\include; <JDK-Version-Path>\include\win32

Detaljnije postavke VS projekta prikazane su na slikama koje slijede.



Slika 3.3. Configuration Properties - C/C++ - General

Pod postavkama *Configuration properties – C/C++ - Precompiled Headers* potrebno je postaviti opciju *Not using precompiled headers*.



Slika 3.4. Configuration Properties - Linker – Input

Datoteka *libmastif_jni.def* treba sadržavati nazive funkcija, s punim kvalificiranim imenima istih.

libmastif_jni.def

```
EXPORTS
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifCreate @1
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifConfigDescription @2
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifCurrentConfig @3
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifConfig @4
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifProcess @5
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifDstCount @6
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifDstFmt @7
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifDstImg @8
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifGetDstPointsVectorSize @9
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifDstPoints @10
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifGetDstPointDescVectorSize @11
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifDstPointDescriptions @12
  Java_hr_fer_zemris_cv_marker_komponente_libmastif_LibMastifInterface_libMastifDestroy @13
```

Slika 3.5. Prikaz libmastif_jni.def datoteke

3.5. Implementacija metoda

Kod generiranja C++ sučelja, na definiciju funkcije koju smo kreirali u Javi, dodana su još dva parametra: ***JNIEnv*** *jEnv* i ***jobject*** *jThis*. Pomoću njih možemo komunicirati s Javom iz C++-a.

Značenja parametara:

JNIEnv* jEnv – pokazivač koji pokazuje na drugi pokazivač koji pokazuje na tablicu JNI funkcija. Možemo ga shvatiti kao strukturu koja sadrži sučelje prema JVM-u. Nativne metode pristupaju JVM-u isključivo preko JNI funkcija.

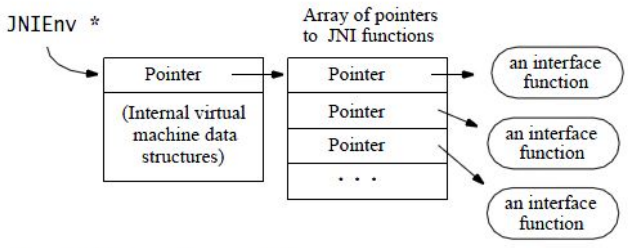


Figure 3.1 The JNIEnv Interface Pointer

Slika 3.6. JNIEnv *

jobject jThis – ovaj argument može imati dva oblika, ovisno da li se radi o statičnoj metodi ili metodi na nekoj instanci. Ako se radi o instanci, onda je ovo referenca na objekt nad kojem se poziva metoda. Ako se pak radi o statičnoj metodi, onda je ovo referenca na klasu u kojoj je ona definirana.

Ostali parametri odgovaraju parametrima originalne deklaracije iz komponente u Jave, npr `int` je postao `jint`, koji je definiran kao `long int`. Definicije svakog od tipova podataka mogu se vidjeti u *jni.h* i *jni_md.h*. Sve ih je vrlo jednostavno koristiti jer se može pretvarati iz jednog tipa u drugi (cast). Izuzetak je `jstring` kojeg nije moguće pretvoriti (castati) u običan string zbog korištenja različitog formatiranja.

3.6. Primjer jedne funkcije

```
Const char** javaGetStringArray(JNIEnv* jEnv, jobject jThis, char const* field, jint
vectorLen)

  {
   jfieldID jFid = javaGetFieldId(jEnv, jThis, field, "[Ljava/lang/String;");
   jobjectArray objArr = (jobjectArray)(jEnv)->GetObjectField(jThis, jFid); const char**
   ptrCharArr = (const char**) malloc (vectorLen * sizeof(char*));


   jstring temp;


   for(int i=0; i < vectorLen; i++){
              temp = (jstring) (jEnv)->GetObjectArrayElement(objArr,i); ptrCharArr[i] =
              (jEnv)->GetStringUTFChars(temp, NULL); std::cout<<"ptrCharArray C++
              "<<ptrCharArr[i]<<std::endl;

   }


   return ptrCharArr;
}
```

Ova funkcija dohvaća niz stringova proslijeđen iz Jave u C++. Parametar `field` označuje naziv traženog polja u Javi, a `vectorLen` njegovu duljinu.

Dohvaćanje započinjemo pomoću metode `javaGetFieldId`, koja je samo *wrapper* oko metode `GetFieldID` i vraća `ID` koji se pomoću nje dohvaća. Predajemo joj jEnv, jThis, naziv traženog polja i opis tipa podatka. Zadnji argument u toj funkciji (`"[Ljava/lang/String;"`) označuje da se radi o polju stringova. Dio parametra „[„ označuje da se radi o poljima, a „java/lang/String;“ da se radi o stringovima unutar polja. Za slučaj polja int-ova, koristimo „[I“.

Kada smo dobili ID polja, objekt jednostavno dohvatimo pomoću metode `GetObjectField`. Nakon toga podatke iz njega izvlačimo ugrađenom metodom `(jEnv)->GetObjectArrayElement(objArr,i)`. Međutim, ta metoda vraća `jstring` koji ne možemo jednostavno pretvoriti (*castati*) u C varijantu stringa `(char *)`, već ga konvertiramo pomoću funkcije `GetStringUTF`.

Bitno je napomenuti sljedeće - ukoliko elemente polja kojeg dohvaćamo „odozdo“, iz C/C++-a, trebamo proslijediti „gore“, u Javu, potrebno je nakon odrađenog posla pozvati funkciju za „otpuštanje“ elemenata polja, primjerice:

```
void javaSetDoubleArray(JNIEnv* jEnv, jobject jThis, char const* field, double* dArray,
jint vectorLen)
  {
   jfieldID jFid = javaGetFieldId(jEnv,jThis,field,"[D"); jdoubleArray doubleArr =
   (jdoubleArray)(jEnv)->GetObjectField(jThis, jFid); jdouble* arr = (jEnv)-
   >GetDoubleArrayElements(doubleArr, NULL);

   // pridruživanje vrijednosti polja iz C++-a polju iz Jave for(int i=0;i<vectorLen;i++)
   {
   arr[i]=dArray[i];
           }
   // "otpuštanje" polja
   (jEnv)->ReleaseDoubleArrayElements(doubleArr, arr, 0);
}
```

3.7. Povezivanje s projektom u Javi

Nakon implementacije svih potrebnih metoda u C++-u i pokretanja izgradnje projekta, kreirani DLL potrebno je smjestiti na jednu od lokacija:

- Direktorij *<JavaHome>/jre6/bin*
- Neki drugi direktorij sadržan u varijabli okoline *PATH*

Uz dobiveni DLL, potrebno je u isti direktorij kopirati i sve dll-ove koje on koristi. U našem slučaju to je samo *libmastif.dll*.

Prije pozivanja nativnih funkcija iz Jave, potrebno je izvršiti naredbu

```
System.load(„<putanja>/imeKreiranogDlla.dll“);
```

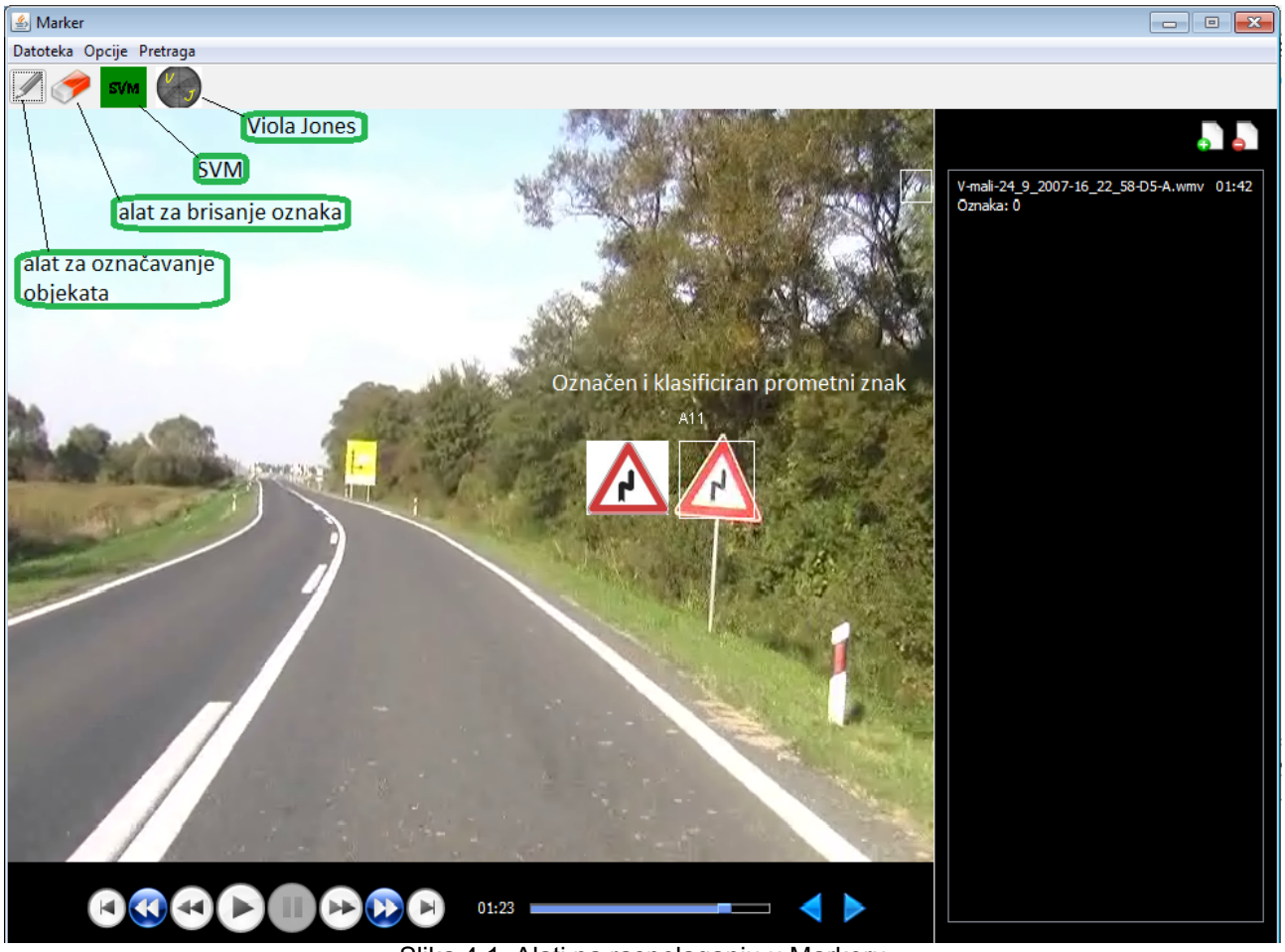
koja učitava kreirani DLL.

Nakon toga možemo bezbrižno pozivati nativne funkcije iz Jave.

4. Povezivanje komponenti s programskim sustavom Marker

Marker je programski sustav, pisan u Javi, čija je svrha označavanje prometnih znakova u videu ili na slici. Početni cilj Markera bio je mogućnost učitavanja i reproduciranja velikog broja formata izvora, te korištenje njegovih ugrađenih alata za označavanje znakova te odabir klase označenog znaka u odgovarajućem izborniku. Sav posao morao je ručno raditi korisnik aplikacije, pa se pokazalo da je označavanje malo duljih video snimki vremenski vrlo zahtjevan posao.

Korištenjem gotovih komponenti i algoritama za detekciju, prepoznavanje i praćenje objekata u videu, rad Markera bi se značajno ubrzao. Na ovom projektu razvijena je komponenta za prepoznavanje (SVM), a komponenta za detekciju razvijena je na projektu koji se odvijao paralelno koristeći poznate metode (Viola Jones algoritam). Jedan od ciljeva projekta bila je ugradnja gotovih komponenti na modularan način u korisničko sučelje programskog sustava Marker kako bi se postigla poluautomatska efikasnost rada. Korisnik sada više ne bi morao svaki objekt na slici označavati pravokutnikom kako bi potom prolazio kroz dva izbornika te naposljetku dodao klasu znaka uz označeni objekt. Poluautomatski način rada nudi korisniku ulančavanje i automatizaciju nekoliko gore navedenih radnji: na pritisak gumba za pauziranje videa uključuje se komponenta za detekciju te detektira objekte na slici, na nju se nadovezuje komponenta za prepoznavanje koja prepoznaje klasu znaka te je pridodaje označenom objektu. Korisnik sada može dalje gledati video dok ne uoči neki novi znak te zaustaviti video kako bi znak bio detektiran, prepoznat i označen. Korisnikova uloga sada je svedena na minimum.



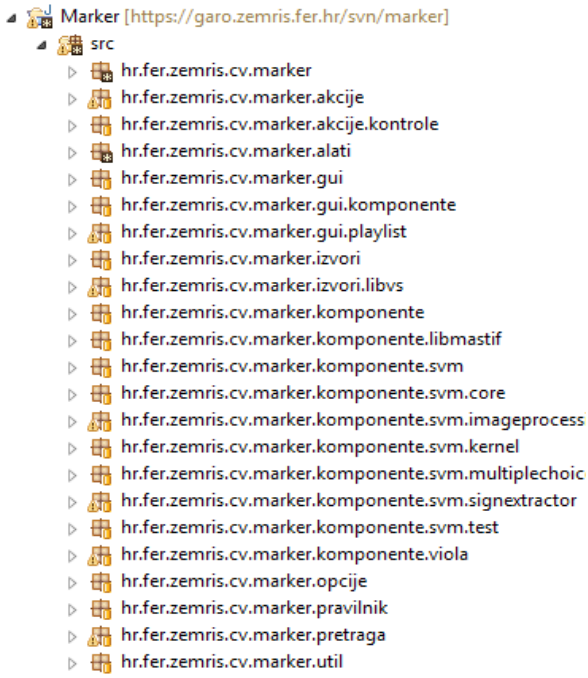
Slika 4.1. Alati na raspolaganju u Markeru

4.1. Organizacija programskog sustava Marker

Kao i svaki dobro organiziran projekt u programskom jeziku Java, Marker se sastoji od paketa koji sadrže klase bliske po namjeni. Za ugrađivanje komponenti u Marker bitan je potpaketi **hr.fer.zemris.cv.marker.komponente.** Paket **komponente** sadrži tri podpaketa:

- svm – sastoji se od dijelova SVM podprojekta koji služe za učenje i raspoznavanje prometnih znakova. Dio za učenje čini posebnu cjelinu nepovezanu s Markerom te se koristi kao komandnolinijski program koji generira podatke koje Marker učitava te pomoću kojih raspoznaje znakove.
- libmastif – skup JNI (Java Native Interface) poveznica prema odgovarajućim C++ knjižnicama koje odrađuju detekciju objekata. JNI predstavlja most između Jave u kojem je napisano korisničko sučelje te C++-a koji efikasnije odrađuje zahtjevne algoritamske operacije pri detekciji objekata
- viola – sadrži klase odgovorne za pokretanje Viola-Jones algoritama iz Markera.

Važnije klase iz ovih paketa obradit ćemo kasnije.



Slika 4.2. Paketi u programu Marker

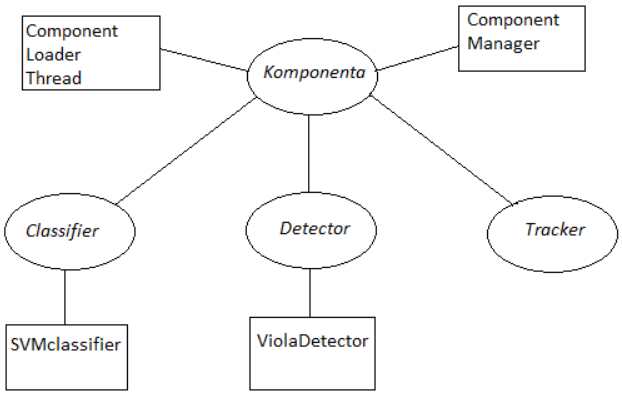
4.2. Izgradnja univerzalnih i modularnih sučelja za komponente

Kako bi projekt bio modularan potrebno je osigurati valjanu razinu razdvojenosti njegovih komponenata te omogućiti laku nadogradivost (ili zamjenjivost) istih. Iz perspektive programera tome je najlakše pristupiti primjenom *sučelja* (eng. Interface). Sučelja su klase koje sadrže samo zaglavlja funkcija koje pružaju. Konkretne klase koje implementiraju sučelja moraju implementirati sve metode koje sučelje nudi, te moraju poštivati tipove podataka koje funkcije u sučelju primaju kao argumente ili vraćaju. Ovaj pristup nudi tri bitne odlike. Razradit ćemo ih na primjeru sučelja Classifier koje implementiraju SVM moduli:

- različitim komponentama koje implementiraju isto sučelje moguće je baratati kao jednom komponentom (polimorfizam). Npr. ako na dva različita načina implementiramo poveznicu s SVM komponentom, njome kasnije baratamo na jednak način jer je ona tipa Classifier (implementira to sučelje).
- program razvijamo imajući unaprijed u vidu da ćemo svim objektima tipa Classifier baratati jednako, bez obzira kako oni unutar sebe mogu raditi na različit način. Ako u budućnosti

dođe do promjene implementacije SVM modula u samom Markeru nije potrebno mijenjati kod.

- ako SVM modul razvija tim koji ne radi na povezivanju Markera s njegovim komponentama (što je i slučaj), odgovarajuće sučelje će biti njihova dodirna točka. Ako netko iz grupe što razvija Marker napiše sučelje Classifier i tamo specificira što i kako očekuje za rad komponente, grupa koja radi SVM mu može pružiti upravo to što traži na efikasan način (jer eksplicitno znaju što i kako traži).



Slika 4.3. Klase i sučelja koje one implementiraju u Markeru

Slika 4.3. sadrži shematski opis povezanosti sučelja za komponente unutar Markera. Sučelja su označena kosim slovima, a konkretne klase normalnim. Hijerarhija se sastoji od sučelja Komponenta koje je na vrhu, te koje nasljeđuju podsučelja Classifier (komponente za prepoznavanje), Detector (komponente za detekciju), te Tracker (komponente za praćenje). U ovoj su fazi implementirane i funkcionalne komponente SVMclassifier, dakle Classifier koji radi pomoću SVM metode te ViolaDetector, Detector koji radi po Viola-Jonesu. Ostavljen je prostor za komponentu za praćenje budući da je ona planirana za razvijanje i integraciju. Iz grafikona se jasno vide prednosti ovakvog koncepta. ComponentManager je uslužna klasa koja sadrži instancirane objekte svih komponenti te ih lako koristi budući da ih sve gleda kao Komponente. Preko njega se dohvaćaju pojedinačne komponente potrebne za obavljanje odgovarajućeg zadatka. ComponentLoaderThread je klasa koja implementira radnu dretvu za učitavanje potrebnih podataka paralelno s pokretanjem ostatka programa. Trenutno je koristi samo SVM modul budući da on učitava datoteke za raspoznavanje prometnih znakova izgenerirane tokom učenja. Međutim, uzeto

je u obzir da bi u budućnosti i druge komponente mogle koristiti ovu dretvu pa ona barata isključivo s klasama koje implementiraju sučelje Komponenta. Zanimljivo je da je Komponenta zapravo samo nazivno sučelje koje služi samo kako bi se sve komponente objedinile pod istim nazivom i tako mogle koristiti:

```
package hr.fer.zemris.cv.marker.komponente;

public interface Komponenta {

}
```

4.0.1. Implementaciju komponente pokazat ćemo na primjeru klasa Classifier i SVMClassifier:

```
Public interface Classifier extends Komponenta{

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

}
```

Sučelje Classifier nudi dvije metode za klasifikaciju znakova – jednu za pojedinačno klasificiranje te drugu za višestruko (gdje predajemo listu slika i dobivamo mapu klasificiranih znakova). Marker ovom sučelju ne predaje cijelu sliku koja je trenutno u videu, nego samo izrezuje označeni dio (pomoću ImageCropper klase iz paketa **util**).

```
public class SVMClassifier implements Classifier{

   private MultipleChoiceSVM svm;

   public SVMClassifier(){
   this.svm = new MultipleChoiceSVM(new File("komponente" + File.separator +
     "istreniraniSVMovi"));
         }

   @Override
   public ArrayList<Sign> classifySingle(BufferedImage buffImage) { return
   svm.recognizeImage(buffImage);
         }

   @Override
   public HashMap<BufferedImage, String> classifyBulk (ArrayList<BufferedImage>
     images){
```

```
    throw new UnsupportedOperationException("Not supported yet :(");
        }
}
```

SVMClassifier implementira sučelje Classifier, pa prema tome i njegove metode. U konstruktoru se pokreće učitavanje naučenih SVM-ova. SVMClassifier implementira metodu sučelja Classifier za prepoznavanje pojedinačnih znakova. Druga metoda nije implementirana (jer nema potrebe), ali je ostavljena u slučaju potrebe u budućnosti.

4.3. Daljnji rad na Markeru

U planu je bilo dodati komponentu za praćenje. Taj bi dio programa omogućio da se u videu prati objekt kroz više uzastopnih slika koristeći KLT algoritam. Napisano je sučelje *Tracker* koje bi navedena komponenta mogla implementirati. Pritom treba implementirati metodu koja prima početnu sliku u kojoj je objekt prvi puta detektiran i dogovoreni broj uzastopnih slika gdje bi se objekt trebao pratiti te vratiti mapu odgovarajućih rednih brojeva slika gdje je lokacija objekta predviđena, i koordinate njegove lokacije u svakoj slici.

Zasad je potrebno ručno učiti novi SVM da bi se mogao koristiti u Markeru. U daljnjem radu bi se moglo omogućiti učenje direktno iz Markera.

Kada bi se komponente za praćenje i detekciju mogle optimizirati tako da mogu raditi u stvarnom vremenu (tj. da rade barem toliko brzo koliko ima slika po sekundi u videu), moglo bi se razmišljati o implementaciji u potpunosti automatskog načina rada Markera. Tada bi se video samo morao reproducirati u stvarnom vremenu, a komponente bi automatski napravile posao označavanja i spremanja rezultata. Uloga korisnika Markera svela bi se na učitavanje videa u program i evaluaciju rezultata nakon uspješnog izvođenja.

Osim toga, razmišljalo se i o stvaranju kompaktnijeg formata zapisa rezultata rada programa.

5. Zaključak

Programski sustav Marker ima potporu za označavanje, poluautomatsko pronalaženje i prepoznavanje znakova. Cilj ovog projekta je bio povezati sve dostupne algoritme za raspoznavanje i detekciju i upogoniti ih u Markeru, time povećavajući njegovu funkcionalnost.

Kako je rad na projektu gotovo nemoguć bez doticaja s nečijim tuđim kodom, a time i idejama, potrebno je dostupni kod dovesti u stanje koje je općenito razumljivo. To se postiže reorganizacijom dijelova koda koji nisu dovoljno jasni, refactoringom. Postupak pomaže razumijevanju aplikacije osobi koja ga provodi, a ako je provedeno na ispravan način, i budućim korisnicima koda. Primijenjen je na aplikaciji implementacije algoritma potpornih vektora koju je trebalo integrirati u Marker. Ponovnim proučavanjem nakon prvog stadija refactoringa uočene su daljnje mogućnosti poboljšanja organizacije koda kao što su podjela posla koji se nalazi u klasi SVMTools metodom *Extract Class*. Algoritam je implementiran u Javi jer je samo proces učenja spor. Kako se učenje odvija prije upotrebe u Markeru, to ne predstavlja problem. Zasada je algoritam prepozavanja primijenjen jedino na trokutastim znakovima u nedostatku potrebnog broja slika ostalih znakova.

Algoritmi za pronalaženje i praćenje su zbog procesorski zahtjevne obrade prilikom izvođenja pisani u programskom jeziku C++. Da bi se mogli upotrijebiti u aplikaciji pisanoj u Javi, pozivaju se iz dinamičke biblioteke kreirane JNI-jem. Implementacija povezivanja je završena i uspješno testirana za algoritam prepoznavanja zvan Viola – Jones algoritam. Komponenta za praćenje, KLT (Kanade-Lucas-Tomasi), još uvijek nije uspješno integrirana u Marker, ali se detaljne upute koje pruža 2. poglavlje mogu iskoristiti za njenu implementaciju.

Marker trenutno omogućava 2 načina pronalaska znaka na slici. Prvi podrazumijeva da korisnik ručno označi znakove, a druga da pritiskom ja VJ gumb pokrene automatsko pronalaženje znakova. Nakon toga moguće je ručno označiti kategoriju znaka ili pokrenuti automatsko prepoznavanje koje vraća od 1 do 3 rezultata najvjerojatnijih znakove koje potom korisnik može odabrati pritiskom na sliku ispravnog znaka. U slučaju da niti jedan od ponuđenih znakova nije ispravan, moguće je ignorirati rezultate prepoznavanja i ručno označiti znak.

Kako Marker svaki put iznova učitava sve naučene SVM-ove, uspješnost prepoznavanja je jednostavno poboljšati dodavanjem novih ili izmjenom starih SVM-ova. Njihovu uspješnost se može testirati korištenjem gotovih funkcija neke od kojih nisu priložene u Marker (poput rotacije testnih slika), ali se mogu naći u repozitoriju https://garo.zemris.fer.hr/svn/SVM.

6. Literatura

1. I. Kusalić: *Raspoznavanje prometnih znakova metodom potpornih vektora*, završni rad, 2009.
2. M. Fowler, K. Beck, J. Brant, W. Opdyke: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.
3. S. Liang: *The Java Native Interface: Programmer's Guide and Specification*, Sun Microsystems, 2002.
4. C. Hsu, C. Chang, C. Lin: *A Practical Guide to Support Vector Classification*, National Taiwan University, 2009.