Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Boštjan Cigan

Razvoj sistema za oddaljeni nadzor strežnika na Androidu

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Rok Rupnik

Ljubljana, 2014

Izjava o avtorstvu magistrskega dela

Spodaj podpisani Boštjan Cigan, z vpisno številko **63060008**, sem avtor magistrskega dela z naslovom:

Razvoj sistema za oddaljeni nadzor strežnika na Androidu

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom doc. dr. Roka Rupnika,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, 26. septembra 2014 Podpis avtorja:

Zahvaljujem se mentorju, doc. dr. Rok Rupniku za strokovne nasvete pri izdelavi dela.

Za vzpodbudne besede pri zadnjih izdihljajih obveznosti na fakulteti in podporo pri izdelavi dela se zahvaljujem punci Tjaši.

Za neskončno zalogo čokolad in drugih dobrin se zahvaljujem teti Lilijani in babici Ani.

Hvala tudi moji družini za podporo skozi celoten študij.

Kazalo

Seznam uporabljenih kratic

JVM (*angl.*)	java virtual machine javanski virtualni stroj
JRE (*angl.*)	java runtime environment okolje za izvajanje jave
J2EE (*angl.*)	java 2 enterprise edition java za podjetniške aplikacije
WORA (*angl.*)	Write Once Run Anywhere napiši ekrat poženi kjerkoli
CPU (*angl.*)	central processing unit centralna procesna enota
FQCN (*angl.*)	full qualified class name polno kvalificirano ime razreda
HTML (*angl.*)	hypertext markup language markirni jezik za hipertekst
CSS (*angl.*)	cascading style sheets datoteke za stiliranje spletne strani

SWOT (*angl.*)	strengths, weaknesses, opportunities and threats prednosti, slabosti, priložnosti nevarnosti
SDK (*angl.*)	software development kit paket za razvoj programske opreme
HTTP (*angl.*)	hypertext transfer protocol protokol za izmenjavo hiperteksta
TOTP (*angl.*)	time based one time password časovno omejeno veljavno geslo
IDE (*angl.*)	integrated development environment integrirano razvojno okolje
API (*angl.*)	application programming interface vmesnik za dostop do delov aplikacije
VPS (*angl.*)	virtual private server privatni virtualni strežnik
JSON (*angl.*)	javascript object notation format zapisa podatkov v ključ-vrednost obliki

Povzetek

Cilj naloge je ustvariti razširljivo aplikacijo tipa strežnik-odjemalec, ki omogoča nadzor orodij na strežniku. Pregledali smo obstoječe rešitve in analizirali njihove prednosti ter slabosti. Za lažje razumevanje izdelave razširljivih aplikacij predstavimo dinamično nalaganje v Javi. Pred razvojem aplikacije smo naredili zajem zahtev in načrtali arhitekturo. Za izvajanje ukazov na strežniku smo razvili Java aplikacijo, ki sprejema zahtevke in na njih odgovarja z JSON datotekami. Za prikaz grafičnega vmesnika smo realizirali hibridno aplikacijo na Androidu s pomočjo knjižnice Cordova. Delovanje smo testirali z obremenitvenim testom in rezultate ustrezno predstavili. Naredili smo tudi SWOT analizo in predstavili ideje za nadalnji razvoj.

Ključne besede:

dinamično nalaganje, refleksija, java, strežnik, odjemalec, cordova, nginx, swot

Abstract

The purpose of this thesis is to create a plugginable server-client application that allows the control of various software packages on a server. Current server managment tools are analyzed with their strength and weaknesses presented. Dynamic loading in java is presented for better understanding of creating plugginable applications. Before the application development, planning of the system architecture was performed and the functional and non-functional specifications were presented thorough. For managing the server we created a Java server that accepts commands and responds with JSON files. We developed a client on Android using a hybrid based approach with the library Cordova. A SWOT analyis of the application is presented and a server stress test is performed. The conclusion presents findings and ideas for the continued development of the application.

Keywords:

dynamic loading, reflection, java, server, client, cordova, nginx, swot

Poglavje 1

Uvod

V današnjih časih si težko predstavljamo upravljanje strežnikov brez namenskih orodij. Prva namenska orodja so bila ustvarjena leta 1996 z namenom, da bi poenostavili vsakdanja opravila na strežniku in hkrati tudi prihranili na času.

Za gostovanje spletnih strani uporabljamo namenske strežnike, za katere smo včasih najemali pakete gostovanja ali se odločili za takrat dražjo rešitev, najem privatnega strežnika. Danes se je cena najema takšnih strežnikov bistveno zmanjšala zaradi uvajanja virtualnih strojev in nižjih cen prenosa podatkov. Eno izmed podjetij, ki ponuja te pakete je DigitalOcean, kjer se cena najema giblje od pet do šestoštirideset dolarjev na mesec. Zaradi takšnih cen se večina trga danes odloča za najem privatnega strežnika.

Ker se danes svetovni splet zelo hitro razvija in se za razvoj spletnih aplikacij uporabljajo različne tehnologije, so namenska orodja zelo nemodularna in začetnim uporabnikom neprijazna. Zaradi tega razloga smo se lotili razvoja nove rešitve, ki bi vzdrževanje strežnika omogočala tudi ljudem brez predznanja iz področja sistemske administracije, hkrati pa ponujala tudi razširljivost skozi sistem za razvoj razširitev.

V drugem poglavju dela se lotimo predstavitve obstoječih rešitev. V tretjem poglavju predstavimo teoretične osnove dinamičnega nalaganja v Javi (kar nam omogoča razvoj sistema razširitev[10]). Opisali smo sestavo razredov, delovanje nalagalca razredov in pojem refleksije.

V četrtem poglavju je predstavljena načrtovalna faza razvoja rešitve, pri kateri smo naredili zajem zahtev, predstavili arhitekturo sistema in ustrezne diagrame. V petem poglavju so opisani ključni deli končne rešitve in realizacija sistema razširitev.

V šestem poglavju je predstavljena SWOT[26] analiza naše rešitve in testiranje obremenitve strežnika. Predstavljene so tudi možnosti za nadaljnji razvoj.

V zadnjem poglavju so predstavljene sklepne ugotovitve našega dela.

Poglavje 2

Pregled obstoječih pristopov

2.1 Orodja za lažje delo s strežniki

Ko je svetovni splet postal dostopen vsem, smo s tem doživeli tudi množično povečevanje strežnikov. V devetdesetih letih so te strežnike vzdrževali sistemski administratorji preko lupine (shell). Delo je bilo izjemno zamudno, hkrati pa si je bilo potrebno zapomniti veliko ukazov (primer 2.1 - nameščanje FTP strežnika vstpd).

```
1 bostjan@ZeroNETPC3:~$ wget https://security.appspot.com/
     downloads/vsftpd-2.3.4.tar.gz
2 bostjan@ZeroNETPC3:~$ tar -xvzf vsftpd-2.3.4.tar.gz
3 bostjan@ZeroNETPC3:~$ ./configure && make && make install
```

Koda 2.1: Nameščanje FTP strežnika vsftpd.

V današnjih časih za takšna opravila uporabljamo orodja, ki delno avtomatizirajo proces upravljanja s strežnikom, najbolj znana izmed njih so:

- cPanel,
- DirectAdmin,
- ajenti,

- Plesk,
- Webmin.

2.1.1 cPanel

cPanel je nadzorna plošča za strežnike, ki jo je ustvarilo podjetje cPanel inc. Prva verzija je bila izdana leta 1996, prvotno je bila namenjena kot rešitev za vzdrževanje strežnikov pri podjetju Speed Hosting, kasneje pa se je razvila v samostojno aplikacijo namenjeno drugim podjetjem, ki ponujajo gostovanje.

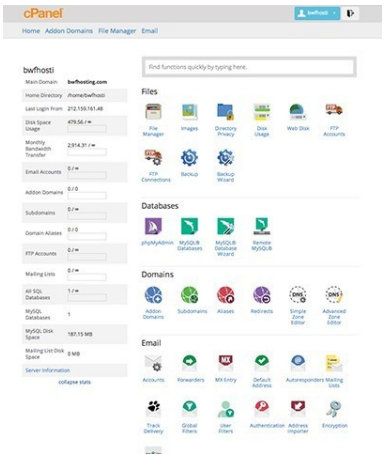
Sama arhitektura aplikacije je razdeljena na tri uporabniške dele:[4]

- administratorji - namenjen administriranju celotnega strežnika ali kopice strežnikov,
- prodajalci - namenjen ustvarjanju računov za prodajanje gostovanja na trenutnem strežniku,
- končni uporabniki - namenjen vsem, ki uporabljajo del strežniške kapacitete za gostovanje.

Med podprte aplikacije (ob namestitvi) spadajo Apache, PHP, MySQL, PostgreSQL, Perl in BIND (DNS) za e-pošto pa podpira POP3, IMAP In SMTP.

Za dostopanje do uporabniškega vmesnika (slika 2.1) aplikacija uporablja vrata številka 2083, ta tečejo na `HTTPS`.

Med njegove pomankljivosti lahko štejemo predvsem ceno (letna naročnina je tudi do dvesto dolarjev), težko dostopnost do razširitev (te so plačljive poleg redne cene naročnine) in težko odstranitev iz strežnika (ko je aplikacija



Slika 2.1: Uporabniški vmesnik cPanel.

enkrat nameščena, je potrebno formatiranje strežnika in ponovna namestitev vseh programov [5]).

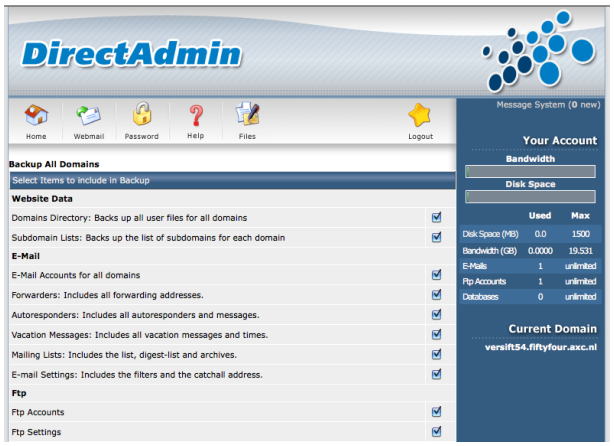
2.1.2 DirectAdmin

DirectAdmin je orodje, ki je bolj kot vzdrževanju strežnika namenjeno prodajanju gostovalnih paketov drugim strankam. Razvilo ga je podjetje JBMC Software, ponuja pa ga pod ceno devetindvajsetih dolarjev na mesec ali za 299 dolarjev za doživljenjsko licenco. Napisan je v programskem jeziku C in C++, uporabniški vmesnik pa je viden na sliki 2.2.

Med njegove slabosti štejemo visoko ceno ter nerazširljivost in omejen nabor podprtih aplikacij (trenutno vzdrževanje podatkovnih baz, FTP strežnikov, DNSjev in datotek na sistemu[6]).

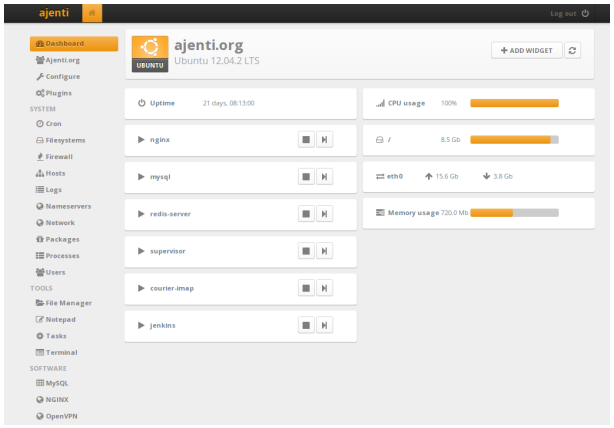
2.1.3 ajenti

Ajenti je orodje, ki ga je razvil Eugene Pankov. Podpira nadzorovanje večine odprtokodnih aplikacij na strežniku (nginx, PHP, cron naloge itd.). Teče na



Slika 2.2: Uporabniški vmesnik DirectAdmin.

vratih 8000 in komunicira preko SSL certifikata (če ga uporabnik ne poda, ga aplikacija ob namestitvi generira sama).



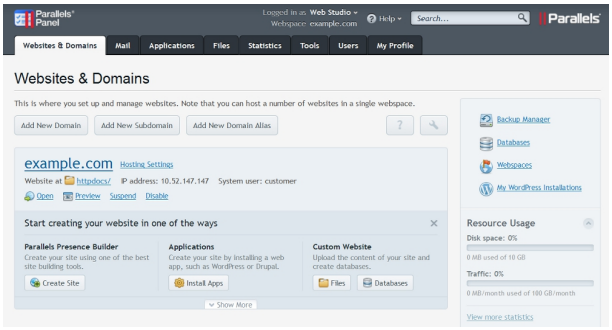
Slika 2.3: Uporabniški vmesnik ajenti.

Orodje je napisano v Pythonu in bazira na Webminu (kar omogoče lažjo izdelavo razširitev). Orodje je dostopno brezplačno za osebno rabo.

Njegove pomankljivosti so iste kakor pri webminu, ker ajenti delno bazira na njem (2.1.5).

2.1.4 Plesk

Plesk je orodje, ki ga je razvilo podjetje Parallels. Omogoča podobne dejavnosti kakor cPanel, te so ustvarjanje paketov za prodajo, ustvarjanje novih poštnih računov, novi DNS zpisi ipd. (uporabniški vmesnik je viden na sliki 2.4).



Slika 2.4: Uporabniški vmesnik Plesk.

Njegove slabosti so pomanjkanje vtičnikov in omejeno nameščanje aplikacij na strežnik (Plesk uporablja poseben format za nameščanje vsebin imenovan APS).

2.1.5 Webmin

Webmin je brezplačno orodje, ki ga je ustvaril Jamie Cameron. Podpira vzdrževanje Apache, PHP, MySQL, diskovne kvote in je razširljiv z moduli. Napisan je v jeziku Perl, teče pa na vratih 10000 (grafični uporabniški vmesnik je viden na sliki 2.5).

Med njegove pomankljivosti lahko štejemo težko pisanje modulov (predpogoj za pisanje modula je, da je modul sposoben razumeti vse datoteke za program, ki ga upravlja) in neprijaznost nekaterih funkcionalnosti začetnim uporabnikom.

Slika 2.5: Uporabniški vmesnik Webmin.

2.2 Razlogi za razvoj nove rešitve

Glavne prednosti nove rešitve, ki jo predlagamo so:

- **lažja namestitev aplikacij** - pri ostalih je potrebno namensko aplikacijo pred namestitvijo modula (če jih ta podpira) namestiti preko terminala,
- **lažje ustvarjanje razširitev** - ustvariti želimo skupnost ustvarjalcev razširitev, ki je podobna WordPress skupnosti, kar bi omogočalo lažje sledenje novim razširitvam in tudi večje število brezplačnih modulov,
- **podpora mobilnim platformam** - večina aplikacij ne ponuja namenske mobilne aplikacije,
- **enostavnejši uporabniški vmesnik** - zaradi pocenitev strežniških kapacitet smo dobili nov uporabniški segment (končni uporabniki ne najemajo več paketov gostovanja, temveč celotne strežnike, primer za velik razmah je podjetje DigitalOcean), nakup in vzdrževanje lastnega strežnika je postalo enostavnejše opravilo,
- **platformna in modulna neodvisnost** - želimo, da bi napisane razširitve z ustreznimi modifikacijami delovale tudi na drugih platformah.

Poglavje 3

Dinamično nalaganje v Javi

Za realizacijo naše aplikacije potrebujemo programski jezik v katerem lahko enostavno nalagamo novo programsko kodo. Java nam to omogoča s pomočjo dinamičnega nalaganja, hkrati pa omogoča tudi platformno neodvisnost.

3.1 Razvoj Jave

Programski jezik Java se je razvil iz jezika OAK[9], ki je bil razvit v zgodnjih devetdesetih letih kot platformno neodvisni jezik, namenjen povezovanju konzol in videorekorderjev, njegov primarni cilj pa je bil omogočanje storitve video na zahtevo. Ustvarjalci strojne opreme so označili jezik za "kompleksen", zato so se razvijalci jezika (James Gosling, Mike Sheridan, in Patrick Naughton) odločili, da se osredotočijo na razvoj za podporo svetovnemu spletu (na tej stopnji se je jezik tudi preimenoval v Java).

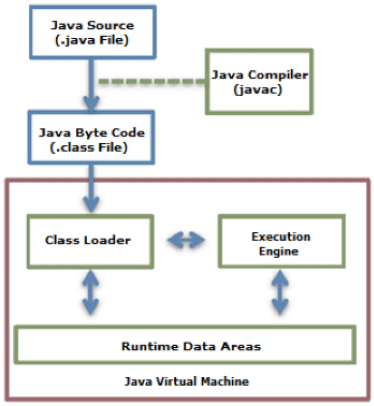
Prva javna verzija je bila izdana leta 1995, njen glavni cilj pa je bila paradigma "Write Once, Run Anywhere (WORA)" (v prevodu; napiši enkrat, poženi kjerkoli).[8] Java se je začela širiti, ko so veliki spletni brskalniki dodali podporo poganjanju javanskih appletov. Septembra 1999 se je Java razdelila na več verzij (ločenih glede na konfiguracijo), Java 2 Enterprise Edition (J2EE; namenjena podjetniškim aplikacijam), Java 2 Standard Edi-

tion (J2SE) in Java 2 Mobile Edition (J2ME; namenjena razvoju na mobilnih platformah).

Leta 2006 je podjetje Sun večino izvorne kode programskega jezika Java izdala pod GNU licenco, leta 2009 pa je podjetje Sun prevzel Oracle. Trenutne statistike pravijo, da je vsako leto okoli 930 milijonov prenosov Java Runtime Environmenta (JRE) in da trenutno tri bilijone mobilnih telefonov uporablja Javo.

3.2 Javanski virtualni stroj

Java Runtime Environment (v nadaljevanju JRE) sestavlja Java API in Javanski Virtualni Stroj (v nadaljevanju JVM). Vloga JVM je, da prebere Javansko aplikacijo skozi nalagalca razredov in jo izvede z Java APIjem. Potek prevajanja programa in izvajanja je prikazan na sliki 3.1.



Slika 3.1: Potek izvajanja programa v JVM.

Navidezni stroj je programska implementacija stroja (računalnika), ki izvaja programe kot stroj. Prvotno naj bi Java tekla na virtualnem stroju, ki je ločen od fizičnega stroja (WORA), vendar danes JVM teče na različnih strojnih

konfiguracijah in izvaja bitno java kodo brez da bi spreminjal izvajalno kodo. JVM ima naslednje lastnosti:[18]

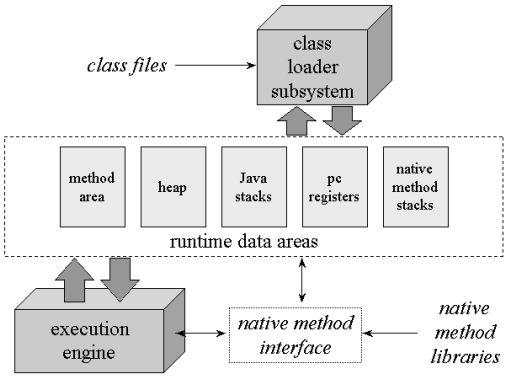
- **Skladovni stroj** - najbolj znane arhitekture (Intel X86, ARM) tečejo na podlagi registra, JVM teče na skladu,
- **Simbolično referenciranje** - simbolično se lahko referenciramo na vse tipe (razredi in vmesniki), razen primitivnih podatkovnih tipov
- **Čiščenje pomnilnika (garbage collection)** - Instanca razreda, ki jo je ustvaril uporabnik (programer) se avtomatično uniči,
- **Neodvisnost platforme s strogo definicijo podatkovnega tipa** - Za razliko od tradicionalnih jezikov (C, C++), kjer se velikost tipa definira glede na platformo na kateri teče program, JVM vnaprej definira velikost primitivnih podatkovnih tipov, da obdrži kompatibilnost in zagotovi neodvisnost platforme.

Možna je izdelava lastnega JVM, če se sledi specifikacijam[11], zaradi česar danes obstaja tudi več različnih javanskih virtualnih strojev kot sta Oracle Hotspot JVM in IBM JVM, delno pa na podanih specifikacijah bazira tudi Dalvik (Android).

3.2.1 Izvajalni pogon

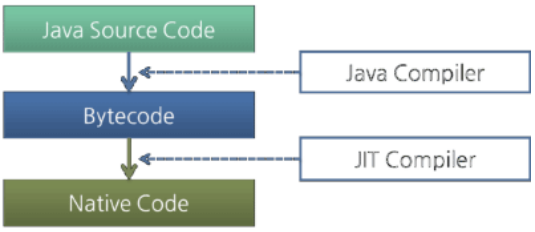
Del JVM je tudi izvajalni pogon (angl. execution engine). Bajtna koda (3.3), ki jo prejme JVM preko nalagalca razredov (3.5) je izvedena preko izvajalnega pogona. Izvajalni pogon bajtno kodo spremeni v kodo, ki jo razume JVM. To lahko naredi na dva načina:

- **interpreter** - prebere, interpretira in izvaja inštrukcije bajtne kode eno za drugo,



Slika 3.2: Struktura JVM.

- **JIT prevajalnik** - je bil uveden, da kompenzira slabosti interpreterja; izvajalni pogon najprej požene interpreter nato pa ob ustreznem času prevajalnik JIT celotno bajtno kodo prevede v nativno kodo, po tem procesu izvajalni pogon ne interpretira več metod vendar direktno izvaja nativno kodo (posledično je s tem povečana hitrost).



Slika 3.3: Shema prevajanja java datoteke.

V primerjavi z interpretiranjem prevajalnik JIT porabi več časa, vendar je JIT hitrejši kadar isti izsek kode izvajamo večkrat. JVM, ki uporablja JIT, interno pregleduje, kako pogosto se metoda izvaja in to metodo prevede s pomočjo JIT prevajalnika samo takrat, kadar je pogostost izvajanja visoka (ta se računa s pomočjo enostavnih hevristik[21]).

3.3 Java bytecode (bajtna koda)

Za implementacijo WORA JVM uporablja javansko bajtno kodo, ki je vmesni jezik med Javo (programerjem) in strojnim jezikom. Javanska bajtna koda je najmanjša enota, ki izvaja Javansko kodo in je hkrati tudi najpomembnejši element JVM.

JVM je emulator, ki emulira javasko bajtno kodo. Prevajalnik ne prevede programa direktno v strojno kodo, kakor nekateri drugi višjenivojski jeziki (C, C++) ampak ga prevede v jezik, ki ga razume JVM. Ker je bajtna koda platformno neodvisna, jo lahko izvajamo tudi drugje (tudi če se zamenja CPU ali operacijski sistem). Velikost prevedene kode je skorajda ista velikosti izvorne kode, tako da je zlahka prenosljiva in izvajana tudi preko omrežja.

```
1  public class ByteCode {
2
3          public static void main(String[] args) {
4                  int i = 10;
5                  int j = 20;
6
7                  int k = i + j;
8
9                  System.out.println("Sestevek:␣");
10                 System.out.println(k);
11
12         }
13
14 }
```

Koda 3.1: Preprost program v Javi, ki sešteje dve števili.

Prevedena javanska koda je v binarni obliki in je težko berljiva, zato so ustvarjalci JVM dodali orodje `javap`, ki omogoča razstavljanje že prevedene kode:[3]

```
1  javap -c ByteCode
```

Ukaz nam vrne izpis prevedene javanske kode v nam berljivi obliki:

```
1  public static void main(java.lang.String[]);
2    descriptor: ([Ljava/lang/String;)V
3    flags: ACC_PUBLIC, ACC_STATIC
4    Code:
5      stack=2, locals=4, args_size=1
6         0: bipush        10
7         2: istore_1
8         3: bipush        20
9         5: istore_2
10        6: iload_1
11        7: iload_2
12        8: iadd
13        9: istore_3
14       10: getstatic     #2
15       13: ldc           #3
16       15: invokevirtual #4
17       18: getstatic     #2
18       21: iload_3
19       22: invokevirtual #5
20       25: return
21     LineNumberTable:
22       line 4: 0
23       line 5: 3
24       line 7: 6
25       line 9: 10
26       line 10: 18
27       line 12: 25
28 }
```

Koda 3.2: Razstavljena koda iz primera 3.1 z orodjem `javap`.

Če pogledamo vrstico 6 v kodi 3.2 vidimo, da je sintaksa vsaj malce podobna nam znanim nizkonivojskim jezikom. Ukaz sestoji iz `OpCode` in `Operand`. Z ukazom `bipush` (`OpCode`) dodamo številki 10 (`Operand`)) in 20 na sklad, nakar jih z ukazom `iadd` seštejemo, z ukazom `istore` pa shranimo.

Če pogledamo še malce bolj zapleten primer, ki vsebuje tudi metode:

```
1 public class ByteCode {
```

```
2
3           public static void main(String[] args) {
4                   int k = sestej(1, 10);
5                   System.out.println("Sestevek: ");
6                   System.out.println(k);
7           }
8
9           public static int sestej(int i, int j) {
10                  int k = i + j;
11                  return k;
12          }
13
14  }
```

Koda 3.3: Seštevek dveh števil z metodo v Javi.

Razstavljena koda z orodjem `javap` potem izgleda takole:

```
1  public class ByteCode {
2    public ByteCode();
3      Code:
4         0: aload_0
5         1: invokespecial #1
6         4: return
7
8    public static void main(java.lang.String[]);
9      Code:
10        0: iconst_1
11        1: bipush        10
12        3: invokestatic  #2 // Method sestej:(II)I
13        6: istore_1
14        7: getstatic     #3
15       10: ldc           #4
16       12: invokevirtual #5
17       15: getstatic     #3
18       18: iload_1
19       19: invokevirtual #6
20       22: return
21
22   public static int sestej(int, int);
23     Code:
24        0: iload_0
25        1: iload_1
26        2: iadd
27        3: istore_2
```

```
28          4: iload_2
29          5: ireturn
30 }}
```

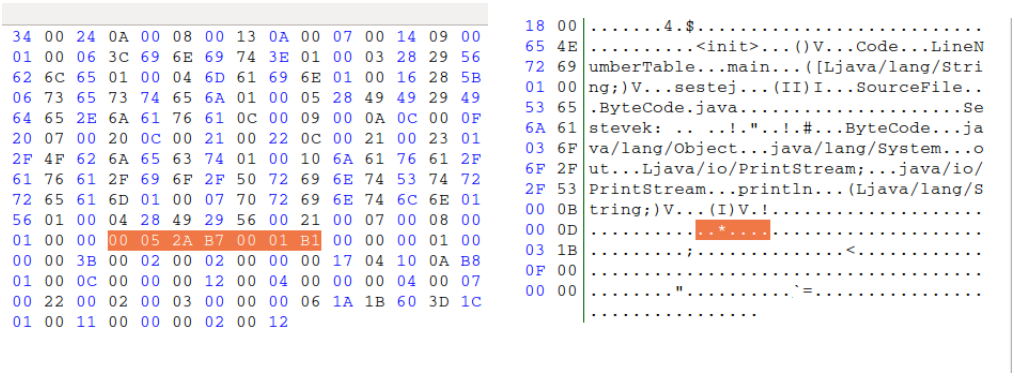
Koda 3.4: Razstavljena koda iz primera 3.3 z orodjem `javap`.

Ključna beseda `invokevirtual` je `OpCode`, ki je eden izmed osnovnih ukazov javanske bajtne kode. Poznamo štiri ključne besede, ki kličejo metodo v bajtni kodi:[24]

- **`invokeinterface`** - kliče vmesnik,
- **`invokespecial`** - kliče inicializator, privatno metodo ali nadrazred
- **`invokestatic`** - kliče statične metode,
- **`invokevirtual`** - kliče instance.

Če pogledamo vrstice od 24 do 28 v kodi 3.4 opazimo, da `OpCode` sestoji iz ključne besede in osembitnega števila (`aload_0 = 0x2a`). Najvišje število javanskih bajtnih inštrukcij `OpCode` je torej osem bitov (256). Ukaza `aload_0` in `aload_1` ne potrebujeta operandov, naslednji bajt od `aload_0` je torej `OpCode` za naslednji ukaz. Ukaza `getfield` in `invokevirtual` sta operanda, ki zasedeta dva bajta, torej je naslednji ukaz na prvem bajtu zapisan na četrtem bajtu (dva bajta preskočimo; vidno na sliki 3.4).

V bajtni kodi je instanca razreda označena z `L;`, void z `V;` in podobno. V našem primeru v vrstici 12 v kodi 3.4 je naša metoda `sestej`, ki vrača celo število, označena z `(II)I`, kar pomeni, da metoda vrača celo število in sprejme dve celi števili. V tabeli 3.3 je seznam pogosto uporabljenih izrazov za tipe.[11, str. 78]



(a) Hex urejevalnik (leva stran). (b) Hex urejevalnik (desna stran).

Slika 3.4: Pregled ukaza `invokestatic` in `istore` v urejevalniku HEX.

Bajtna koda Java	Tip	Opis
B	byte	
C	char	
D	double	
F	float	
I	int	
J	long	
L<ime razreda>	referenca	instanca razreda <ime razreda>
S	short	
Z	boolean	
[	referenca	enodimenzionalna tabela

Tabela 3.1: Tabela izražanja podatkovnih tipov v bajtnem jeziku Java.

Podatkovni tipi se torej prevedejo v javansko bajtno kodo, če vzamemo dva primera:[11, str. 79]

- `double d[][][]` se prevede v `[[[D`,
- `Object metoda(int I, double d, Thread t)` se prevede v `(IDLjava/lang/Thread;)Ljava/lang/Object;`.

3.4 Razredi

Kadar prevedemo javansko kodo, dobimo tako imenovane razrede (angl. class). Te datoteke vsebujejo enake podatke kakor njihovi ekvivalenti v java formatu, vendar je njihova koda prevedena v bajtno kodo (podobna zbirniku)[12], izvaja pa se na javanskem virtualnem stroju.

3.4.1 Pravila imenovanja in vidljivost

Vsak razred pripada paketu, ki definira vidljivost njegovih članov.

$$PackageName ::= Identifier \mid PackageName\ '.'\ Identifier$$

Različni razredi in vmesniki se lahko pojavijo z istimi imeni v različnih paketih. Da jih ločimo med seboj uporabljamo tako imenovani FQCN (polno ime razreda), kar pomeni, da je njihovo ime določeno z imenom razreda in paketom, kateremu pripadajo:

$$Class ::= PackageName\ '.'\ Identifier$$

Razredi in vmesniki definirajo polja in metode. Imenska prostora obeh nimata skupnih elementov. Polja razreda so spremenljivke razreda, spremenljivke metod in konstante. Ker Java dovoljuje ponovno definiranje metod (považanje metod oz. tako imenovani “overloading”) potrebujemo tip argumentov, da lahko ločimo med metodami, ki imajo ista imena. Za ta namen uporabljamo set metod (par ime metode - podpis argumentov), ki so v paru z razredom (ali vmesnikom), ki jih definira.[24]

$$
\begin{aligned}
FieldDesc &= Identifier \\
MethodDesc &= Identifier \times Type \\
MemberDesc &= FieldDesc \uplus MethodDesc \\
Field &= Class \times FieldDesc \\
Method &= Class \times MethodDesc \\
Member &= Field \uplus Method - Class \times MemberDesc
\end{aligned}
$$

Set osnovnih tipov kot so `void`, `int`, `boolean` so reprezentirani kot *BaseType*. Vsi razredi, vmesniki in polja so referenčni tipi, *RefType* pa je definiran kot najmanjši set, ki zagotavlja pogoju:

$$
\begin{gathered}
Class \subset RefType \\
\forall t \in RefType \cup BaseType \;.\; t[] \in RefType
\end{gathered}
$$

kjer je polje elementov tipa t zapisanih kot *t[]*. Set tipov je unija osnovnih tipov in referenčnih tipov:

$$Type = BaseType \cup RefType$$

Vsi razredi in vmesniki imajo tudi določeno vidljivost, ki se jo lahko spreminja z dostopnimi določili:

$$Modifier = \{\texttt{public}, \texttt{protected}, \texttt{default}, \texttt{private}\}$$

Dostop je določen s kombinacijo omejitev na paketu, razredu in instanci. Za razrede in vmesnike lahko uporabimo le `public` in `default`. Do javnih razredov lahko dostopamo iz kjerkoli, medtem ko z `default` določilom lahko do razreda dostopoma le iz trenutnega paketa. Za metode velja, da če so `private`, so dostopne le trenutnemu razredu, pri `protected` pa so na voljo podrazredom in istemu paketu.[24]

3.4.2 Sestava razreda

Formalno razred definiramo kot:[24]

$$\begin{aligned}
name: &\quad ClassFile \to Class \\
super: &\quad ClassFile \hookrightarrow Class \\
implement: &\quad ClassFile \to \mathcal{P}(Class) \\
member: &\quad ClassFile \to \mathcal{P}(Member) \\
member - type: &\quad ClassFile \times Member \hookrightarrow Type \\
member - modifier: &\quad ClassFile \times Member \hookrightarrow Modifier \\
class - modifier: &\quad ClassFile \to Modifier \\
references: &\quad ClassFile \to \mathcal{P}(Class)
\end{aligned}$$

kjer je

- $name$ - ime razreda,
- $super$ - ime nadrazreda,
- $implement$ - tabela (lahko tudi prazna) implementiranih vmesnikov,
- $member$ - seznam metod,
- $member - type$ - tip (ali tip, ki se vrača) metod,
- $member - modifier$ - dostopno določilo metode (spremenljivke),
- $class - modifier$ - dostopno določilo razreda,
- $references$ - seznam referenc na druge razrede.

Funkcije *member-type* in *member-modifier* so definirane le na metodah trenutnega razreda. Podobno velja tudi za *super*, ker razred `java.lang.Object` nima nadrazreda. Za podan razred *cf* torej velja:

$$\forall m \in member(cf);$$
$$origin(m) = name(cf)$$

Iz strojnega vidika razred vsebuje sklope osmih bitov. Vse reprezentacije podatkov v šestnajstih, dvaintridesetih in šestinštirideset bitih so reprezentirane z branjem dveh, štiri ali osem zaporednih osem bitnih sklopov.

V praksi (JVM specifikacijah) je razred definiran kot sledeča struktura (`ClassFile`): [11, str. 70]

```
1  ClassFile {
2      u4             magic;
3      u2             minor_version;
4      u2             major_version;
5      u2             constant_pool_count;
6      cp_info        constant_pool[constant_pool_count-1];
7      u2             access_flags;
8      u2             this_class;
9      u2             super_class;
10     u2             interfaces_count;
11     u2             interfaces[interfaces_count];
12     u2             fields_count;
13     field_info     fields[fields_count];
14     u2             methods_count;
15     method_info    methods[methods_count];
16     u2             attributes_count;
17     attribute_info attributes[attributes_count];
18 }
```

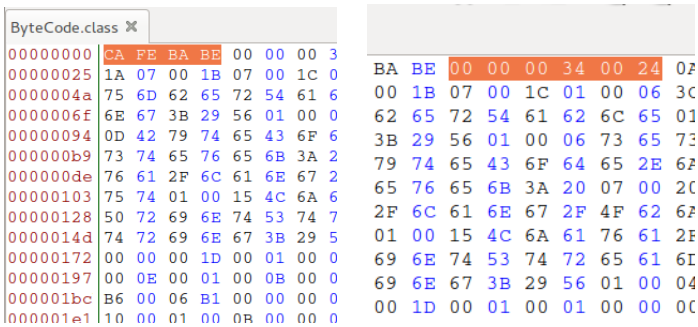
Koda 3.5: Osnovna sestava razreda.

Datoteka vsebuje sledeče podatke:[11, str. 70–74]

- **`magic`** - prvi štirje bajti razreda, je vnaprej določena vrednost, ki določa, če je datoteka razred (ni pa vrednost, ki bi nam zagotavljala, da je datoteka razred), v urejevalniku HEX vidimo da je začetna vrednost `0xCAFEBABE` (slika 3.5(a)),
- **`minor_version, major_version`** - naslednji štirje bajti vsebujejo informacijo o verziji Jave, na kateri je bila datoteka prevedena in katera

je minimalna verzija, ki je potrebna, da se koda uspešno izvede. Na sliki 3.5(b) vidimo, da je verzija JDK-ja `54.0`, kar je Java 1.8,

- **`constant_pool_count, constant_pool[]`** - takoj za verzijo sledi bazen konstant, v našem primeru je ta vrednost `0x0024`, kar je 35 indeksov (36-1),
- **`access_flags`** - zastavica, ki vsebuje dostopna določila razreda (`public`, `final`, `abstract` ipd.),
- **`this_class, super_class`** - vsebuje indeks, ki kaže na bazen konstant (kaže na razreda, ki je trenutni ali nadrazred),
- **`interfaces_count, interfaces[]`** - indeks v bazenu konstant, ki pove, koliko imamo implementiranih vmesnikov,
- **`fields_count, fields[]`** - prvo vsebuje število `field_info` struktur v `fields` tabeli, `field_info` struktura predstavlja spremenljivke razreda in instance spremenljivk, ki jih implementira ta razred ali vmesnik[11, str. 73],
- **`methods_count, methods[]`** - število metod v razredu in podatki o metodah v razredu (ime, tip, število parametrov, tip, ki ga metoda vrača, indeks v bazenu konstant in podatki o izjemah); vsaka je shranjena v strukturo `method_info`, njena velikost pa je omejena z 65535 bajti,
- **`attributes_count, attributes[]`** - vsaka vrednost v `attributes` vsebuje strukturo `attribute_info`, ki je sestavljena iz indeksa imena, dolžine atributa in dodatnih informacij o atributu.[11, str. 100]



(a) HEX identifikator razreda. (b) HEX verzija in velikost bazena spremenljivk.

Slika 3.5: HEX reprezentacija razreda.

Če poženemo ukaz `javap -verbose` na primeru 3.3, dobimo sledeče informacije o razredu:

```
1  public class ByteCode
2    SourceFile: "ByteCode.java"
3    minor version: 0
4    major version: 52
5    flags: ACC_PUBLIC, ACC_SUPER
6  Constant pool:
7     #1 = Methodref          #8.#19
8     #2 = Methodref          #7.#20
9     #3 = Fieldref           #21.#22
10    #4 = String             #23
11
12    ...
13
14 {
15   public ByteCode();
16     descriptor: ()V
17     flags: ACC_PUBLIC
18     Code:
19       stack=1, locals=1, args_size=1
20          0: aload_0
21          1: invokespecial #1 // Method java/lang/Object."<
               init>":()V
22          4: return
23       LineNumberTable:
24         line 1: 0
```

```
25    ...
26
27    public static void main(java.lang.String[]);
28      descriptor: ([Ljava/lang/String;)V
29      flags: ACC_PUBLIC, ACC_STATIC
30      Code:
31        stack=2, locals=2, args_size=1
32           0: iconst_1
33
34    ...
35
36    public static int sestej(int, int);
37      descriptor: (II)I
38      flags: ACC_PUBLIC, ACC_STATIC
39      Code:
40        stack=2, locals=3, args_size=2
41           0: iload_0
42      ...
43  }
```

Koda 3.6: Razstavljena koda iz primera 3.3 z uporabo `verbose` značke.

V vrstici 3 vidimo `minor_version` in `major_version`, v vrstici 6 je viden bazen konstant za naš razred, v vrsticah 29 in 38 pa so vidne zastavice za dostopna določila naših razredov, pri razredih `main` in `sestej` sta ti ustrezno `ACC_PUBLIC` in `ACC_STATIC`.

3.5 Nalagalec razredov

Namen nalagalca razredov je podpora pri dinamičnemu nalaganju programskih enot na Javansko platformo. Razrede ustvarijo prevajalniki iz osnovnih javanskih datotek, lahko pa se naložijo v katerikoli javanski virtualni stroj. Razred je lahko shranjen tudi v pomnilniku ali pa pridobljen iz omrežja.[10]

Za nalagalca razredov so značilne naslednje lastnosti:

- **hierarhična struktura** - nalagalci razredov so organizirani v hierarhijo starš-otrok, pri čemer je nalagalec `Bootstrap` starš vseh nalagalcev

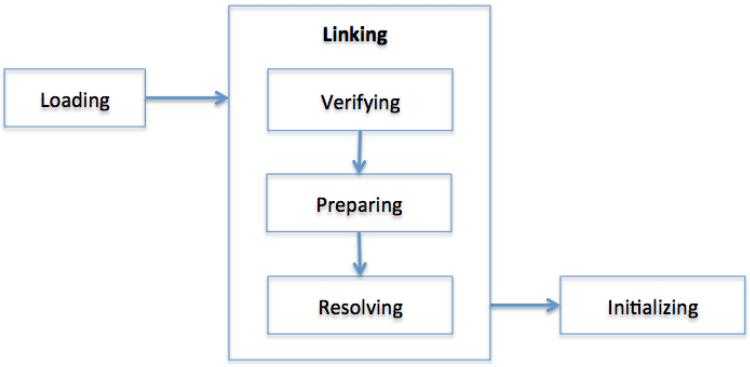
(vidno na sliki 3.9),

- **delegacija** - na podlagi hierarhične strukture se nalaganje razredov porazdeli med nalagalci, ko je razred naložen, trenutni nalagalec preveri ali je razred že naložen pri starših (ne prihaja do podvajanja nalaganj),
- **omejena vidnost** - otrok nalagalca razreda lahko vidi starša, starš pa ne vidi svojih sinov,
- **prekinitev** - prekinitev razreda, ki je že naložen ni dovoljena, za to je potrebno trenutni nalagalec razredov prekiniti in ustvariti novega.

3.5.1 Pregled nalaganja razredov

Nalaganje poteka v petih fazah (vidno na sliki 3.6):

1. **Nalaganje** (*Loading*),
2. **Verifikacija** (*Verifying*),
3. **Priprava strukture** (*Preparing*),
4. **Reševanje referenc** (*Resolving*),
5. **Inicializacija** (*Initializing*).



Slika 3.6: Potek nalaganja razreda.

3.5.1.1 Nalaganje razreda

Razredi se naložijo le kadar jih je potrebno izvesti. Ob nalaganju razreda se naložijo vsi njegovi nadrazredi in vmesniki. Ko nalagamo vmesnik, so naloženi tudi vsi njegovi nadvmesniki. Vsi razredi se naložijo samo enkrat. Kako poteka nalaganje, in v kakšnem vrstnem redu lahko opazujemo z značko `-verbose`.

V sledečem primeru (3.7) ob izdelavi objekta `B` naložimo `B` in vse njegove nadrazrede.

```
1 class A{}
2 class B extends A{}
3 class Test {
4   public static void main(String[] args) {
5     B b = new B();
6   }
7 }
```

Koda 3.7: Nalaganje razreda.

Ob prevajanju in izvedbi z ukazom `java -verbose Test` dobimo sledeč izpis,

```
1 [Loaded java.lang.Error from /usr/lib/jvm/java-8-oracle/jre/
     lib/rt.jar]
2 ...
3 [Loaded Test from file:/home/bostjan/]
4 [Loaded A from file:/home/bostjan/]
5 [Loaded B from file:/home/bostjan/]
6 ...
```

Koda 3.8: Izpis pri izvedbi razreda Test.

ki nam pokaže, da se pred izvajanjem naložijo vsi nadrazredi.

3.5.1.2 Verifikacija

Verifikacija poteka na bajtni kodi. Njena naloga je:

- preveriti, če je bajtna koda pravilno strukturirana,

- da je cilj vsakega `goto` stavka inštrukcija,
- da sklad ne preseže alociranega pomnilnika,
- da so spoštovana pravila JVM (tipizacija).

Prve tri naloge niso direktno vidne programerjem, medtem ko se na četrto lahko vpliva. Bajtna koda vsebuje opise tipov, metod in spremenljivk (3.4.2). Verifikacija samo preverja hierarhijo dedovanja, podtipe za prejemnika in argumenete metod, ne preveri pa podtipov za lokalne spremenljivke.
Kjer so zahtevani podtipi pri vmesnikih, verifikacija ne preveri če je T podtip T. Če je T_2 vmesnik ne preverja če je T_1 podtip T_2.

Kadar hoče verifikacija preveriti, če je T_1 podtip T_2, najprej pogleda že obstoječe naložene razrede. Če vsaj eden izmed T_1 ali T_2 še ni naložen, jih bo verifikacija naložila in preverila razmerje med njima.

Obnašanje verifikacije lahko preverimo z zastavico `-noverify`:

```
1 ...
2 [Loaded Test from file:/home/bostjan/]
3 1
4 [Loaded C from file:/home/bostjan/]
5 [Loaded A from file:/home/bostjan/]
6 [Loaded B from file:/home/bostjan/]
7 ...
```

Koda 3.9: Uporaba `-verbose` na kodi 3.11.

```
1 ...
2 [Loaded Test from file:/home/bostjan/]
3 1
4 [Loaded C from file:/home/bostjan/]
```

Koda 3.10: Uporaba `-noverify` na kodi 3.11.

```
1 class A { }
2 class B extends A { }
```

```
3  class C {
4    void m1(A a) { }
5    void m2( ) { m1(new B()); }
6  }
7  class Test {
8    public static void main(String[] args) {
9      System.out.println(1);
10     C c = new C();
11   }
12 }
```

Koda 3.11: Testni primer za verifikacijo.

Verifikacija razreda zahteva tudi verifikacijo njegovih nadrazredov. Če poženemo primer 3.12 z in brez zastavice `noverify`, dobimo izpis 3.14 in 3.13.

```
1  class A { }
2  class B extends A { }
3  class C extends A { }
4  class D {
5    void m1(A a) { }
6  }
7  class E {
8    void m2() { new D().m1(new B()); }
9  }
10 class F extends E {
11   void m3() { new D().m1(new C()); }
12 }
13 class Test {
14   public static void main(String[] args) {
15     F f = new F();
16   }
17 }
```

Koda 3.12: Testni primer za verifikacijo nadrazredov.

```
1  ...
2  [Loaded Test from file:/home/bostjan/]
3  [Loaded E from file:/home/bostjan/]
4  [Loaded F from file:/home/bostjan/]
5  [Loaded A from file:/home/bostjan/]
6  [Loaded B from file:/home/bostjan/]
7  [Loaded C from file:/home/bostjan/]
8  ...
```

Koda 3.13: Uporaba `-verbose` na kodi 3.12.

Ker pri metodi `m3` razreda `E` uporabljamo razred `B`, verifikacija razreda `E` zahteva nalaganje razredov `A` in `B`. Ker je `F` podrazred razreda `E`, verifikacija razreda `F` zahteva, da sta naložena razreda `A` in `C`. Ko je verifikacija vključena, je klicana na `E` in `F` preden se ustvari objekt `F` v metodi `main`. Med tem procesom se naložijo razredi `A`, `B` in `C`. Razred `D` se ne naloži, ker ga nikjer implicitno ne pokličemo.[15]

```
1 ...
2 [Loaded Test from file:/home/bostjan/]
3 [Loaded E from file:/home/bostjan/]
4 [Loaded F from file:/home/bostjan/]
5 ...
```

Koda 3.14: Uporaba `-noverify` na kodi 3.12.

Verifikacija je tudi "optimistična" (angl. optimistic) kar pomeni, da pri nalaganju tipa T ne preverjamo, če je podtip samega sebe. Verifikacija tudi ne preverja, če razred implementira vmesnik. V primeru 3.15 ob izpisu z značko `verbose` dobimo le `[Loaded Test]`.

```
1 class A { }
2 class Test {
3   A m() { return new A(); }
4   public static void main(String[] args) { }
5 }
```

Koda 3.15: Testni primer za verifikacijo samega sebe.

V primeru 3.16 se ob ustvarjanju instance objekta `D` zahteva verifikacija `D`. V razredu `D` se metoda `m1` z argumentom tipa `I` kliče v metodi `m2`, njen parameter pa je razred `B`. Ko verifikator vidi da je `I` vmesnik, se ne preverja, če razred `B` implementira `I` in tako tudi ne naloži razredov `A` in B. Izhod z značko `verbose` po vrsti izpiše `Test`, `D` in `I`.

```
1 interface I { }
2 class A implements I { }
3 class B extends A { }
4 class D {
5    void m1(I i) { }
6    void m2( ) { m1(new B()); }
7 }
```

Koda 3.16: Testni primer za verifikacijo vmesnikov.

3.5.1.3 Priprava strukture

Priprava določi sestavo razreda, ustvari metode in preiskovalne tabele, ki vsebujejo imena spremenljivk in konstante, celotni strukturi določi tudi pomnilnik. V primeru, da struktura prekorači dovoljeno količino pomnilnika, Java vrne `OutOfMemory` izjemo.

3.5.1.4 Reševanje referenc

Binarne datoteke (prevedene Java datoteke) vsebujejo reference na druge razrede, polja, metode in vmesnike. Ta korak preveri, če so reference pravilne. Reševanje referenc lahko vrže napako če:

- spremenljivka ali metoda določenega tipa ne obstaja v razredu,
- metoda iz vmesnika vsebuje isto ime kakor razred,
- metoda ali spremenljivka iz razreda pripada vmesniku.

Java ob napaki običajno vrže izjemo `IncompatibleClassChangeError` ali eno izmed sledečih:

- `InstantiationError`; se pojavi ob spremembi razreda v vmesnik, spremembi vmesnika v razred, spremembi razreda v abstraktni razred,
- `NoSuchFieldError` - če navedena spremenljivka ne obstaja,
- `NoSuchMethodError` - če navedena metoda ne obstaja,

- `IllegalAccessError` - samo, če je verifikator že bil pognan,
- `UnsatisfiedLinkError`

3.5.1.5 Primer nalaganja razreda

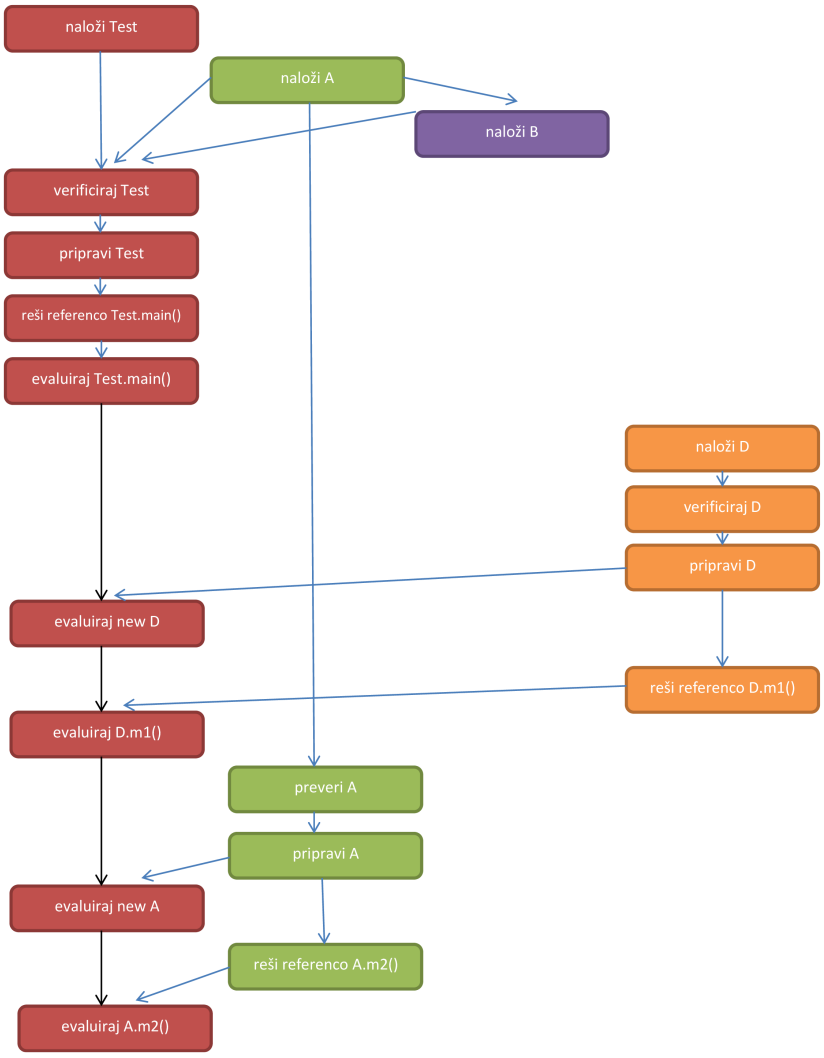
Na primeru 3.17 vidimo vse zgoraj naštete faze nalaganja in odvisnosti razredov med seboj.

```
1  class A {
2    public void m2() { }
3  }
4  class B extends A {
5    public void m3() { A a = new C(); a.m2(); }
6  }
7  class C extends A { }
8  class D {
9    public void m1() { }
10 }
11 class Test {
12   public static void main(String[] args) {
13     new D().m1();
14     new A().m2();
15   }
16   void g() { A a = new B(); a.m2(); }
17 }
```

Koda 3.17: Primer za nalaganje razreda.

Na sliki 3.7 vidimo potek izvajanja programa v primeru 3.17. Modre puščice označujejo, da naslednji razred, na katerega puščica kaže, potrebuje prejšnjega, črna puščica pa označuje, da prvemu koraku sledi drug korak.

Da uspešno izvedemo `Test.main()`, se mora evaluacija izvesti po v zaporedju `Test.main()`, `new D()`, `D.m1()`, `new A()` in `A.m2()`.



Slika 3.7: Diagram nalaganja razredov po primeru 3.17.

Klic metode main zahteva, da je `Test` že pripravljen, kar pomeni, da mora biti tudi verificiran. Verifikacija `Test` zahteva, da je bil `Test` naložen in da

je `B` podrazred `A`. Da to ugotovimo, je potrebno da sta razreda `A` in `B` (pred nalaganjem le tega je potrebno, da je `A` že naložen) že uspešno naložena. Razred `B` je naložen za potrebe verifikacije razreda `Test`, vendar ga ni potrebno verificirati. Ker razreda `B` ni potrebno verificirati, razreda `C` ne naložimo.
V spodnji tabeli vidimo dva možna načina izvajanja razreda `Test`, leno (angl. lazy execution) in požrešno (angl. eager execution). Java pri nalaganju razredov uporablja prvi pristop.[16, str. 141].

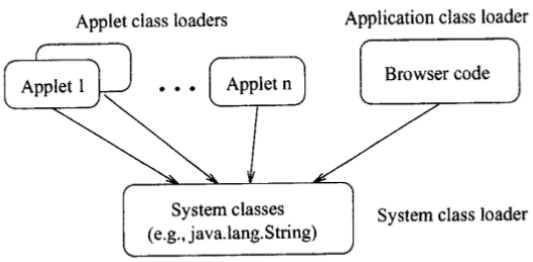
	Leno izvajanje	**Požrešno izvajanje**
1	`load Test`	`load Test`
2	`load A`	`load A`
3	`load B`	`load B`
4	`verify Test`	`verify Test`
5	`prepare Test`	`verify A`
6	`resolve Test.main()`	`load C`
7	`evaluate Test.main()`	`verify B`
8	`load D`	`verify C`
9	`verify D`	`load D`
10	`prepare D`	`verify D`
11	`evaluate new D()`	`prepare Test`
12	`resolve D.m1()`	`prepare A`
13	`evaluate D.m1()`	`prepare B`
14	`verify A`	`prepare C`
15	`prepare A`	`prepare D`
16	`evaluate new A()`	`resolve Test.main()`
17	`resolve A.m2()`	`evaluate Test.main()`
18	`evaluate A.m2()`	`evaluate new D()`
19	-	`resolve D.m1()`
20	-	`evaluate D.m1()`
21	-	`evaluate new A()`
22	-	`resolve A.m2()`
23	-	`evaluate A.m2()`

Tabela 3.2: Primerjava lenega in požrešnega izvajanja programa 3.17.

3.5.2 Uporaba večih nalagalcev razredov

Javanska aplikacija lahko uporablja več različnih nalagalcev razredov, kjer lahko vsak upravlja z različnimi komponentami programske opreme, ki smo

jo napisali. Na sliki 3.8 vidimo dva različna nalagalca, eden je definiran s strani programerja, drugi pa je sistemski, ki ga upravlja JVM.



Slika 3.8: Nalaganje razredov v spletni aplikaciji.

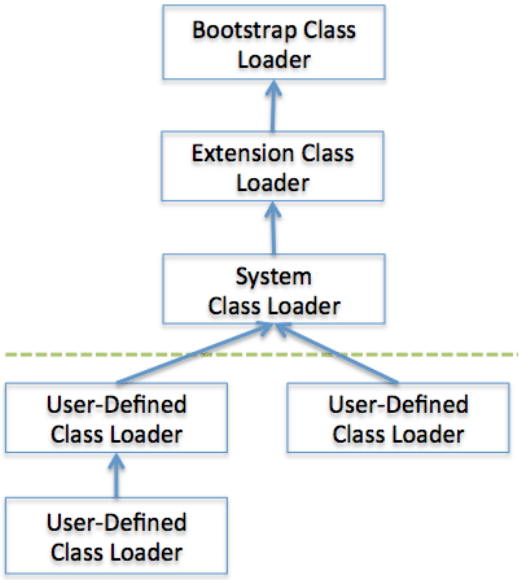
Puščice na sliki 3.8 nakazujejo na razmerje med nalagalci razredov (delegacija, 3.5). Nalagalec razreda L_1 lahko zahteva od drugega nalagalca razreda L_2 da naloži razred C v njegovem imenu. V takšnem primeru L_1 delegira C nalagalcu L_2. Na sliki 3.8 nalagalci spletnih aplikacij in appletov delegirajo vse sistemske razrede sistemskemu nalagalcu, posledično so sistemski razredi deljeni med appleti in aplikacijo samo. S tem zagotavljamo tudi varnost tipov, ker bi lahko drugače nalagalci definirali osnovne tipe kot so `java.lang.String`.

Na sliki 3.9 je viden delegacijski model nalagalcev[13]. Ta je sestavljen iz:

- **Nalagalca Bootstrap** - ustvari se ob zagonu JVM, naloži Java API-je,
- **Nalagalca razširitev** - naloži različne varnostne razširitve in Java API-je brez osnovnih tipov (teh, ki jih je že naložil Bootstrap),
- **Sistemskega nalagalca** - naloži razrede aplikacije v `$CLASSPATH`, ki ga poda uporabnik,

- **Uporabniškega nalagalca** - nalagalec, ki ga je ustvaril programer v aplikaciji sami.

Ko se zahteva nalaganje razreda, sam nalagalec najprej preveri, če razred že obstaja v predpomnilniku, staršu trenutnega nalagalca ali njemu samemu. Če razreda ne najde tudi v *Bootstrap* nalagalcu, trenutni nalagalec poišče razred in ga naloži (opisano v 3.5.1).



Slika 3.9: Hierarhija nalaganja razredov.

Z delegacijo ohranjamo imenski prostor (3.4.1) hkrati pa lahko delimo osnovne razrede med različnimi nalagalci.

Ko en nalagalec razredov delegira nalaganje drugemu, ni nujno da je tisti nalagalec, ki je zahteval nalaganje tudi tisti, ki je nalaganje izvedel. Za primer vzemimo sledečo kodo:

```
1 MyClassLoader cl = new MyClassLoader("/some");
2 Class stringClass = cl.loadClass("java.lang.String");
```

Koda 3.18: Nalaganje sistemskega razreda.

Instance razreda `MyClassLoader` delegirajo nalaganje `java.lang.String` sistemskemu nalagalcu. Posledično je nalagalec `java.lang.String` sistemski, čeprav je bilo nalaganje razreda naročeno iz nalagalca `cl`.

3.5.3 Primer ustvarjanja nalagalca razreda

Nalagalec razreda, ki ga definira uporabnik, je vedno podrazred razreda `ClassLoader`. Podrazredi lahko povozijo metodo `loadClass` s katero definirajo svoj nalagalec razredov[19]. V kodi 3.19 vidimo nalagalec, ki naloži razrede v podanem direktoriju.[10]

```
1  private directory;
2
3  public CustomClassLoader(String dir) {
4    directory = dir;
5  }
6
7  public synchronized Class loadClass(String name) {
8    Class c = findLoadedClass(name);
9    if(c != null) {
10     return c;
11   }
12   try {
13     c = findSystemClass(name);
14     return c;
15   } catch (ClassNotFoundException e) {
16     // keep looking
17   }
18   try {
19     byte[] data = getClassData(directory, name);
20     return defineClass(name, data, 0, data.length());
21   } catch (IOException e) {
22       throw new ClassNotFoundException();
23   }
24 }
25
26 private byte[] getClassData(String name) throws IOException {
```

```
27
28    InputStream stream = getClass().getClassLoader().
          getResourceAsStream(name);
29    int size = stream.available();
30    byte buff[] = new byte[size];
31
32    DataInputStream in = new DataInputStream(stream);
33    in.readFully(buff);
34    in.close();
35
36    return buff;
37  }
```

Koda 3.19: Preprost nalagalec razredov.

Konstruktor zgornjega nalagalca razredov sprejme direktorij, iz katerega bomo naložili razrede. Metoda `loadClass` naloži razred. Pred nalaganjem je potrebno preveriti, ali je razred že naložen, za to uporabimo metodo `findLoadedClass`. Če ta vrne `null`, je potrebno preveriti, če je razred, ki ga nalagamo sistemski (metoda `findSystemClass`). Šele če obe preverjanji ne vrnejo razreda, kličemo metodo `getClassData`, ki dobi podatke razreda, ki ga nalagamo. S klicem `defineClass` konstruiramo reprezentacijo razreda. Da se izognemo nalaganju istega razreda ob istem času (niti) uporabimo pri metodi `loadClass` ključno besedo `synchronized`.

3.5.4 Refleksija

Refleksija omogoča programu, da analizira trenutno programsko okolje in spremeni program glede na dan scenarij. Za uspešno analizo potrebujemo reprezentacijo programa, ki se izvaja, tej informaciji pravimo metapodatki. V objektno orientiranih jezikih so metapodatki organizirani v metaobjekte, analiza izvajanja pa se imenuje introspekcija. S pomočjo refleksije dosežemo fleksibilnost programske opreme, hkrati pa se aplikacije lažje prilagajajo tekočim spremembam.[16]

Introspekciji sledi sprememba obnašanja programa, katero lahko izvedemo na tri načine:

- direktna sprememba metaobjektov,
- posredovanje v različnih fazah programa,
- dinamična invokacija metod.

V primeru 3.20 prikažemo osnove uporabe refleksije v Javi (invokacija). Najprej pridobimo razred `AppTest` iz ustreznega paketa, nato naredimo instanco tega razreda. S pomočjo objekta `Method` in metode `invoke` kličemo metodo tega razreda z ustreznimi parametri.

```
1  package com.test.reflection;
2
3  public class AppTest {
4
5    public void print() {
6      System.out.println("Print - no params");
7    }
8
9    public void printSomething(String something) {
10     System.out.println("Printing , added string: "+something);
11   }
12
13 }
14
15 public class ReflectApp {
16
17   public static void main(String[] args) {
18
19     // v try catch bloku ...
20
21     Class noparams[] = {};
22     Class[] paramString = new Class[1];
23     paramString[0] = String.class;
24
25     Class cls = Class.forName("com.test.reflection.AppTest");
26     Object obj = cls.newInstance(); // Naredimo novo instanco
           zgornjega razreda
27
28     Method method = cls.getDeclaredMethod("print", noparams);
           // Pridobimo metodo print, kateri ne dodamo
          parametrov
```

```
29      method.invoke(obj, null); // klicemo metodo print
30
31      method = cls.getDeclaredMethod("printSomething",
            paramString); // Pridobimo metodo print in ji dodamo
            parameter String
32      method.invoke(obj, new String("natisni nekaj")); //
            klicemo metodo printSomething
33
34      // ...
35
36    }
37
38  }
```

Koda 3.20: Primer uporabe refleksije v Javi

Pri večjih aplikacijah se zaradi kompleksnosti realizirane refleksije v Javi poveča število vrstic kode, hkrati pa aplikacije zgubijo preglednost. Zaradi tega danes obstajajo naprednejše knjižnice, ki omogočajo lažjo in preglednejšo uporabo refleksije, ena izmed njih je *ReflectionSupport*.[14]

3.5.5 Instrumentacija razredov

Ena izmed značilnosti dinamičnega nalaganja v Javi (poleg nalaganja razredov ob izvajanju programa) je tudi možnost spreminjaja razreda med nalaganjem. Ta proces imenujemo instrumentacija. Dokler se programer drži samega formata datoteke, lahko po nalaganju razreda spreminja in dodaja metode, spreminja in dodaja spremenljivke, pri odstranjevanju pa je potrebno paziti, saj obstaja možnost, da se razred ne bo uspešno povezal z ostalimi.

Nalagalec razredov lahko spreminja le tiste razrede, ki jih je naložil sam. Četudi na novo definiramo razred `String`, tega razreda ne moremo uporabiti pri metodi, ki sprejema javanski `String`, ker bo JVM na to napako opozoril in prožil izjemo.

3.5.6 Ohranjanje varnosti tipov

Ker Java podpira več nalagalcev razredov, je preverjanje varnosti tipov težje opravilo. V času prevajanja vsak statični tip pripada imenu razreda. Ob zaganjanju pa nalagalci uvedejo več imenskih prostorov. Razred, ki ga poganjamo, definirajo njegovo ime in ime njegovega nalagalca.

3.5.6.1 Lokalna nekonsistenca varnosti tipov

Metoda `loadClass` lahko vrne različne tipe razredov glede na podano ime v podanem času. Da JVM ohranja konsistenco tipov, mora vedno vračati isto reprezentacijo razreda. V primeru 3.21 vidimo, da če bi nalagalec razreda `C` mapiral dve pojavitvi razreda `X` v različna tipa, bi bila ogrožena varnost tipa v metodi `g`.

```
1 class C {
2   void f(X x) { ... }
3   ...
4   void g() {
5     f(new X());
6   }
7 }
```

Koda 3.21: Ohranjanje varnosti tipov.

JVM se ne more zanašati na to, da nalagalci, ki so ustvarjeni s strani uporabnika, vračajo vedno isti tip na podano ime. JVM zaradi tega shranjuje, katere razrede je že naložil in kateremu nalagalcu pripadajo. Ko JVM pridobi razred iz metode `loadClass` opravi naslednje operacije:

- preveri se, če se pravo ime razreda ujema s tistim, ki ga prejema metoda `loadClass` (drugače se proži izjema),
- če se ime ujema, si JVM zapomni, da ga je naložil (shrani ga v predpomnilnik); JVM metode `loadClass` nikoli ne kliče na razredih, ki jih je že naložil (če se uporablja isti nalagalec razredov).

3.5.6.2 Konsistenca med delegiranimi nalagalci

Ko uporabljamo več nalagalcev, lahko pride do težav z varnostjo tipov. Naj predstavimo tip razreda z notacijo $\langle C, L_d \rangle^{L_i}$, kjer C označuje ime razreda, L_d njegov nalagalec in L_i nalagalec, ki je prožil nalaganje. Ko nas ne zanima delegirani nalagalec, uporabimo preprostejšo notacijo C^{L_i}, kjer označimo, da je L_i nalagalec razreda C. Ko nas ne zanima nalagalec, ki je prožil nalaganje, uporabimo $\langle C, L_d \rangle$, da označimo, da je C definiran z L_d. Če L_1 delegira nalaganje C L_2, potem velja, da je $C^{L_1} = C^{L_2}$.

Da pokažemo nevarnost varnosti tipov, uporabimo sledeč primer:[10]

```
1 class ⟨C, L1⟩ {
2   void f() {
3     ⟨Spoofed, L1⟩^L1 x = ⟨Delegated, L2⟩^L1 .g();
4   }
5 }
6 class ⟨Delegated, L2⟩ {
7   ⟨Spoofed, L2⟩^L2 g() { ... }
8 }
```

Koda 3.22: Ohranjanje varnosti tipov.

Razred C definira nalagalec L_1, posledično je L_1 uporabljen za inicializacijo nalaganja razredov $Spoofed$ in $Delegated$, ki sta referencirana v metodi `f` v razredu C. L_1 definira $Spoofed$, vendar L_1 delegira nalaganje $Delegated$ L_2, ki potem definira $Delegated$. Ker je $Delegated$ definiran z L_2, bo $Delegated$`.g()` uporabil L_2 za inicializacijo nalaganja $Spoofed$. L_2 definira drugačen tip $Spoofed$. C pričakuje instanco $\langle Spoofed, L_1 \rangle$ iz $Delegated$`.g()`, vendar $Delegated$ vrača instanco $Spoofed, L_2$, ki je drug razred od pričakovanega.

To je nekonsistenca med imenskimi prostori L_1 in L_2. Če to dovolimo, lahko tipe ponarejamo kot druge tipe z uporabo delegacijskih nalagalcev:[10]

```
1 class ⟨Spoofed, L1⟩ {
2   public int secret_value;
3   public int[] forged_pointer;
4 }
5 class ⟨Spoofed, L2⟩ {
6   private int secret_value;
7   private int forget_pointer;
8 }
```

Koda 3.23: Razred z istim imenom naložen z L_1 in L_2.

Razred $\langle C, L_1 \rangle$ lahko odkrije vrednost privatne spremenljivke instance $\langle Spoofed, L_2 \rangle$ in dereferencira kazalec:

```
1 class ⟨C, L1⟩ {
2   void f() {
3     ⟨Spoofed, L1⟩^L1 x = ⟨Delegated, L2⟩^L1.g();
4     System.out.println("secret value = " + x.secret_value);
5     System.out.println("stolen content = " + x.forged_pointer
          [0]);
6   }
7 }
```

Koda 3.24: Ponarejanje tipov in pridobivanje privatnih spremenljivk.

Dostopamo lahko do privatne spremenljivke `secret_value` v instanci $\langle Spoofed, L_2 \rangle$, ker je ta spremenljivka deklarirana kot `public` v $\langle Spoofed, L_1 \rangle$. Zaradi dostopnega določila `public` v spremenljivki `forged_pointer` lahko tudi dereferenciramo kazalec. Na to težavo je opozoril *Vijay Saraswat*[2], njena rešitev pa je razložena v 3.5.6.3.

3.5.6.3 Reševanje ponarejanja tipov

Ključna ideja pri reševanju težave ponarejanja tipov je hranjenje setov pravil nalagalcev, ki se dinamično posodabljajo skozi nova nalaganja razredov. V primeru 3.22 namesto nalaganja $Spoofed$ v L_1 in L_2 shranimo pravilo $Spoofed^{L_1} = Spoofed^{L_2}$. Če $Spoofed$ kasneje naloži L_1 ali L_2, je potrebno preveriti, da ne kršimo obstoječih setov pravil.

Pri temu načinu se držimo trditve, da vsak razred, ki je trenutno naložen v JVM ustreza vsem pravilom. To ohranjamo na sledeč način:

- vsakič ko dodajamo nov razred, preverimo, da ne kršimo obstoječih pravil, če razreda ne moremo naložiti, vržemo izjemo,
- vsakič ko dodamo novo pravilo, moramo preveriti, da vsi že naloženi razredi ustrezajo tudi novemu pravilu; če novo pravilo ne ustreza vsem že naloženim razredom, se operacija, ki je prožila dodajanje novega pravila, prekine.

V primeru 3.22 izvajanje metode `f` povzroči nalaganje pravila $Spoofed^{L_1} = Spoofed^{L_2}$. Če sta L_1 in L_2 že naložila razred $Spoofed$, ko smo generirali pravilo, bo Java prožija izjemo, drugače bo pravilo uspešno dodano. Če $Delegated$`.g()` najprej naloži $Spoofed^{L_2}$, bo prožena izjema, ko bo C`.f()` kasneje poskusil naložiti $Spoofed^{L_1}$.

Kadar en razred referencira drug razred in sta oba bila naložena z različnimi nalagalci uporabljamo sledeča pravila:[10]

- Če $\langle C, L_1 \rangle$ referencira spremenljivko,

  $$\mathrm{T}\ \textit{ime spremenljivke};$$

  ki je deklarirana v razredu $\langle D, L_2 \rangle$, potem generiramo pravilo:

  $$T^{L_1} = T^{L_2}.$$

- Če $\langle C, L_1 \rangle$ referencira metodo,

  $$T_0\ \textit{ime metode}\ (T_1, \ldots, T_n);$$

  ki je deklarirana v razredu $\langle D, L_2 \rangle$, potem generiramo pravila:

  $$T_0^{L_1} = T_0^{L_2}, \ldots, T_n^{L_1} = T_n^{L_2}.$$

- Če $\langle C, L_1 \rangle$ povozi metodo,

$$T_0 \; ime\; metode \; (T_1, \ldots, T_n);$$

ki je deklarirana v razredu $\langle D, L_2 \rangle$, potem generiramo pravila:

$$T_0^{L_1} = T_0^{L_2}, \ldots, T_n^{L_1} = T_n^{L_2}.$$

Par omejitev $\{T^{L_1} = T^{L_2}, T^{L_2} = T^{L_3}\}$ označuje, da mora biti T naložen kot isti tip, ki je bil naložen z L_1 in L_2 in L_3. Če med izvajanjem programa T ne naložimo z L_2, potem T tudi ne moremo naložiti z L_1 in L_3. Če so pravila kršena, Java proži izjemo `java.lang.LinkageError`. Pravila se odstranijo, kadar je odstranjen nalagalec.

Formalni dokaz zgornje implementacije (narejeno v JDK 1.2) sta opravila *Tozawa in Hagiya*, v postopku dokazovanja pa sta odkrila napako pri implementaciji (odpravljena v JDK 1.3):[17]

```
1 public class D {
2   static boolean t = true,
3   public D() {
4     A a;
5     if (t) a = new B(); else a = new C();
6     a.speakUp();
7   }
8 }
```

Koda 3.25: Preprečevanje ogrožanja varnosti tipov (napaka).

Preverjanje v JDK 1.2 ne naloži razreda `A` (čeprav JVM to specifično določa), zato vrednost, ki jo ustvarimo s klicem `new B()`, ni preverjena.

Poglavje 4

Načrtovanje rešitve

4.1 Orodja

V načrtovalni fazi smo uporabili orodji Sybase PowerDesigner in Pencil.

4.1.1 Sybase PowerDesigner

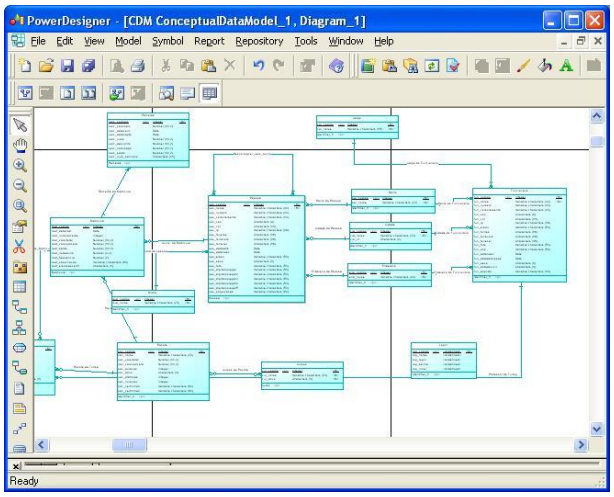
PowerDesigner (slika 4.1 je orodje, ki se uporablja pri modeliranju sistemov. Razvilo ga je podjetje Sybase, njegov tržni delež pa dosega 39%. Na voljo je za operacijski sistem Microsoft Windows, možna pa je tudi uporaba v razvojnem orodju Eclipse (kot vtičnik).

Orodje smo uporabili za načrtovanje diagrama primerov uporabe in diagrama hierarhije funkcij.

4.1.2 Pencil

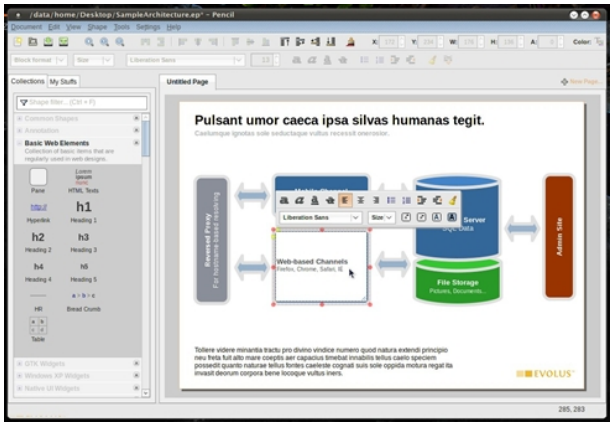
Pencil je odprtokodno orodje (licenca GPL), namenjeno izdelovanju prototipov uporabniških vmesnikov in izdelavi različnih diagramov. Na voljo je v samostojni različici ali kot dodatku za brskalnik Firefox. Trenutno je na voljo v verziji 2.0.5 in omogoča:

- izdelavo diagramov in prototipov uporabniških vmesnikov,



Slika 4.1: Orodje Sybase PowerDesigner.

- izvoz v HTML, PNG, PDF, OpenOffice dokumente,
- dodajanje zunanjih objektov.



Slika 4.2: Orodje Pencil.

Orodje smo uporabili pri načrtovanju uporabniškega vmesnika in za izdelavo papirnatih prototipov.

4.2 Zajem zahtev

4.2.1 Predpogoji

Za uporabo aplikacije je potrebna izpolnitev naslednjih predpogojev:

- nameščena ustrezna Linux distribucija,
- za poganjanje strežnika nameščena Java 1.7 ali novejša ter podatkovna baza MySQL 5.1 ali novejša,
- poznavanje osnovne uporabe terminala,
- odprta vrata 10355 in
- poznavanje osnovne uporabe usmerjevalnika.

4.2.2 Funkcionalne zahteve

Aplikacije bo uporabljala princip strežnik-odjemalec, zato so naloge tudi ustrezno razdeljene.

4.2.2.1 Strežnik

Strežnik predstavlja osrčje programa, ki sprejema ukaze odjemalca in izvaja ustrezne akcije na ustreznih razširitvah.

1. **Prijava uporabnika**
    - preveri, če uporabniško ime obstaja,
    - preveri, če se geslo ujema z geslom v podatkovni bazi,
    - preveri, če je uporabnik presegel število nepravilnih prijav (omejitev so tri prijave na trideset minut),
    - pridobi vlogo uporabnika in njegove pravice,
    - preveri če je uporabnik blokiran,

- preveri, če katera razširitev doda dodatne pogoje k prijavi,
- vzpostavi sejo.

2. **Urejanje uporabnikov**
   - dodeljevanje pravice urejanja le tistih uporabnikov, do katerih strežnikov ima trenutni uporabnik pravico,
   - pridobivanje seznama uporabnikov,
   - brisanje uporabnikov,
   - urejanje uporabniških pravic.
3. **Vzdrževanje vzpostavljenih sej**
   - ob novem zahtevku podaljšaj aktivnost seje,
   - brišejo se seje, pri katerih je čas od preteka zadnjega zahtevka večji od dvajset minut,
   - superadministrator lahko vzdržuje vse seje na vseh strežnikih,
   - administrator lahko vzdržuje seje na njemu dodeljenih strežnikih.
4. **Dodajanje strežnika**
   - dodaj nov strežnik v omrežje,
   - preveri, če je uporabnik superadministrator,
   - določi unikatno ime strežnika,
   - določi IP naslov strežnika.
5. **Urejanje strežnika**
   - preveri, če je uporabnik superadministrator,
   - določi novo ime strežnika,
   - določi nov IP strežnika,
   - izbriši srežnik iz omrežja.

6. **Dodajanje razširitve**
    - preveri, če je uporabnik superadministrator ali administrator,
    - če je superadministrator, lahko doda razširitev na katerikoli strežnik,
    - če je administrator, lahko doda razširitev le na njemu dodeljenih strežnikih.
7. **Odstranjevanje razširitve**
    - preveri, če je uporabnik superadministrator ali administrator,
    - če je superadministrator, lahko odstrani razširitev na kateremkoli strežniku,
    - če je administrator, lahko doda razširitev le na njemu dodeljenih strežnikih.
8. **Ustvarjanje dnevnikov**
    - na vsakem strežniku se zapisujejo dnevniki in vse akcije vsakega uporabnika,
    - na glavnem strežniku se zapisujejo dnevniki ukazov in lokacije, kamor so bili ti ukazi poslani (hkrati se zapiše tudi ura in uporabnik).
9. **Odziv strežnika odjemalcu**
    - vsi odzivi strežnika se vračajo v formatu JSON,
    - v tem odzivu so navedena tudi sporočila klientu.
10. **Avtentifikacija odjemalca**
    - preveri, če odjemalec ne more vzpostaviti stalne povezave,
    - če stalna povezava ni možna, generiraj ključ, s pomočjo katerega se vzpostavljajo nadaljnje povezave.

4.2.2.2 Odjemalec

Odjemalec je grafični uporabniški vmesnik, ki omogoča prijave uporabnikom in v ozadju izdeluje ukaze in jih pošilja strežniku v izvedbo.

1. **Prijava uporabnika**
   - pošlji podatke za prijavo na strežnik,
   - preberi odziv strežnika in preveri, če je prijava uspela.
2. **Avtentifikacija odjemalca**
   - pošlji podatke strežniku za nadaljevanje seje,
   - preberi odziv strežnika.
3. **Pošiljanje ukazov na strežnik**
   - generiranje ukaza na odjemalcu,
   - pošiljanje ukaza na strežnik ustreznemu vtičniku,
   - ustrezna interpretacija odziva in prikaz uporabniku.
4. **Branje odzivov strežnika**
   - branje JSON odzivov,
   - predstavitev podatkov v uporabniku razumljivi obliki.
5. **Izbira strežnika**
   - po uspešni prijavi prikaži strežnike, ki jih lahko uporabnik nadzira,
   - omogoči izbiro strežnika, na katerega se bodo ukazi pošiljali.

4.2.3 Nefunkcionalne zahteve

Varnost

Zaradi nadzora nad celotno strežniško infrastrukturo se zahteva visoka varnost. Potrebno je beleženje vseh akcij uporabnikov. Administrativni vmesnik omogoča celoten nadzor nad vsemi strežniki, zato mora biti ustrezno zaščiten pred vdori. Ker se dodatne funkcionalnosti lahko dodajajo preko razširitev, je zahtevano varno okolje z omejenimi pravicami za izvajanje teh akcij.

Dosegljivost in odzivnost

Zaradi nadzora nad aplikacijami na strežniku mora aplikacija biti vedno dosegljiva in odzivna, možna so le odstopanja v vrednosti enega procenta ali manj (ponovno zaganjanje strežnika in redna servisiranja).

Sočasni dostop

Aplikacijo bo sočasno uporabljalo 2000 ali več uporabnikov, zato mora aplikacija ustrezno servisirati toliko uporabnikom ali več.

Prokol in vrata

Aplikacija uporablja vrata 10355, na katerih sprejema JSON zahtevke in tudi odgovarja z JSON zahtevki.

4.2.4 Identifikacija uporabnikov

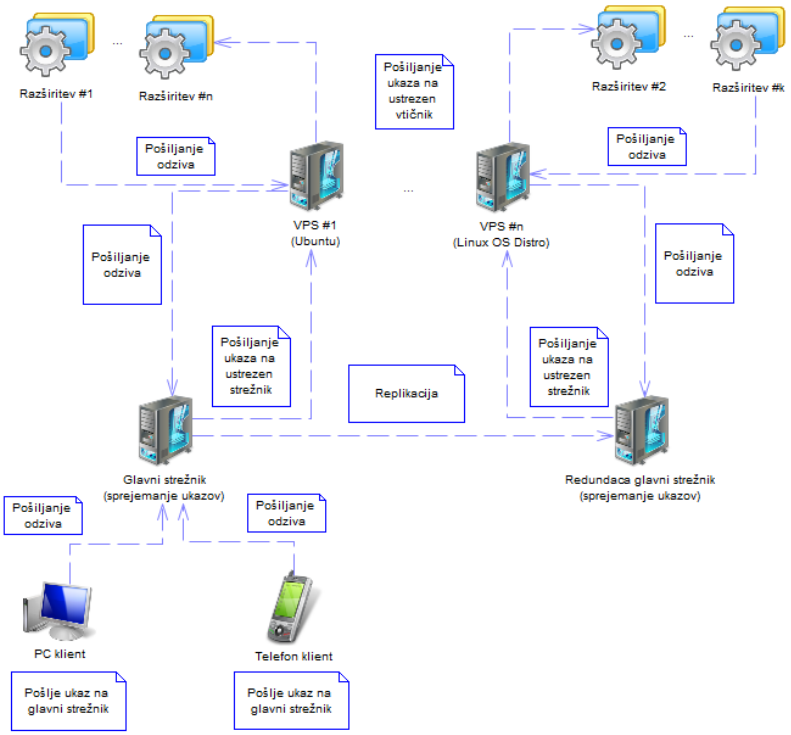
V našemu sistemu imamo več uporabniških vlog. Vnaprej določene vloge so administrator, superadministrator in uporabnik. Identifikacija poteka preko uporabniškega imena in gesla, nujno pa se zahteva tudi možnost razširjanja preverjanja avtentičnosti uporabnika z drugimi viri (kot so npr. preverjanje certifikata ali TOTP[1]).

[1]Ena izmed znanih implementacij algoritma je Google Authenticator.

4.3 Arhitektura aplikacije

Aplikacija bo delovala na principu strežnik-odjemalec (osnovna shema sistema je prikazana na sliki 4.3). Strežnik bo namenjen sprejemanju in izvajanju ukazov, ki jih bo pošiljal odjemalec (podobno kot telenet). Prednosti takšne arhitekture so avtonomija podatkovne baze, večja varnost in neodvisnost grafičnega uporabniškega vmesnika.

Takšna arhitektura nam omogoča tudi platformno neodvisnost. Ker strežnik deluje na podlagi sprejemanja in interpretiranja ukazov, lahko odzive pošiljamo na poljubno napisan klient. Klient je lahko napisan v kateremkoli programskem jeziku, ki podpira uporabo vtičev in omogoča varno komunikacijo.



Slika 4.3: Arhitektura aplikacije.

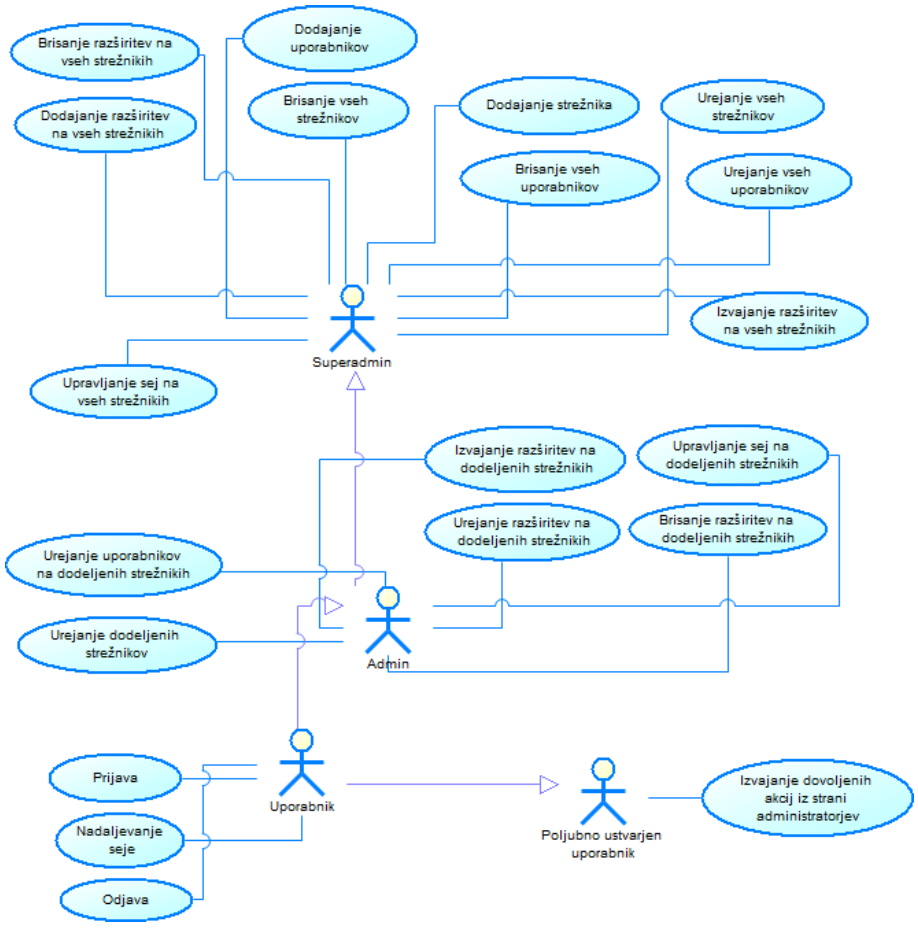
4.3.1 Diagram hierarhije funkcij

Diagram hierarhije funkcij nam služi za prikaz razdelitve funkcionalnosti. Vsaka hierarhična struktura se začne na vrhu z eno samo vseobsegajočo enoto (koren strukture) in se nadaljuje vse do listov, ki predstavljajo elementarne funkcionalnosti. Na sliki 4.4 je prikazan diagram hierarhije funkcij našega sistema.

Slika 4.4: Diagram hierarhije funkcij našega sistema.

4.3.2 Diagram primerov uporabe

Z modelom primerov uporabe dokumentiramo obnašanje obravnavanege sistema. Ta vsebuje funkcije sistema (primeri uporabe), njegovo okolje (akterje) in odnose med primeri uporabe in akterji (diagram primerov uporabe). Najvažnejša vloga modela je komunikacija; omogoča izmenjavo mnenj o funkcionalnosti in obnašanju sistema med strankami oziroma končnimi uporabniki in tistimi, k iga načrtujejo. Na sliki 4.5 je prikazan diagram primerov uporabe sistema.[23]



Slika 4.5: Diagram primerov uporabe sistema.

Uporabnik

Uporabnik je osnovni akter sistema, iz katerega so izpeljanji vsi ostali akterji. Na voljo ima naslednje funkcionalnosti:

- prijavo v sistem - uporabnik, ki se nahaja v sistemu, mora za dostop poznati svoje uporabniško ime in geslo, ki ju je določil administrator ali superadministrator,
- odjavo iz sistema - uporabnik se lahko kadarkoli odjavi iz sistema,

- nadaljevanje seje - uporabnik lahko nadaljuje že obstoječo sejo, če se IP naslov ujema in seja ni potekla.

Poljubno ustvarjen uporabnik

Poljubno ustvarjen uporabnik je akter, ki ima na voljo vse funkcionalnosti uporabnika. Lahko izvaja tudi aktivnosti, do katerih je dobil pravice iz strani administratorja ali superadministratorja.

Administrator

Administrator ima pooblastila za vse operacije na dodeljenih strežnikih (ki mu jih je dodelil superadministrator). Poleg primerov uporabe navadnega uporabnika ima še naslednje možnosti:

- urejanje uporabnikov - administrator lahko ureja vse uporabnike na strežnikih, do katerih ima dodeljenje pravice (geslo, pravice) in dodaja nove,
- urejanje strežnikov - administrator lahko ureja vse dodeljene strežnike,
- izvajanje razširitev - administrator lahko izvaja ukaze na vseh vtičnikih na njemu dodeljenih strežnikih,
- urejanje razširitev - administrator lahko ureja funkcionalnosti vseh razširitev,
- upravljanje sej - administrator lahko upravlja z vsemi sejami na njemu dodeljenih strežnikih,
- brisanje razširitev - administrator lahko ustvari ali izbriše vse razširitve na vseh strežnikih, ki so mu dodeljeni,
- upravljanje sej - administrator lahko upravlja z vsemi sejami na njemu dodeljenih strežnikih.

Superadministrator

Superadministrator ima pooblastila za vse operacije sistema. Poleg primerov uporabe navadnega uporabnika in administratorja ima še naslednje možnosti:

- dodajanje uporabnikov - superadministrator lahko dodaja uporabnike na vseh strežnikih,
- brisanje uporabnikov - superadministrator lahko briše uporabnike na vseh strežnikih,
- urejanje uporabnikov - superadministrator lahko ureja uporabnike na vseh strežnikih,
- brisanje razširitev - superadministrator lahko ustavi ali izbriše razširitve na vseh strežnikih,
- dodajanje razširitev - superadministrator lahko doda razširitve na vseh strežnikih,
- dodajanje strežnikov - superadministrator lahko doda strežnike v sistem,
- brisanje strežnikov - superadministrator lahko odstrani strežnike iz sistema,
- urejanje strežnikov - superadministrator lahko ureja strežnike,
- izvajanje razširitev - superadministrator lahko izvaja ukaze na vseh razširitvah na vseh strežnikih,
- upravljanje sej - superadministrator lahko upravlja seje na vseh strežnikih.

4.3.3 Podatkovni model

V modelu smo definirali osnovne in podporne entitete. Podatkovni model na sliki 4.6 prikazuje entitetno relacijski diagram (v nadaljevanju ERD) podatkovne baze.

4.3.3.1 Osnovne entitete

ERD vsebuje naslednje entitete:

1. **Uporabniki** (`TANGAKWUNU_USERS`)
    - hrani seznam uporabnikov,
    - uporabniško ime,
    - elektronski poštni naslov,
    - serializiran JSON objekt uporabniških pravic.
2. **Strežniki** (`TANGAKWUNU_SERVERS`)
    - hrani IP naslov strežnika,
    - SHA512 enkodirano vrednost ukaza `dmidecode`.
3. **Razširitve** (`TANGAKWUNU_PLUGINS`)
    - hrani ime razširitve (JAR datoteka),
    - serializiran JSON objekt pravic razširitve,
    - unikatno vrednost nalagalca razredov, ki je naložil razširitev,
    - vrednost, če je razširitev dobila pravice izvajanja iz strani uporabnika.
4. **Zahtevki** (`TANGAKWUNU_REQUESTS`)
    - hrani čas zahtevka,
    - zahtevek,

- SHA512 enkodirano vrednost zahtevka.

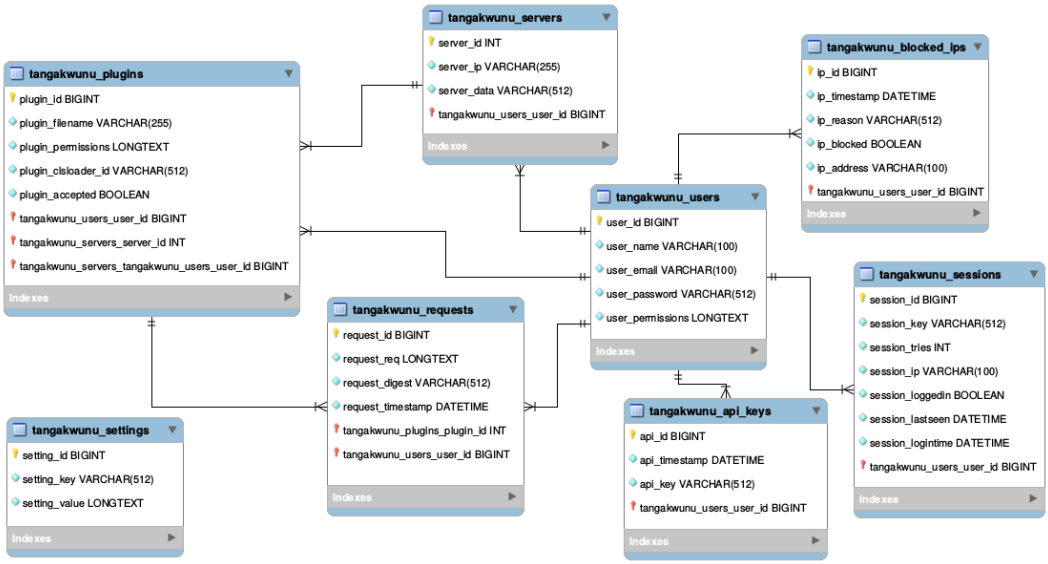
5. **API ključi** (`TANGAKWUNU_API_KEYS`)
   - hrani API ključe za dostop do aplikacije,
   - čas izdaje ključa,
   - vrednost ključa.
6. **Seje** (`TANGAKWUNU_SESSIONS`)
   - hrani seje uporabnikov,
   - čas prijave uporabnika,
   - zadnji čas zahtevka uporabnika,
   - vrednost, če se je uporabnik uspešno prijavil ali ne,
   - število poskusov prijave,
   - IP naslov klienta, ki se je poskušal prijaviti.
7. **Blokirani IP naslovi** (`TANGAKWUNU_BLOCKED_IPS`)
   - hrani blokirane IP naslove,
   - čas blokade naslova,
   - razlog za blokiranje naslova,
   - IP naslov,
   - vrednost, če je naslov blokiran ali deblokiran.

4.3.3.2 Podporne entitete

Poleg osnovnih entitet so za pravilno delovanje aplikacije potrebne še naslednje podporne entitete:

1. **Nastavitve** (`TANGAKWUNU_SETTINGS`)
   - hrani nastavitve,

- ključ (ime) nastavitve,
- serializiran JSON objekt, ki hrani nastavitev.



Slika 4.6: Podatkovni model Tangakwunu.

4.3.4 Prototipi uporabniških vmesnikov

Pred izdelavo same aplikacije smo se lotili izdelave papirnatih prototipov. V razvoju aplikacij nam papirnati prototipi prihranijo čas in denar in omogočajo razvijalcem, da testirajo uporabniške vmesnike preden se lotijo razvoja. Tak način razvoja omogoča enostavno in cenejše spreminjanje obstoječega izgleda.[20]

Na sliki 4.7 vidimo končni prototip glavne zaslonske maske. Strežniki so prikazani v stolpcih, spodaj so njihove informacije, do nove maske za nadzor strežnika pa pridemo z dotikom na izbrani strežnik.

Slika 4.7: Glavna zaslonska maska aplikacije.

Na sliki 4.8 vidimo končni prototip glavne zaslonske maske s stranskim menijem. Meni je drsni, njegova vsebina pa je odvisna od zaslonske maske, na kateri se trenutno nahajamo.

Slika 4.8: Glavna zaslonska maska s stranskim menijem.

Poglavje 5

Razvoj rešitve

Razvoj rešitve smo ločili v dve fazi. Prva faza je vključevala razvoj strežnika (Java), druga faza pa razvoj klienta (Android). V poglavju se bomo posvetili ključnim delom aplikacije, ki se osredotočajo na sistem za nalaganje razširitev in predstavitev podatkov na klientu.

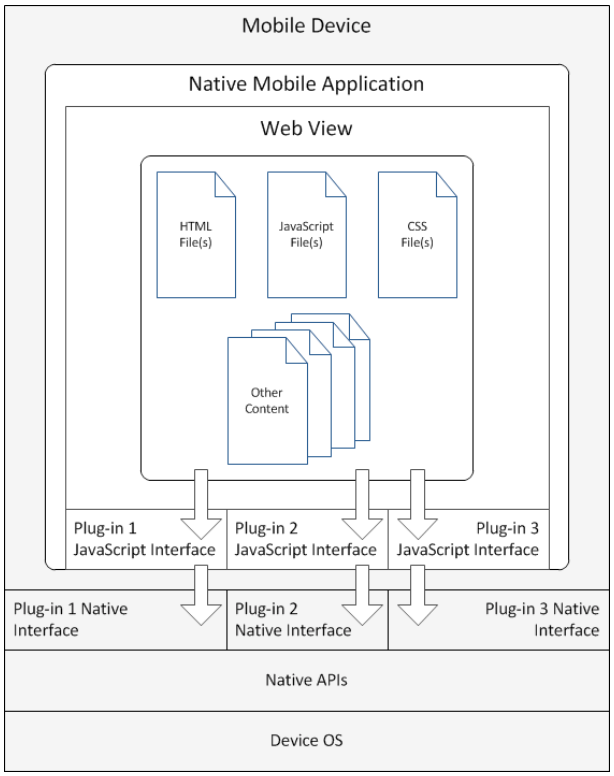
5.1 Uporabljene knjižnice

Pri razvoju rešitve smo uporabili veliko odprtokodnih knjižnic, vendar se bomo v delu posvetili le opisu najpomembnejših.

5.1.1 Apache Cordova

Apache Cordova je odprtokodno ogrodje za izdelavo prenosljivih hibridnih (kombinacija spletne kode in kode na ciljni platformi) aplikacij za mobilne platforme z uporabo HTML5. Prvotni cilj ustvarjalcev je bil ustvariti ogrodje, ki omogoča več dostopa do naprave kakor nativne spletne aplikacije. Cordova je za dostop do teh funkcionalnosti (GPS, seznam kontaktov, idp.) uvedla posebne APIje (arhitektura ogrodja je vidna na sliki 5.1).[25]

Trenutno je podprt razvoj aplikacij na Androidu, iOS, bada, BlackBerry, Firefox OS, Tizen, Windows Phone 7 in Ubuntu Touch, zaradi omejenega



Slika 5.1: Arhitektura ogrodja Cordova.

nabora osnovnih APIjev pa je omogočeno razvijanje razširitev, s katerimi lahko uporabljamo nativno kodo v poljubne namene.

V razvoju smo knjižnico uporabili za izdelavo klienta na Androidu. Za ta pristop razvoja smo se odločili zaradi lažje manipulacije z grafičnim uporabniškim vmesnikom, lažjim dodajanjem razširitev in lažjim prehodom na druge platforme v prihodnjih različicah.

5.1.2 Fries

Fries[1] je ogrodje za izdelavo Android uporabniških vmesnikov (avtor Jaune Sarmiento) s pomočjo tehnologij HTML, CSS in Javascript. Omogoča uporabo istih komponent kakor v nativnih Android aplikacijah, podprti pa so

tudi dialogi in notifikacije.

Ogrodje smo uporabili zaradi boljše prilagoditve izgleda hibridne aplikacije nativnim Android aplikacijam.

5.1.3 TouchSwipe

TouchSwipe[7] je razširitev za knjižico JQuery, ki omogoča uporabo gest s prsti na mobilnih spletnih straneh (avtorja Matta Brysona). Prvotno je bila razširitev razvita leta 2010 za podjetje Renault, uporabljala pa se je pri njihovi galeriji.

Iz nje lahko pridobimo podatke o smeri, razdalji in času geste, lahko pa jo uporabljamo tudi pri večprstnih gestah.

5.2 Razvoj strežnika

Zaradi kompleksnosti arhitekture strežnika bomo predstavili le najpomembnejše dele razvoja, ki so ključni za delovanje sistema razširitev.

5.2.1 Struktura zahtevkov in procesiranje

Strežnik za komunikacijo uporablja JSON datoteke. Struktura je prikazana v 5.1, vsebuje pa naslednje podatke:

- **server** - vsebuje identifikacijo strežnika s katerim upravljamo,
- **type** - če je tip plugin, se izvajajo akcije na razširitvah, če je basic, se izvajajo osnovne operacije na strežniku (prijava, odjava, statistika ipd.),
- **operation** - ime osnovne operacije, ki se izvaja na strežniku (definirana, če je definiran `type`),

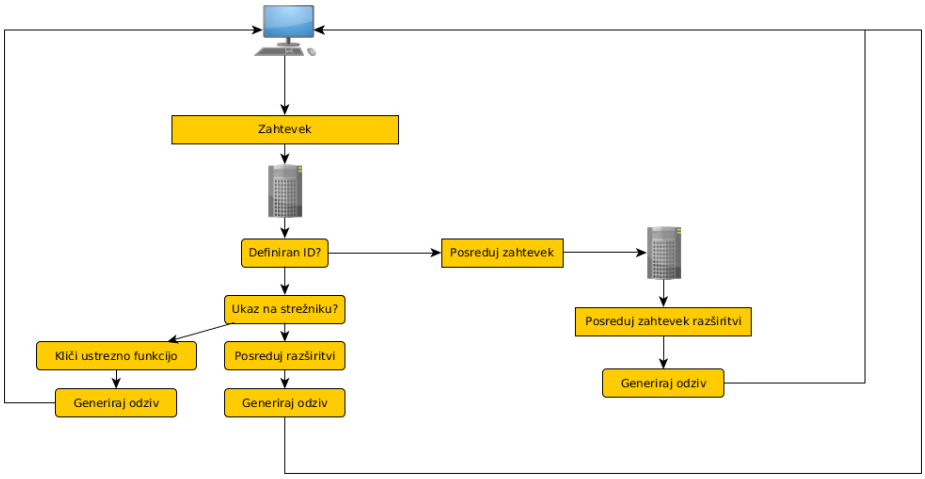
- **gui_mask_id** - vsebuje niz trenutne zaslonske maske iz katere prihaja zahtevek,
- **plugin_id** - vsebuje identifikacijsko številko razširitve,
- **data** - če je definiran, pošiljamo podatke na strežnik (stanje trenutne zaslonske maske),
- **session_id** - vsebuje številko seje,
- **response** - če je definiran, pomeni, da je klient dobil odziv in je v njem definirana zaslonska maska,
- **message** - sporočilo v zahtevku,
- **code** - identifikacijska koda zahtevka.

```
1  {
2    "server": "id␣of␣server",
3    "type": "basic",
4    "operation": "login",
5    "gui_mask_id": "login",
6    "plugin_id": "",
7    "data": [
8      {"user": "zerocool", "password": "geslo"}
9    ],
10   "session_id": "",
11   "response":  [
12     {"session_id": "random␣session␣id", "html_data": "
         zaslonska␣maska"}
13   ],
14   "message": "",
15   "code": ""
16 }
```

Koda 5.1: Struktura JSON zahtevka na strežnik.

Shema, ki prikazuje, kako strežnik reagira na zahtevek je vidna na sliki 5.2. Če je tip operacije osnoven, se kličejo osnovne funkcije strežnika, če je definiran `server` se zahtevek pošlje naprej ustreznemu strežniku, če ne, se izvede

na korenskemu.



Slika 5.2: Shema procesiranja zahtevka na strežniku.

5.2.2 Nalagalec razširitev

Nalagalec razširitev je jedro aplikacije, ki omogoča, da se razširitve naložijo dinamično med izvajanjem aplikacije. Za potrebe nalaganja smo ustvarili razreda `TangakwunuPluginLoader` (koda 5.2) in `TangakwunuPluginWatcher` (koda 5.3).

Razred `TangakwunuPluginLoader` razširi razred `URLClassLoader`. V konstruktorju prejmemo za argument pot do JAR datoteke in niz `loader_id`, ki je zgoščena vrednost datoteke, ki jo nalagamo in hkrati tudi API ključ (razloženo v poglavju 5.2.4).

```
1 public class TangakwunuPluginLoader extends URLClassLoader {
2
3   private String loader_id;
4
```

```
5   public TangakwunuPluginLoader(URL jarFileUrl, String
        loader_id) {
6     super(new URL[] { jarFileUrl });
7     this.loader_id = loader_id;
8   }
9
10  public String getLoaderID() {
11    return this.loader_id;
12  }
13
14 }
```

Koda 5.2: Razred za nalaganje razširitve.

Razred `TangakwunuPluginWatcher` je odgovoren za redno pregledovanje direktorija *plugins*. Če se v njem pojavi nova datoteka JAR, se izbriše ali spremeni, se proži dogodek, pri katerem dodamo (ali izbrišemo) razširitev v globalni spremenljivki `plugins`.

```
1  public class TangakwunuPluginWatcher extends Thread {
2
3    private String dir;
4    private HashMap<String, Object> plugins;
5
6    public TangakwunuPluginWatcher(String dir, HashMap<String,
         Object> plugins) {
7      this.dir = dir;
8      this.plugins = plugins;
9    }
10
11   public void run() {
12
13     // ... inicializiramo WatcherService v try catch bloku
14
15     watcher = FileSystems.getDefault().newWatchService();
16     File plugin_dir_f = new File(this.dir);
17     Path plugin_path = Paths.get(plugin_dir_f.toURI());
18
19     // try catch blok za registracijo poti watcherja
20
21     plugin_path.register(watcher, ENTRY_CREATE, ENTRY_DELETE,
           ENTRY_MODIFY);
22
```

```
23      while(true) {
24        WatchKey key;
25        try {
26          key = watcher.take();
27        } catch(InterruptedException ex) {
28          return;
29        }
30
31        for(WatchEvent<?> event : key.pollEvents()) {
32          WatchEvent.Kind<?> kind = event.kind();
33
34          WatchEvent<Path> ev = (WatchEvent<Path>) event;
35          Path fileName = ev.context();
36
37          if(kind.name() == "ENTRY_CREATE" || kind.name() == "
               ENTRY_MODIFY") {
38            // preveri checksum datoteke in nalozi razsiritev
39            // v primeru iste razsiritve ukini obstojeci
                 nalagalec razreda za
40            // to razsiritev
41          }
42          else if(kind.name() == "ENTRY_DELETE") {
43            // pobrisi razsiritev in ukini nalagalec razreda te
                  razsiritve
44          }
45
46        }
47
48        boolean valid = key.reset();
49        if(!valid)  {
50          break;
51        }
52      }
53    }
54  }
```

Koda 5.3: Razred za nalaganje razširitve.

Zaradi lažjega odstranjevanja in dodajanja razširitev vsaki razširitvi dodelimo svojo instanco nalagalca razredov `TangakwunuPluginLoader`. Če med izvajanjem razširitve pride do klica za izbris, nadgradnjo ali ponovno namestitev, nad instanco `TangakwunuPluginLoader` pokličemo metodo `close`, kar povzroči, da se vsi razredi v tej razširitvi dereferencirajo.

Pred inicializacijo glavnega razreda razširitve preverimo, če razred implementira vmesnik `TangakwunuPluginInterface`. To preverimo s ključno besedo `instanceof`. Če razred implementira ta vmesnik, spremenljivki `api_key` določimo vrednost s pomočjo refleksije:

```
1  // ...
2  // izracunamo API kljuc in ga dodelimo spremenljivki
       api_key
3  // ...
4
5  Class<?> cls = plugin_loader.loadClass("com.tangakwunu.
       plugins.TangakwunuPlugin");
6  Object plugin_instance = cls.newInstance();
7
8  if(plugin_instance instanceof TangakwunuPluginInterface) {
9    Field api_field = plugin_instance.getClass().
         getDeclaredField("api_key");
10   api_field.set(plugin_instance, api_key);
11
12   // ...
13   // API kljuc in podatke o razsiritvi shranimo v
         podatkovno bazo
14   // ...
15
16 }
17 else {
18   plugin_instance.close();
19 }
```

Koda 5.4: Določanje API ključa razširitvi s pomočjo refleksije.

Z metodo `newinstance` ustvarimo novo instanco razreda, z metodo `getDeclaredField` pa dobimo spremenljivko, ki ji bomo določili vrednost. Vrednost API ključa nastavimo z metodo `set`.

5.2.3 Omejevanje pravic razširitvam

Za potrebe omejevanja pravic razširitvam smo implementirali razred `TangakwunuSecurityPolicy`, ki razširja javanski razred `Policy`.[22]

Razred `Policy` je odgovoren za ugotavljanje ali ima koda, ki jo bomo izvajali pravice do varnostno občutljivih operacij. V danem JRE lahko hkrati obstaja samo en razred `Policy`.

```
1  public class TangakwunuSecurityPolicy extends Policy {
2
3    public PermissionCollection getPermissions(ProtectionDomain
         domain) {
4      if(isPlugin(domain)) {
5        return pluginPermissions();
6      }
7      else {
8        return applicationPermissions();
9      }
10   }
11
12   private boolean isPlugin(ProtectionDomain domain) {
13     return domain.getClassLoader() instanceof
      TangakwunuPluginLoader;
14   }
15
16   private PermissionCollection pluginPermissions() {
17     Permissions permissions = new Permissions();
18     return permissions;
19   }
20
21   private PermissionCollection applicationPermissions() {
22     Permissions permissions = new Permissions();
23     permissions.add(new AllPermission());
24     return permissions;
25   }
26
27 }
```

Koda 5.5: Razred za omejevanje pravic razširitvam.

V kodi 5.5 nastavimo dva seta pravic, eden pripada razširitvam (nobenih pravic), drug pa glavni aplikaciji (vse pravice). Če je naložen razred razširitev, preverjamo z metodo `isPlugin`, kjer preverimo, če je bil naložen razred z nalagalcem `TangakwunuPluginLoader` (vrstica 13).

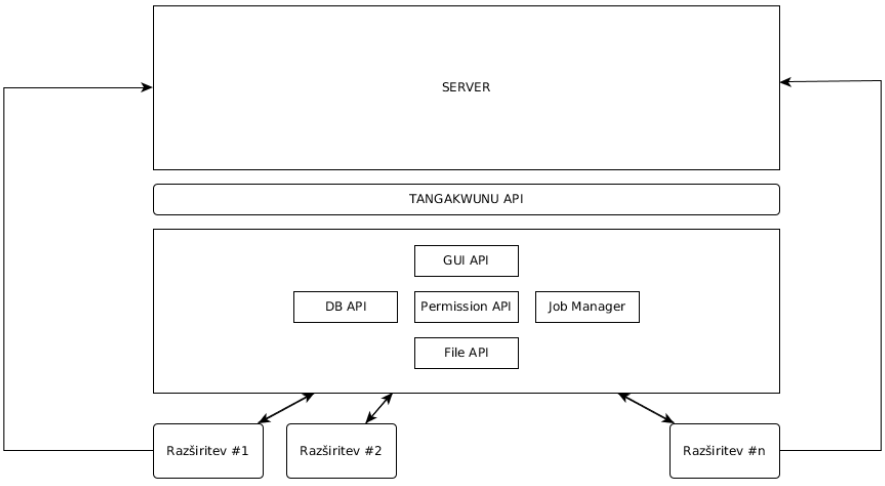
```
1 Policy.setPolicy(new TangakwunuSecurityPolicy());
2 System.setSecurityManager(new SecurityManager());
```

Koda 5.6: Nastavljanje varnostne politike.

Pred začetkom glavne zanke strežnika definiramo varnostno politiko s kodo 5.6, kjer povozimo privzeto varnostno politiko z našo. Če želi razširitev dostopati do varnostno občutljivih operacij, mora za to zahtevati pravice in uporabljati API (razloženo v 5.2.4).

5.2.4 Razvoj API za razširitve

Za potrebo manipulacij z datotekami in drugimi pravicami smo razvili API (shema delovanja je vidna na sliki 5.3), ki razširitvam omogoča pisanje in branje datotek, uporabo podatkovne baze in ustvarjanje novih, ter generiranje in pošiljanje zahtevkov na strežnike.



Slika 5.3: Shema delovanja API naše aplikacije.

Vsaka razširitev mora biti v paketu `com.tangakwunu.plugins`, ime glavnega razreda mora biti `TangakwunuPlugin`, razred pa mora implementirati vme-

snik `TangakwunuPluginInterface` (koda 5.7).

```
1  public interface TangakwunuPluginInterface {
2    public String pluginVersion();
3    public String pluginAuthor();
4    public String pluginName();
5    public String pluginCompany();
6    public ArrayList<String> install_plugin();
7    public ArrayList<String> uninstall_plugin();
8    public TangakwunuRequestResponse processPluginRequest(
       JSONObject request_data, ArrayList<TangakwunuJobResponse
       > job_response);
9    public ArrayList<TangakwunuPermission> pluginPermissions();
10   public ArrayList<TangakwunuHookOverload> hookOverload();
11   public HashMap<String, TangakwunuHTMLFieldSet>
       gui_window_new();
12 }
```

Koda 5.7: Vmesnik za razvoj razširitev.

Metode vmesnika so naslednje:

- `pluginVersion` - metoda vrača verzijo razširitve,
- `pluginAuthor` - metoda vrača ime avtorja razširitve,
- `pluginName` - metoda vrača ime razširitve,
- `pluginCompany` - metoda vrača ime podjetja, ki je ustvarilo razširitev,
- `install_plugin` - metoda vrača imena paketov, ki so potrebna za uspešno namestitev razširitve (paketi se namestijo z ustreznim orodjem; `apt-get`, `yum` in podobni),
- `uninstall_plugin` - metoda vrača imena paketov, ki jih moramo odstraniti ob brisanju razširitve,
- `processPluginRequest` - metoda sprejme JSON objekt, ki vsebuje vse podatke o zahtevku in tabelo objektov `TangakwunuJobResponse`, ki je `null` ob prvotnemu zahtevku in definirana, kadar procesiramo zahtevek,

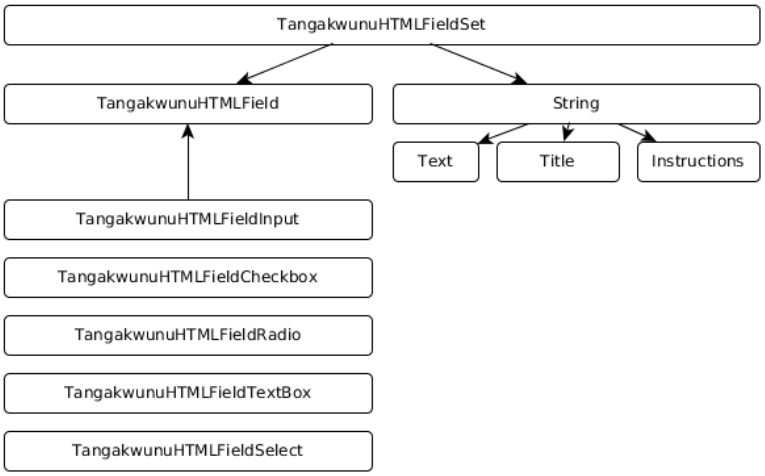
- `pluginPermissions` - metoda vrača polje objektov `TangakwunuPermission`, dovoljenja zahtevajo potrditev administratorja ob prvi inicializaciji razširitve,
- `hookOverload` - metoda vrača polje objektov `TangakwunuHookOverload`, ki vsebujejo imena metod, katerim razširitev dodaja funkcionalnost,
- `gui_window_new` - vrača zgoščeno tabelo objektov, prvi parameter pove ime grafičnega uporabniškega vmesnika, drugi pa objekt `TangakwunuHTMLFieldSet`, ki vsebuje HTML polja, ki se prikažejo na grafičnem uporabniškem vmesniku.

Podrobneje se bomo posvetili opisu razreda `TangakwunuHTMLFieldSet`, `TangakwunuPermission` in `TangakwunuHookOverload`.

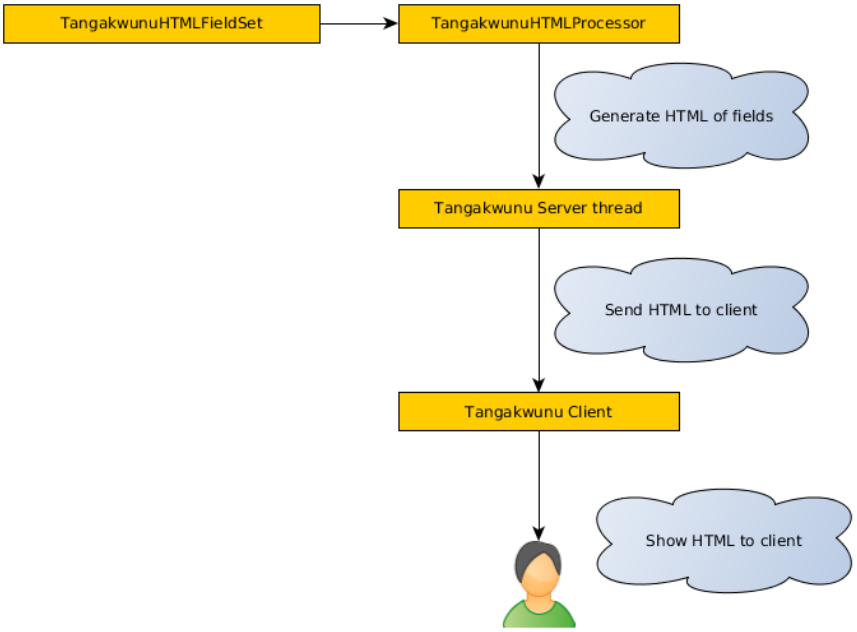
5.2.4.1 Razred `TangakwunuHTMLFieldSet`

`TangakwunuHTMLFieldSet` je lista objektov, njegova shema je prikazana na sliki 5.4. V konstruktor prejme tri nize (tekst, ime in navodila; ta so prikazana na začetku zaslonske maske) in polje objektov `TangakwunuHTMLField`.

Razred `TangakwunuHTMLField` je nadrazred razredov `TangakwunuHTMLFieldInput`, `TangakwunuHTMLFieldCheckbox`, `TangakwunuHTMLFieldRadio`, `TangakwunuHTMLFieldTextBox` in `TangakwunuHTMLFieldSelect`. Vsi so odgovorni za objektno reprezentacijo HTML elementov, ki bodo prisotni na zaslonski maski na klientu. Vse te objekte pred pošiljanjem na klienta renderira strežniški razred `TangakwunuHTMLProcessor`, ki objekte pretvori v ustrezno HTML obliko in jih vrne klientu kot del JSON odziva (prikazano na sliki 5.5).



Slika 5.4: Shema razreda TangakwunuHTMLFieldSet.



Slika 5.5: Shema procesa renderiranja razreda `TangakwunuHTMLProcessor`.

5.2.4.2 Razred `TangakwunuPermission`

Razred `TangakwunuPermission` hrani pravice, ki jih določena razširitev zahteva. Ob prvi namestitvi razširitve se pošlje zahtevek na klienta, ki zahteva odgovor, ali končni uporabnik dovoli takšne nastavitve. Ob potrditvi se zastavica pravic v podatkovni bazi nastavi na `true`. Razredi `TangakwunuReadPermission`, `TangakwunuWritePermission` in `TangakwunuDBPermission` vsi razširjajo osnovni razred `TangakwunuPermission`. Prvi je namenjen branju datotek (kot argument prejme pot na sistemu do katere želi pravice), drugi pisanju (kot argument prejme pot na sistemu do katere želi pravice) v datoteke, tretji pa manipulaciji s podatkovno bazo (kot argument prejme razširitve razreda, katere omogočajo različne pravice kot so ustvarjanje novih podatkovnih baz in tabel).

Vse pravice se ob prvi namestitvi in nalaganju razširitve shranijo v globalno spremenljivko nalagalcev razredov. Ob klicu ustrezne operacije strežnik preveri ali ima razširitev pravico do te operacije.

5.2.4.3 Razred `TangakwunuHookOverload`

Če želimo z razširitvijo nadgraditi osnovne funkcionalnosti, uporabimo razred `TangakwunuHookOverload`. Razred kot argument prejme ime funkcionalnosti, ki jo razširjamo, in ime metode, ki jo bo naša razširitev implementirala. Ob nalaganju razširitve strežnik preveri, če metode, ki jih razširitev implementira vračajo ustrezne vrednosti.

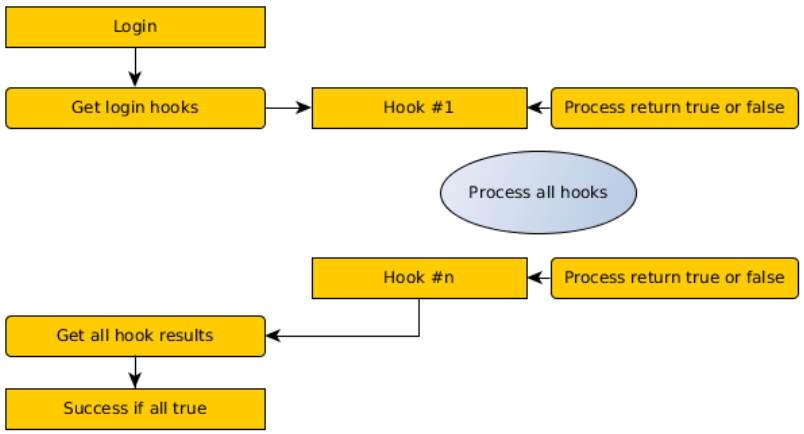
Če metoda vrača pravilno vrednost preverjamo s pomočjo refleksije. Ob inicializaciji objekta naredimo spremenljivko `m`, ki pobere ustrezno metodo, nakar na njej kličemo `m.getReturnType()`. Vrednost te spremenljivke primerjamo z ustreznim tipom. Če je tip neustrezen, nalagalca razredov, ki je razširitev naložil, zapremo in o tem obvestimo uporabnika.

Potek si bomo ogledali na primeru prijave. Ob prijavi uporabnik poda upo-

Algoritem 1: Preverjanje pravilnosti vračanja tipov metod.

```
Data: Class loader of plugin
Result: True or false if methods are implemented correctly or
        incorrectly
initialization;
forall the elements of TangakwunuHookOverload array do
    get method name;
    check if method returns correct type;
    if method does not return correct type then
        return false;
    end
end
return true;
```

rabniško ime in geslo. Če v naši razširitvi kot ime funkcije uporabimo `login` in ime metode `login_overload`, je potrebno v razširitvi implementirati funkcijo z imenom `login_overload`, ki vrača tip `boolean`. Potek izvedbe prijave je prikazan na diagramu 5.6.

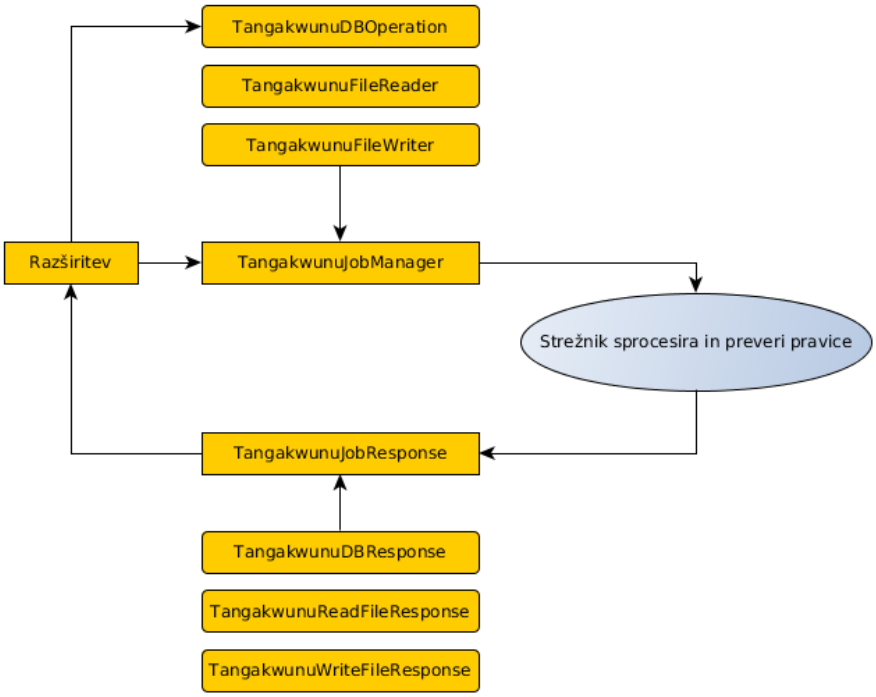


Slika 5.6: Shema procesa prijave z dodanimi funkcionalnostmi razširitev.

Na podoben način delujejo vse ostale funkcije, ki jim lahko razširjamo osnovno funkcionalnost.

5.2.4.4 Izvajanje varnostno občutljivih operacij

Za izvajanje operacij ima API na voljo razred `TangakwunuJobManager`. Ta kot argument prejme seznam objektov tipa `TangakwunuJob`. Razredi `TangakwunuFileReader`, `TangakwunuFileWriter` in `TangakwunuDBOperation` so ustrezne razširitve tega razreda. Prva kot argument prejme pot do datoteke in vrača vsebino datoteke, druga pot do datoteke in vsebino, ki jo je potrebno zapisati, zadnja pa ustrezne operacije, ki jih razširitev potrebuje za potrebe manipulacij podatkov v bazi. Shema delovanja je prikazana na sliki 5.7.



Slika 5.7: Shema procesa izvajanja varnostno občutljivih operacij.

Operacije se izvedejo ob vračanju podatkov strežniku iz razširitve. Če klic metode `processPluginRequest` vrne v razred `TangakwunuRequestResponse` objekt operacij, se te operacije izvedejo in odzivi ponovno pošljejo nazaj razširitvi.

5.2.5 Razvoj razširitev nginx in vsftpd

Ker sta aplikaciji vsftpd in nginx po arhitekturi zelo kompleksni, se bomo dotaknili le strukture njunih konfiguracijskih datotek.

Program vsftpd vsebuje eno konfiguracijsko datoteko (koda 5.8). Ker želimo dodati vsakemu uporabniku svoje pravice, smo v konfiguracijsko datoteko zapisali pot do direktorija, kjer bomo hranili uporabnike (ukaz `user_config_dir`).

```
1  # Example config file /etc/vsftpd.conf
2  write_enable=YES
3  dirmessage_enable=YES
4  ftpd_banner="Welcome␣to␣thepatch␣FTP␣service."
5  local_enable=YES
6  local_umask=022
7  chroot_local_user=YES
8  anonymous_enable=YES
9  anon_upload_enable=YES
10 anon_umask=022
11 anon_mkdir_write_enable=YES
12 anon_other_write_enable=YES
13 syslog_enable=YES
14 connect_from_port_20=YES
15 ascii_upload_enable=YES
16 ascii_download_enable=YES
17 pam_service_name=vsftpd
18 listen=no
```

Koda 5.8: Konfiguracijska datoteka vsftpd.

Za vsakega uporabnika nato generiramo njegovo konfiguracijsko datoteko na podlagi tega, kar smo dobili iz klienta. Primer konfiguracijske datoteke je viden na kodi 5.9.

```
1  # Primer uporabnika, ki se nahaja v direktoriju /home/ftp/
       uporabnik in nima pravice do zapisovanja
2  local_root=/home/ftp/uporabnik
3  write_enable=NO
```

Koda 5.9: Konfiguracijska datoteka uporabnika vsftpd.

Vse ostale konfiguracijske možnosti, ki smo jih pri izdelavi razširitve uporabili, so navedene v dodatku A.

Struktura nginx datotek je podobna strukturi razredov v Javi. Najprej je navedeno ime, nato opcije, ki so v zavitih oklepajih. Vse datoteke, ki urejajo različne konfiguracije domen in poddomen se nahajajo v direktoriju `/etc/nginx/sites-available`. V kodi 5.10 vidimo osnovno konfiguracijsko datoteko orodja nginx.

```
1  server {
2    server_name default;
3    listen 80 default_server;
4    root /usr/share/nginx/www;
5
6    location ~ /\.ht {
7      deny all;
8    }
9
10   location ~ \.php$ {
11     try_files $uri =404;
12     fastcgi_split_path_info ^(.+\.php)(/.+)$;
13     include fastcgi_params;
14     fastcgi_pass unix:/var/run/php5-fpm.sock;
15     fastcgi_index index.php;
16     fastcgi_param SCRIPT_FILENAME
           $document_root$fastcgi_script_name;
17   }
18 }
```

Koda 5.10: Osnovna konfiguracijska datoteka orodja nginx.

V osnovnem bloku `server` so prisotne naslednje spremenljivke:

- root - pot kamor kaže domena ali poddomena,
- listen - na kateri številki vrat poslušamo zahtevke,

- server_name - ime strežnika.

Da ustvarimo novo domeno, je potrebno v direktoriju `/etc/nginx/sites-available` ustvariti novo datoteko, ki vsebuje isto vsebino kakor zgornja z ustreznimi parametri. Ko se datoteka ustvari, razširitev proži ponoven zagon nginx, da se spremembe uveljavijo.

5.3 Razvoj klienta

Ker gre pri razvoju klienta predvsem za manipulacijo HTML elementov s pomočjo knjižnice JQuery in gradnjo JSON zahtevkov, bomo razložili le razvoj razširitve za Cordovo, ki nam omogoča varno komuniciranje s strežnikom.

Razširitve za Cordovo so sestavljene iz dela, ki teče v WebView (javascript) in dela, ki teče na platformi (v našem primeru Java).

Za izvajanje razširitve v javascriptu vedno uporabljamo metodo `exec`, ki jo moramo ustrezno definirati (prikazano v kodi 5.11).

```
1  function SocketClient() {
2  }
3
4  SocketClient.prototype.sendSocket= function (address, port,
       message, successCallback, errorCallback) {
5    cordova.exec(successCallback, errorCallback, "SocketClient"
         , "sendSocket", [address, port, message]);
6  };
7
8  SocketClient.install = function () {
9    if (!window.plugins) {
10     window.plugins = {};
11   }
12
13   window.plugins.socketClient = new SocketClient();
14   return window.plugins.socketClient;
15 };
16
```

```
17 cordova.addConstructor(SocketClient.install);
18
19 // posiljanje zahtevka na streznik
20 window.plugins.socketClient.sendSocket(
21                 "target-domain.com",
22                 "port",
23                 "data",
24                 function(e) {
25                   /* uspesno izveden zahtevek, podatki
26                   so v spremenljivki e */
27                 },
28                 function(e) {
29                   /* neuspesen zahtevek */
30                 }
31 );
```

Koda 5.11: Implementacija Javascript dela razširitve in primer klica.

Vsaka razširitev razširja razred `CordovaPlugin`. Kar kličemo v javascript metodi `exec`, se pošlje v glavni razred razširitve v metodo `exec`. Pred izvajanjem je potrebno preveriti, če je ukaz pravilen (celoten postopek je viden na kodi 5.12).

```
1  package com.tangakwunu.cordova.plugins;
2
3  public class SocketClient extends CordovaPlugin {
4
5    private static final String ACTION_SEND_SOCKET = "
       sendSocket";
6
7    public boolean sendSocket(String dstAddress, int dstPort,
       String message, CallbackContext callbackContext) {
8
9      String response = "";
10     Socket socket = null;
11     SSLSocketFactory sf = (SSLSocketFactory) SSLSocketFactory
         .getDefault();
12
13     try {
14       socket = sf.createSocket(dstAddress, dstPort);
15       // Send message and read server response code ...
16       // Write response to string response ...
17     }
```

```
18      catch (Exception e) {
19        // catch exceptions
20      }
21      finally{
22        if(socket != null) {
23          socket.close();
24        }
25      }
26
27      callbackContext.success(response);
28      return false;
29    }
30
31    public boolean execute(String action, JSONArray args, final
          CallbackContext callbackContext) throws JSONException {
32      if(ACTION_SEND_SOCKET.equals(action)){
33        final String address = args.getString(0);
34        final String port= args.getString(1);
35        final String message= args.getString(2);
36        new Thread(new Runnable() {
37          public void run() {
38            int int_port =  Integer.parseInt(port);
39            sendSocket(address,int_port, message,
                  callbackContext);
40          }
41        }).start();
42
43        return true;
44
45        } else {
46          callbackContext.error("Not a supported function.");
47        }
48    }
49  }
```

Koda 5.12: Cordova razred za pošiljanje zahtevkov na strežnik.

Poglavje 6

Analiza aplikacije

6.1 SWOT analiza

SWOT analiza (oziroma SWOT matrika) je strukturirana metoda načrtovanja za vrednotenje prednosti, slabosti, priložnosti in nevarnosti v projektu. Lahko se izvaja za izdelek, kraj, industrijo ali osebo. Vključuje specificiranje ciljev tega produkta in faktorjev, ki bi lahko bili odgovorni za uspešno ali neuspešno doseganje njih. Analizo izvedemo tako, da preverimo štiri kategorije značilnosti:[26]

- prednosti (pred konkurenčnimi produkti),
- slabosti (kaj konkurenčni produkti delajo boljše od naše aplikacije),
- priložnosti (možnosti za pridobivanje prednosti pred konkurenčnimi produkti),
- nevarnosti (katere značilnosti predstavljajo možnost za izgubo prednosti pred konkurenčnimi produkti).

	PRODUKT				
KRITERIJ	Tangakwunu	cPanel	ajenti	Webmin	Plesk
Možnost razširitev	✓	✓ [1]	✓ [2]	✓ [3]	✓ [4]
Mobilni vmesnik	✓	✓ (omejene funkcionalnosti)	×	×	✓ (omejene funkcionalnosti)
Cena	brezplačno (strežnik), plačljive ali neplačljive razširitve	$20 mesečno, $200 letno za VPS	brezplačno	brezplačno	$11 mesečno za VPS
Enostavnost uporabe	tako za uporabnike brez predznanja kot za sistemske administratorje, lahka namestitev in odstranjevanje	odstranitev brez formatiranja ni mogoča, vmesnik za sistemske administratorje	vmesnik za sistemske administratorje, lahka namestitev	težja namestitev, lahko odstranjevanje, vmesnik za sistemske administratorje	težja namestitev, vmesnik za sistemske administratorje
Modulna neodvisnost	✓	×	×	×	×

Tabela 6.1: Primerjava naše rešitve z ostalimi na trgu.

[1] Ima omejen nabor in ni centralnega repozitorija.
[2] Odvisna je od razširitev na Webminu.
[3] Potrebno je napisati celotno funkcionalnost.
[4] Ima omejen nabor in ni centralnega repozitorija.

V tabeli 6.1 vidimo primerjavo naše rešitve z rešitvami, ki so trenutno aktualne na trgu.

6.1.1 Prednosti

Trenutno je največja prednost naše rešitve zasnovana arhitektura. Naš pristop omogoča ločevanje grafičnega uporabniškega vmesnika, zasnovalci razširitev pa lahko napišejo klienta v kateremkoli jeziku, ki podpira odpiranje vtičev. Prednost je tudi podpora večim strežnikov in dograjevanje funkcij same aplikacije s pomočjo dodajanja klicev funkcijam.

6.1.2 Slabosti

Največja slabost našega sistema je omejen nabor razširitev in omejen API. Pričakujemo, da se bo skozi razvoj dodatnih razširitev dodajalo tudi funkcionalnosti APIju, z njimi pa bo napredovala tudi konkurenčnost naše aplikacije.

6.1.3 Priložnosti

Zaradi zasnovane arhitekture vidimo priložnost v širitvi funkcionalnosti aplikacije. Podjetja, ki omogočajo VPSje strankam, bi lahko naredila lastne vmesnike in lastne aplikacije za nadzor razširitev na strežniku.

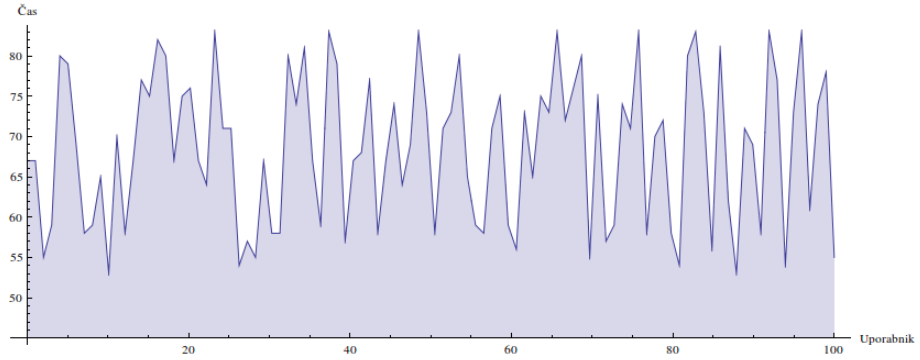
6.1.4 Nevarnosti

Arhitektura aplikacije je bila zasnovana na varnosti, vendar še nismo opravili nobenih penetracijskih testov. V nadaljnjem razvoju in uporabi pričakujemo, da bomo naredili boljše peskovniško okolje za izvajanje razširitev na strežniku in izvedli penetracijske teste ter dodali podporo za uporabo certifikatov na uporabnikih.

6.2 Obremenitveni test strežnika

Za potrebe testiranja hitrosti strežnika smo razvili klienta v Javi, ki pošilja naključne zahtevke na naključne razširitve na strežniku[5]. Ker je izvajanje in preusmeritev zahtevkov na druge strežnike odvisna od hitrosti linije in mreže, smo testiranje izvajali samo na enemu korenskemu strežniku, ki je hkrati posrednik in izvajalec ukazov. Tako smo dobili bolj relevantne rezultate.

Na sliki 6.1 vidimo rezultate testiranja zahtevkov (povprečen čas izvajanja zahtevka vsakega uporabnika), kjer je hkrati v sistem prijavljenih sto uporabnikov (za potrebe testiranja smo avtentifikacijo odstranili), ti pa pošiljajo naključne zahtevke na enega izmed dveh vtičnikov.



Slika 6.1: Test hitrosti strežnika.

Najboljši uporabnik je dosegel 53 milisekund, najslabši pa 83 milisekund. Povprečen čas izvajanja zahtev med stotimi uporabniki je bil 68.55 milisekund. Kljub visokim obremenitvam je naš sistem ostal stabilen.

[5]Strežnik je bil postavljen pri podjetju DigitalOcean, 2GB spomina, dvojedrni procesor, 40GB SSD disk in gigabitna linija.

6.3 Možne izboljšave

Zaradi narave magistrskega naloge in časovne omejitve smo veliko izboljšav naše aplikacije prihranili za prihodnje verzije:

- **omejevanje resursov razširitev** - pri omejevanju resursov razširitvam bi radi dodali časovno omejitev izvajanja vsakega procesa,
- **repozitorij razširitev** - za nameščanje razširitev in lažje iskanje bi radi ustvarili repozitorij razširitev, ki bi omogočal podobni sistem namestitve aplikacij kot Google Play,
- **podpisovanje razširitev** - razvijalcem, ki jim zaupamo bi radi dodali možnost podpisovanja razširitev s certifikati,
- **razširitev funkcionalnosti API** - trenutno je omogočeno branje, pisanje in dostop do podatkovne baze, radi pa bi naredili tudi sistem, ki bi omogočal varno izvajanje sistemskih ukazov (API se bo razširjal tudi po potrebah razširitev),
- **dodatne možnosti razširjanja osnovnih funkcionalnosti strežnika** - trenutno orodje ponuja le razširjanje prijave in odjave, radi pa bi dodali podporo tudi za ostale osnovne funkcionalnosti (statistika, beleženje ipd.),
- **razvoj dodatnih razširitev** - da bi sistem lahko zaživel v praksi, je potreben razvoj dodatnih razširitev (Apache, ddclient ipd.),
- **prenos klienta** - prenos klienta na druge platforme (Windows Phone, iPhone ipd.),
- **testiranje razširitev** - radi bi dodali mehaniko, ki bi omogočala testiranje razširitev (hitrost izvajanja zahtevkov ipd.).

Poglavje 7

Zaključek

V magistrski nalogi smo razvili funkcionalen prototip aplikacije tipa strežnik-odjemalec, ki se uporablja za nadzor programske opreme na strežniku. Prototip je zasnovan na sistemu, ki izboljša trenutno podobno programsko opremo na trgu na področju razširljivosti, odprtosti platforme in enostavnosti uporabe. Z analizo aplikacije smo dokazali tudi stabilnost in hitrost naše arhitekture.

Z razvito arhitekturo bodočim razvijalcem in podjetjem omogočamo poljubno razširjanje in modifikacijo delov aplikacije, medtem ko s platformno neodvisnostjo podpiramo bodoč razvoj klientov na drugih programskih jezikih in platformah.

Aplikacija je trenutno v fazi testiranja in bo kmalu na voljo vsem uporabnikom. Trenutno se dela na novi verziji strežnika, ki bo ponujala več funkcionalnosti. Pri samem razvoju nismo imeli večjih težav, kar je rezultat dobrega načrtovanja in učinkovitega raziskovalnega dela.

Dodatek A

Seznam najpogostejših konfiguracijskih opcij vsftpd

A.1 Opcije tipa boolean

anonymous_enable
Če je ukaz nastavljen na yes, so anonimne prijave na strežnik dovoljene.

anon_upload_enable
Za delovanje te nastavitve mora biti na yes nastavljena tudi opcija *write_enable*.
Opcija označuje ali imajo anonimni uporabniki pravico do nalaganja datotek.

write_enable
Določa ali so dovoljeni ukazi za pisanje na strežniku (STOR, DELE, RNFR, RNTO, MKD, RMD, APPE, SITE).

delete_failed_uploads
Označuje, če se neuspešno naložene datoteke brišejo iz strežnika.

dirlist_enable
Določa, če se prikaže seznam datotek ob prijavi na strežnik.

local_enable
Določa ali imajo lokalni uporabniki pravico do prijave na strežnik (uporabniki shranjeni v /etc/passwd).

port_enable
Označuje, če je dovoljen ukaz PORT.

A.2 Numerične opcije

max_clients
Določa maksimalno število hkrati prijavljenih uporabnikov.

local_max_rate
Maksimalno število prenosa podatkov v bajtih na sekundo.

max_login_failsrate
Maksimalno število neveljavnih prijav.

accept_timeout
Določa, koliko sekund naj največ čaka strežnik, preden odjemalec pošlje zahtevo za pasivno povezavo.

data_connection_timeout
Določa, koliko sekund naj strežnik čaka pri neuspelih prenosih ali prenosih, ki so se ustavili.

anon_max_rate
Določa maksimalno število prenosa podatkov v bajtih na sekundo za anonimne uporabnike.

delay_successful_login
Določa zamik v sekundah pred dovoljeno uspešno prijavo.

delay_failed_login
Določa zamik v sekundah pred javljanjem neuspešne prijave.

A.3 Opcije z nizi

ftp_username
Določa ime anonimnega FTP uporabnika.

anon_root
Označuje pot do direktorija anonimnih uporabnikov.

cmds_allowed
Določa seznam dovoljenih ukazov na FTP strežniku.
Primer: cmds_allowed=PASV,RETR,QUIT

cmds_denied
Določa seznam prepovedanih ukazov na FTP strežniku (sintaksa je ista kot pri cmds_allowed).

dsa_cert_file
Označuje pot do datoteke, kjer se nahaja DSA certifikat za SSL povezave.

dsa_private_key_file
Pot do privatnega ključa DSA za SSL povezavo. Če opcija ni podana, strežnik predvideva, da se privatni ključ nahaja v certifikatu.

ftpd_banner
Prikazno sporočilo strežnika ob prijavi (če opcija ni navedena, se prikaže privzeto vsftpd sporočilo).

deny_file
Označuje seznam prepovedanih datotek (omejevanje nalaganja).
Primer: deny_file={*.mp3,*.jpg}

hide_file
Označuje, katere datoteke naj se skrijejo na strežniku. Sintaksa je identična deny_file.

rsa_cert_file
Označuje pot do datoteke, kjer se nahaja RSA certifikat za SSL povezave.

rsa_private_key_file
Pot do privatnega ključa RSA za SSL povezavo. Če opcija ni podana, strežnik predvideva, da se privatni ključ nahaja v certifikatu.

user_config_dir
Določa direktorij, kjer so shranjeni virtualni uporabniki in njihova pravila.

user_sub_token
Določa oznako za virtualnega uporabnika. Če npr. kot oznako določimo $USER, lahko v ukazu user_config_dir določimo vrednost /etc/vsftpd/$USER.

vsftpd_log_file
Označuje pot do zapisnika strežnika in ime datoteke.

Slike

Tabele

Literatura

[1] Fries UI development framework. Dostopno na: http://getfri.es/about.html. 15.09.2014 15:42.

[2] Java is not type safe. Dostopno na: http://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html. 10.09.2014 15:40.

[3] The Java Class File Disassembler. Dostopno na: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html. 06.09.2014 14:51.

[4] cPanel development kit. Dostopno na: https://documentation.cpanel.net/display/SDK/Software+Development+Kit+Home, . 08.09.2014 15:08.

[5] cPanel installation guide. Dostopno na: http://docs.cpanel.net/twiki/bin/view/AllDocumentation/InstallationGuide/Quick-StartInstallationGuide, . 08.09.2014 15:18.

[6] DirectAdmin spisek funkcionalnosti. Dostopno na: http://www.directadmin.com/features.html. 08.09.2014 15:40.

[7] TouchSwipe JQuery plugin. Dostopno na: http://www.awwwards.com/touchswipe-a-jquery-plugin-for-touch-and-gesture-based-interaction.html. 15.09.2014 15:50.

[8] The History of Java Technology. Dostopno na: `http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html`, . 06.09.2014 12:35.

[9] Once upon an OAK. Dostopno na: `http://www.artima.com/weblogs/viewpost.jsp?thread=7555`, . 06.09.2014 12:35.

[10] S. Liang, C. Bracha. Dynamic Class Loading in the Java Virtual Machine. *OOPSLA 98 Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, str. 36–44, 1998.

[11] T. Lindholm, F. Yellin, G. Bracha, A. Buckley. *The Java Virtual Machine Specification, Java SE7 Edition*. Združene države Amerike: Oracle, 2013.

[12] Z. Qian, A. Goldeberg, A. Coglio. A Formal Specification of Java Class Loading. *OOPSLA 00 Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, str. 325–336, 2000.

[13] C. S. Horstmann, G. Cornell. *Core Java Volume II Advanced Features 9th Edition*. Združene države Amerike: Prentice Hall, 2013.

[14] Z. Shams, S. H. Edwards. Reflection Support: Java Reflection Made Easy. *Open Software Engineering Journal*, 7:38–52, 2013.

[15] S. Drossopoulou, S. Eisenbach. Manifestations of Java Dynamic Linking - an approximate understanding at source language level. 2002.

[16] I. R. Forman, N. Forman. *Java Reflection in Action*, str. 141–142. Združene države Amerike: Manning, 2005.

[17] A. Tozawa, M. Hagiya. Careful Analysis of Type Spoofing. *Java-Informations-Tage*, str. 290–296, 1999.

[18] C. Hunt, B. John. *Java Performance.* Združene države Amerike: Addison-Wesley, 2012.

[19] S. Oaks. *Java Security 2nd Edition.* Združene države Amerike: O'Reilly, 2001.

[20] M. Rettig. Prototyping for tiny fingers. *Community ACM*, 37(4):21–27, 1994.

[21] J. L. Schilling. The Simplest Heuristics May Be The Best in Java JIT Compilers. *ACM Sigplan Notices*, 38:36–46, 2003.

[22] A. Herzog, N. Shamberi. Using the Java Sandbox for Resource Control. *NORDSEC 02 Proceedings of the 7th Nordic Workshop on Secure IT Systems*, str. 135–147, 2002.

[23] T. Pender. *UML Bible.* Združene države Amerike: Wiley Publishing, 2003.

[24] T. Jensen, D. Le Metayer, T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. *Proceedings 1998 International Conference on Computer Languages*, str. 4–15, 1998.

[25] J. M. Wargo. *Apache Cordova 3 Programming*, str. 1–15. Združene države Amerike: Addison-Wesley, 2013.

[26] T. Hill, R. Westbrook. SWOT Analysis: It's time for a product recall. *Long Range Planning*, str. 46–52, 1997.