CORE
Metadata, citation and similar papers at core.ac.uk
Provided by ePrints.FRI

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Dimitar Kotevski

Razvoj razširljive programske opreme v Javi

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana 2014

To delo je ponujeno pod licenco *Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani `http://creativecommons.org/licenses/by-sa/4.0/`.

Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco MIT License. To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani `http://opensource.org/licenses/MIT`.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika dela:

Proučite možnosti za razvoj razširljivih javanskih aplikacij s posebnim poudarkom na uporabi vtičnikov. Analizirajte razpoložljiva ogrodja za izdelavo vtičnikov in na tej osnovi izdelajte lastno ogrodje. Opišite funkcionalnost izdelanega ogrodja, še zlasti rešitve na področju zagotavljanja varnosti, ki preprečujejo, da bi med izvajanjem vtičnika prišlo do zlorab ali izgube podatkov. Uporabo ogrodja prikažite na konkretnem primeru.

MENTOR: izr. prof. dr. Viljan Mahnič

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Dimitar Kotevski, z vpisno številko **63070419**, sem avtor diplomskega dela z naslovom:

Razvoj razširljive programske opreme v Javi

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Viljana Mahniča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 6. septembra 2014 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Viljanu Mahniču za vso strokovno pomoč pri izdelavi diplomskega dela ter družini in prijateljem za veliko podporo in potrpežljivost v casu študija.

Svojim najdražjim.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Razširljiva programska oprema **3**

2.1 Statično razširljiva programska oprema 4
2.2 Dinamično razširljiva programska oprema 5
2.3 Vtičniška arhitektura 5

3 Obstoječe rešitve **7**

3.1 OSGi 7
3.2 Katera je prava izbira? 9

4 Ogrodje jPluggy **11**

4.1 Struktura vtičnikov 11
4.2 Nalaganje vtičnikov 16

5 Varnost pri dinamičnem izvajanju kode **21**

5.1 Izolacija kode s pomočjo peskovnika 23
5.2 Nalaganje razredov 24
5.3 Dovolilnice v Javi 29
5.4 Dovolilnice v jPluggy 34
5.5 Upravljalec varnosti v jPluggy 39

6 Primer uporabe ogrodja jPluggy **43**

6.1 Definicija vmesnikov 43

6.2 Razvoj vtičnikov 45

6.3 Uporaba vtičnikov 47

7 Zaključek **51**

Seznam uporabljenih kratic

kratica	angleško	slovensko
RTP	Real-time Transport Protocol	Transportni protokol v realnem času
AWT	Abstract Window Toolkit	Abstraktno orodje za okna
SQL	Structured Query Language	Strukturirani povpraševalni jezik
IDE	Integrated development environment	Integrirano razvojno okolje
IT	Information technology	Informacijska tehnologija

Povzetek

V diplomskem delu je predstavljen razvoj razširljive programske opreme v Javi. V uvodnem delu diplomske naloge je predstavljen proces razvoja programske opreme in razlogi, zakaj se večina v poslovnem svetu odloča za razvoj razširljive programske opreme. V nadaljevanju sta opisana oba obstoječa tipa razšiljivosti in najbolj uporabljeni vzorec razvoja te (vtičniki). Analizirano je bilo tudi najbolj razširjeno ogrodje, ki obstaja (OSGi), in predstavljen je razlog, zakaj se ga nismo odločili uporabljati in šli čez proces razvoja svojega ogrodja. V osrednjem delu diplomske naloge je opisano ogrodje, ki smo ga razvili, s poudarkom na najbolj pomembnem problemu pri razvoju takega ogrodja - varnosti. V zaključnem delu diplomske naloge je podan primer uporabe ogrodja kot dokaz, da je ogrodje skoraj nevidno za uporabnika, in da ne poveča kompleksnosti aplikacije.

Ključne besede: Razširljiva programska oprema, Java, varnost, ogrodje, vtičniki, OSGi

Abstract

This thesis addresses the subject of developing extensible software in Java. In the introductory part, the analysis of the process of software development is presented as well as the main reasons why the majority in the IT business world decide to develop extensible software. In the following part we have the two types of extensible software and the most used design pattern in that area (plugins). Analysis of the widely used framework for developing extensible software (OSGi) were made and the facts behind the reasons for not using that framework and developing our own are presented. In the central part of the thesis a deep analysis of our newly developed framework with emphasis on the main problem in developing such framework - security has been made. In the final part we present the usage example to prove that the framework is almost invisible to the end user, and that it doesn't add complexity to the application.

Keywords: Extensible software, java, security, framework, plugins, OSGi

Poglavje 1

Uvod

Danes v poslovnem IT svetu poleg glavnega vprašanja ***"Kakšna bo cena tega produkta in kdaj bo končan?"*** stranke, ki naročajo programsko opremo, pogosto vprašajo ***"Kako lahko naredimo morebitne razširitve produkta čim manj »boleče« in poceni?"***. Odprtokodna skupnost in njena hitra razširitev v zadnjih letih je povzročila, da veliko število razvijalcev vsak dan prispeva kodo v odprtokodne programske pakete.

Zgoraj navedena razloga sta odločilna pri odločitvi za razvoj razširljive programske opreme. Dokaz za to je vse večje število aplikacij, ki kot način razširitve uporablja vtičnike. Kot primer si lahko pogledamo spletne brskalnike, razna integrirana razvojna okolja (angl. IDE), novejše urejevalnike besedila itd. Glavna prednost vtičniške arhitekture je to, da lahko vsak z lahkoto prispeva svoj kos kode, glavni problem te arhitekture pa je to, da če zamisel ni izvedena pravilno, lahko pride do resnih varnostnih problemov. Zaradi tega problema je zelo priporočljivo uporabljati katero izmed obstoječih ogrodij pri izdelavi razširljive aplikacije.

Prav s to problematiko se ukvarjamo v diplomskem delu. Po ugotovitvi, da obstaja le eno napredno ogrodje za razvoj razširljive programske opreme, ki pa je za večino uporabnikov preveč obsežno in kompleksno in se prav zaradi tega večina odloči za implementacijo lastnega ogrodja, pri tem pa pozabi na varnost in s tem ogroža računalniške sisteme uporabnikov, smo v sklopu te

diplomske naloge razvili ogrodje za razvoj razširljive aplikacije s podporo za vtičnike in s poudarkom na varnost pri izvajanju le-teh.

V nadaljevanju si bomo ogledali proces razvoja programske opreme in kako vtičniška arhitektura vpliva nanj. Podrobno bomo pogledali strukturo ogrodja, ki smo ga razvili in kako je nam uspelo doseči naša dva glavna cilja: **razviti ogrodje, ki je skoraj nevidno za uporabnika, in varno izvajanje vtičniške kode**.

Poglavje 2

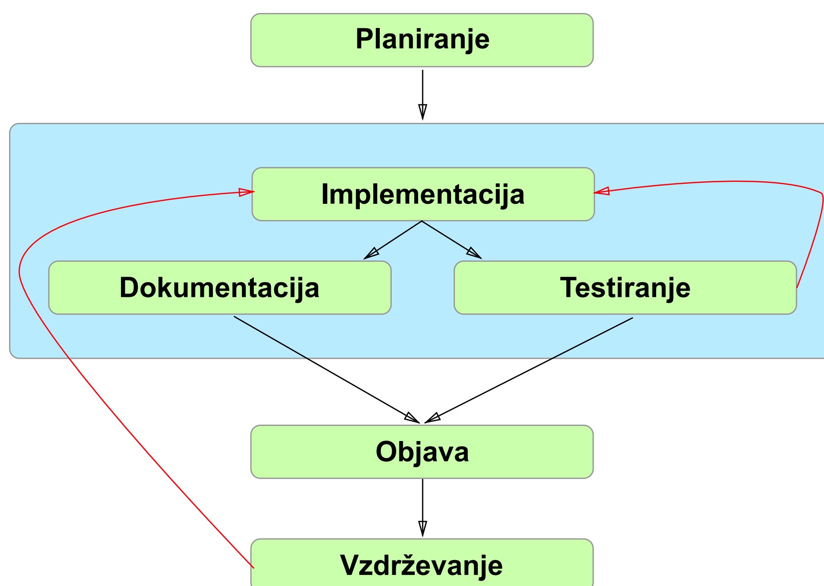
Razširljiva programska oprema

Razvoj programske opreme je proces [4], ki je razdeljen v več faz. Posplošena različica teh faz je tista, ki je najbolj pomembna za razvijalce (Slika 2.1):

- Planiranje
- Implementacija, testiranje in dokumentacija
- Vzdrževanje

Druga faza (Implementacija, testiranje in dokumentacija) je najbolj zamudna. V tej fazi skupina programerjev implementira vse funkcije programske opreme. Ko so vse funkcije implementirane, nastopi faza testiranja, kjer skupina testerjev preveri njeno delovanje. Če ti potrdijo, da programska oprema deluje brezhibno, gre razvoj v naslednjo fazo. V nasprotnem primeru gre razvoj eno fazo nazaj, kjer razvijalci popravijo najdene napake. Ta procesa se ponavljata dokler skupina testerjev ne potrdi, da programska oprema deluje brezhibno.

Razvoj se ponavadi nikoli ne ustavi že pri prvi različici aplikacije. Skoraj vedno dodajamo nove funkcije aplikaciji. Zgornji proces je enak tudi pri dodajanju novih funkcij že obstoječi programski opremi. Ker želimo omejiti čas trajanja tega procesa, je zelo pomembno, da je programska oprema dobro napisana, in da je enostavno razširljiva.



Slika 2.1: Proces razvoja programske opreme

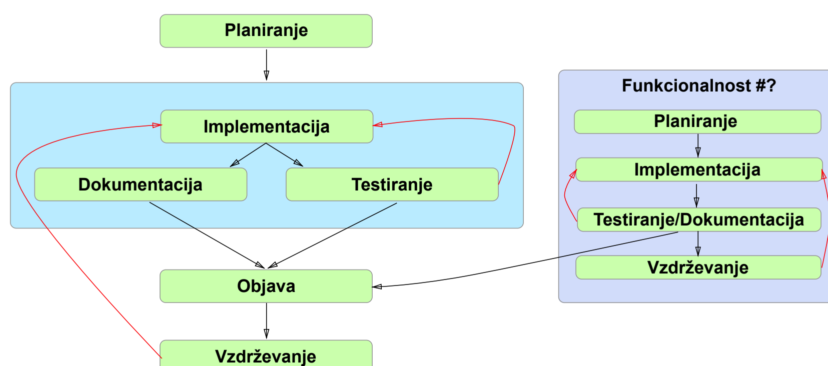
Lahko rečemo, da je vsaka aplikacija razširljiva. Vedno lahko dodamo nove funkcije že obstoječi kodi. Vprašanje pa je, kako lahko to naredimo.

Obstajata dva tipa razširljive programske opreme. **Statično** in **dinamično** razširljiva programska oprema.

2.1 Statično razširljiva programska oprema

Vsaka programska oprema je statično razširljiva. Vsako kodo lahko razširimo na način, da dodamo nove metode in razrede, hkrati pa stare prilagodimo novemu načinu delovanja. Da bi to naredili, je potreben nov cikel razvoja, od implementacije, testiranja in dokumentacije. Kot smo že omenili, je ta proces dolg in se ponavlja, dokler programska oprema ne deluje brezhibno.

Če želimo imeti čim krajši proces statičnega razširjanja aplikacije, moramo imeti to v mislih od samega začetka, že pri načrtovanju arhitekture aplikacije. V nadaljevanju se moramo držati tudi zelo natančnih pravil glede načina kodiranja in dokumentiranja kode, saj se v praksi zelo redko zgodi, da razvijalci, ki so razvijali prvo različico aplikacije, razvijajo tudi razširitve.



Slika 2.2: Proces razvoja dinamično razširljive programske opreme

2.2 Dinamično razširljiva programska oprema

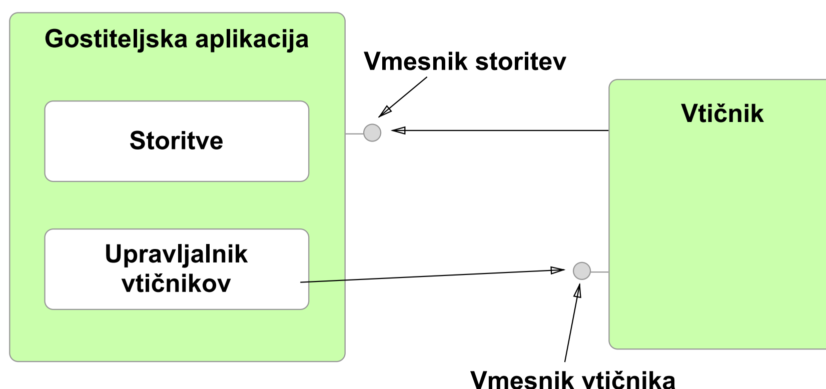
Dinamično razširljiva aplikacija je aplikacija, ki jo lahko razširimo brez spreminjanja originalne kode. Razvijalci aplikacije in "3rd party" razvijalci lahko to naredijo z dodajanjem vtičnikov. Da pa je to mogoče, mora biti aplikacija tako sestavljena oz. mora biti zasnovana z vtičniško arhitekturo.

Pri zasnovi tako aplikacijo razdelimo aplikacijo na dva dela, kot je prikazano na sliki 2.2. Prvi del je gostiteljska aplikacija, ki vsebuje osnovne funkcionalnosti. Drugi del aplikacije je sestavljen iz vtičnikov, ki jih dodatno razvijemo in dinamično vključimo v aplikacijo. S tem dosežemo to, da je razvoj vtičnikov ločen od razvoja gostiteljske aplikacije, in da ne ponavljamo procesa razvoja (implementacija, preverjanje, dokumentacija) za gostiteljsko aplikacijo pri razvoju novih vtičnikov. Stranski učinek tega je **povečevanje stabilnosti** aplikacije in **zmanjševanje cene** razvoja.

2.3 Vtičniška arhitektura

Po definiciji [7] je **vtičnik** *komponenta, ki doda določeno funkcionalnost obstoječi aplikaciji.* To je točno to, kar potrebujemo, da naredimo aplikacijo dinamično razširljivo.

Da lahko aplikacija uporablja vtičnike, mora omogočati določene storitve (angl. Services) preko vmesnikov. Za boljše razumevanje lahko pogledamo



Slika 2.3: Vtičniška arhitektura

sliko 2.3 .

Kot je prikazano na sliki 2.3, aplikacija omogoča storitve preko vmesnika. Preko teh storitev, se vtičniki registrirajo pri aplikaciji in se "dogovorijo" za protokol pri prenosu podatkov. Vtičniki so odvisni od storitev, ki jih ponuja aplikacija in ne delujejo samostojno. Obratno pa velja, da aplikacija deluje samostojno (ni odvisna od vtičnikov). S tem je omogočeno, da uporabnik dodaja ali briše vtičnike dinamično, brez spreminjanja aplikacije.

Aplikacija mora torej poskrbeti za vmesnike, hkrati pa mora skrbeti tudi za življenjski cikel vtičnikov. Ta je zelo odvisen od narave vtičnikov. Zato definiramo najbolj osnovni življenjski cikel vtičnikov kot seznam stanj: *pripravljen*, *zagnan* in *zaustavljen*. Ker upravljanje življenskega cikla vtičnikov ni zelo enostaven proces, je najbolje, da uporabljamo ogrodja, ki nam točno to omogočajo. Dodaten in zelo pomemben razlog za uporabo ogrodja je tudi varnost pri delu z vtičniki. Če ne poskrbimo za to, lahko vtičnik naredi nepopravljivo škodo, kot je recimo izbris vseh podatkov z diska ali kraja občutljivih podatkov.

Poglavje 3

Obstoječe rešitve

Po raziskavi smo ugotovili, da obstaja le ena napredna rešitev za razvoj razširljive programske opreme - OSGi, ki pa je le specifikacija.

3.1 OSGi

OSGi (Open Service Gateway Initiative) je javansko ogrodje za razvoj modularne programske opreme in knjižnic [6].

Aplikacije oz. komponente so v obliki paketa, ki so lahko nameščeni, zagnani, zaustavljeni, odstranjeni na daljavo in to brez zaustavitev glavne aplikacije. Življenjski cikel ene komponente je popolnoma voden s strani OSGi-ja.

Na sliki 3.1 je prikazana arhitektura OSGi-ja. Za boljšo predstavo si bomo na kratko ogledali posamezne plasti ogrodja.

- Paketi (angl. Bundles)

  Paketi so navadni *jar* paketi z dodatnimi podatki v manifest datoteki

- Storitve (angl. Services)

  Ta plast povezuje na dinamičen način tako, da omogoča model objavi-najdi-poveži (angl. publish-find-bind) za navadne Java vmesnike

Slika 3.1: Arhitektura OSGi

(POJI) in navadne Java objekte (POJO).

- Register storitev (angl. Services Registry)

  Definira API za upravljanje s storitvami

- Življenjski cikel (angl. Life Cycle)

  Definira API za življenjski cikel

- Modul (angl. Modules)

  Plast, ki definira enkapsulacija in definicija odvisnosti.

- Varnost (angl. Security)

  Definira varnost

3.1.1 Implementacije

Ker je OSGi le specifikacija, obstaja več implementcij. Najbolj znane so:

- Apache Felix - `http://felix.apache.org/`

- Concierge - `http://concierge.sourceforge.net/`
- Eclipse Equinox - `http://eclipse.org/equinox`
- JBoss - `http://www.jboss.org/jbossas/osgi`
- Hitachi - `http://www.hitachi-solutions.com/superj/sp/sjf/`
- Knopflerfish - `http://www.knopflerfish.org/`

3.1.2 Uporaba

OSGi se uporablja v nekaterih aplikacijah. Najbolj znana aplikacija za razvijalce je Eclipse IDE. Celotni sistem za vtičnike v Eclipse IDE-ju deluje s pomočjo OSGi. OSGi se uporablja tudi v različnih aplikacijah za mobilne telefone, avtmobile, industrijske avtomatizacije, aplikacijske strežnike, itd.

3.2 Katera je prava izbira?

Večina razvijalcev vidi OSGi kot preveč kompleksno rešitev za navadne aplikacije [9] in se zato raje odloča za razvoj lastnega ogrodja. Težava pri tem je, da skoraj nobena takšna rešitev ne upošteva nobenih varnostnih navodil pri izvajanju dinamične kode. Zaradi tega in **predvsem zaradi boljšega razumevanja varnosti pri dinamičnem izvajanju kode** smo se odločili, da bomo tudi mi napisali enostavno ogrodje za delo z vtičniki s poudarkom na varnost. V poglavju 4 si bomo ogledali splošno arhitekturo našega ogrodja, celotno poglavlje 5 pa bomo posvetili varnosti pri dinamičnem izvajanju kode.

Poglavje 4

Ogrodje jPluggy

Kot smo že omenili v 3.2, smo se odločili, da bomo napisali enostavno ogrodje za delo z vtičniki s poudarkom na varnosti pri izvajanju dimanične kode.

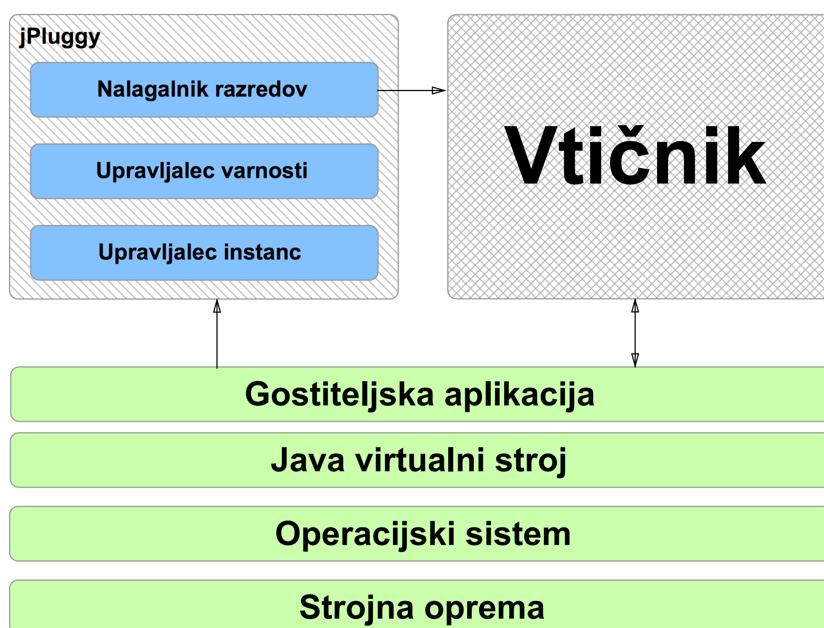
Definicija "ogrodja" [3] pravi, da je ogrodje knjižnica, ki ponuja generično rešitev za določen problem.

JPluggy je zelo fleksibilno ogrodje. Ne omejuje uporabnika in ne uveljavlja svojih pravil. Skrbi le za življenjski cikel vtičnikov in za njihovo varno izvajanje. Fleksibilnost gre do te mere, da je edino pravilo to, da mora uporabnik pridobiti instanco vtičnika preko ogrodja. Vsi nadaljni klici gredo mimo ogrodja. To pomeni, da se po pridobitvi instance vtičnik uporablja kot vsaka navadna Java knjižnica. Za boljše razumevanje lahko pogledamo sliko 4.1.

4.1 Struktura vtičnikov

Vtičnik je lahko sestavljen iz enega razreda ali pa iz več razredov, ki *komunicirajo* med seboj. Ne glede na to, ali je vtičnik enorazredni ali večrazredni, mora imeti vstopno točko oz. vstopni razred. Vstopni razred je vsak razred, ki je označen z zaznambo (angl. annotation) **@Plugin()** in ima konstruktor brez parametrov.

Vsak vtičnik mora biti pravilno zapakiran preden ga lahko ogrodje upora-



Slika 4.1: Struktura ogrodja jPluggy

blja. JPluggy pozna dva načina pakiranja. Vtičnik je lahko zapakiran v *jar* datoteki kot vsaka navadna Java knjižnica, lahko pa so vsi vtičniški razredi v eni datoteki.

Kako ogrodje najde vtičnike in jih naloži, je podrobno opisano v delu 4.2.

Primer enostavnega vtičnika:

```
@Plugin("hello")
public class HelloPlugin {

  public HelloPlugin() {
  }

  public String introduce() {
    return "Hello! I am plugin 'hello'";
  }
}
```

Interno ima ogrodje možnost razvrstiti vtičnike po tipu. To je pomembna

fukncionalnost, ki omogoča aplikaciji, da ima več kot en tip vtičnikov. Na primer, če govorimo o strežniški aplikaciji, ki skrbi za osebne dokumente (fotografije, video posnetki, dokumenti) in ima vmesnik za pregled teh podatkov, ima lahko ta aplikacija vsaj dva tipa vtičnikov: vtičniki za pregled različnih tipov datotek in vtičniki za tok (angl. stream) datotek preko različnih protokolov (DLNA, RTP).

To dosežemo tako, da za vsak tip vtičnikov napišemo po en vmesnik.

```
public interface FileViewer {
  /**
   * List all available items
   */
  public List<Item> list();

  /*
   * Return HTML representation of the file
   */
  public String getHtml(Item item) {
}
```

```
public interface FileStreamer {
  /**
   * Returns the bytes[] array of the file in the appropriate
  format.
   */
   public byte[] getBytes(Item item);
}
```

Vtičniki, ki skrbijo za prikaz osebnih dokumentov, morajo implementirati vmesnik *FileViewer*. Kot primer sta prikazana dva vtičnika: za prikaz slik in za prikaz tekstovnih datotek. Vtičniki, ki skrbijo za tok datotek, pa morajo implementirati vmesnik *FileStreamer*. Kot primer sta prikazana vtičnika, ki ponujata audio datoteke preko DLNA in RTP protokola.

```
@Plugin("pictures_viewer")
public class PicturesViewer implements FileViewer {

  public PicturesViewer() {
  }

  public List<Item> list() {
    /* Return empty list */
    return new ArrayList<Item>();
  }

  public String getHtml(Item item) {
    return "<img src='" + item.getPath() + "'" />
  }
}
```

```
@Plugin("text_viewer")
public class TextViewer implements FileViewer {

  public TextViewer() {
  }

  public List<Item> list() {
    /* Return empty list */
    return new ArrayList<Item>();
  }

  public String getHtml(Item item) {
    return "<pre>" + item.getContent() + "</pre>";
  }
}
```

```
@Plugin("dlna_audio_streamer")
```

```
public class DLNAAudioStreamer implements FileStreamer {

  public DLNAAudioStreamer() {
  }

  public byte[] getBytes(Item item) {

    /* Here implement the DLNA stream protocol */
    return new byte[0];
  }
}
```

```
@Plugin("rtp_audio_streamer")
public class RTPAudioStreamer implements FileStreamer {

  public RTPAudioStreamer() {
  }

  public byte[] getBytes(Item item) {

    /* Here implement the RTP stream protocol */
    return new byte[0];
  }
}
```

JPluggy bo oba tipa vtičnikov obravnaval popolnoma enako. Edina razlika za gostiteljsko aplikacijo bo pri klicu *getPluginsOfType (type)*, ki je definirana v razredu *JPluggyManager* in je edini način za pridobitev instanc vtičnikov. Pri klicu *getPluginsOfType (FileViewer.class)* bo metoda vrnila le razrede, ki implementirajo vmesnik *FileViewer*. Pri klicu *getPluginsOfType (FileStreamer.class)* bo pa metoda vrnila le razrede, ki implementirajo vmesnik *FileStreamer*. Na ta način gostiteljska aplikacija lahko loči vtičnike po tipu.

Primer pridobitve vtičnikov različnega tipa:

```
/* Get only FileViewer plugins */
List<FileViewer> fileViewers = jPluggy.getPluginsOfType(
 FileViewer.class);

/* Get only FileStremer plugins */
List<FileStreamer> fileStreamers = (List<FileStreamer>)
 jPluggy.getPluginsOfType("FileStreamer");

/* Get all plugins */
List<Object> allPlugins = jPluggy.getPluginsOfType(Object.
 class);
```

4.2 Nalaganje vtičnikov

Pri inicializaciji ogrodje s pomočjo Java reflection-a poišče vtičniške razrede v domači mapi vtičnikov, ki je navedena kot parameter v konstruktorju ogrodja.

```
/**
   * Creates a new plugin manager
   *
   * @param pluginsDir
   *       The dir where the plugins live
   * @param pluginsStorageDir
   *       The dir where the plugins can save files
   */
  public JPluggyManager(File pluginsDir, File
 pluginsStorageDir) {...}
```

Najprej ogrodje sestavi dva seznama, seznam *jar* datetok in seznam direktorijev, ki se nahajajo v vtičniškem direktoriju.

```
File[] dirs = pluginsDir.listFiles(DIR_FILTER);
File[] jars = pluginsDir.listFiles(JAR_FILTER);
```

V naslednjem koraku ogrodje za vsako najdeno *jar* datoteko oz. direktorij pokliče metodo *loadPluginClasses(File dirOrJar)*, ki poskuša naložiti vse razrede iz te datoteke oz. mape in istočasno poišče vstopni razred vtičnika. Nalaganje razredov poteka preko posebnega nalagalnika, ki ga ta metoda naredi za vsak vtičnik posebej. To naredimo zaradi varnosti. Postopek je bolj podrobno opisan v poglavju 5.

```
/* Need to create a separate class loader for every plugin */
  PluginClassLoader classLoader = createClassLoader(dirOrJar);

...

  /* Get a list of all class files inside the dir/jar file. */
  Collection<String> classes = getClasses(dirOrJar);

  /* Load the classes in their class loader */
  for (String canonicalName : classes) {
    try {
        Class<?> loadedClass = classLoader.loadClass(
 canonicalName);
          classLoader.addProtectionDomain(loadedClass.
 getProtectionDomain());

          if (loadedClass.isAnnotationPresent(Plugin.class)) {
            pluginClass = loadedClass;
          }
      } catch (Throwable e) {...}
  }
```

Če se nalaganje razredov izvede brez napak in direktorij oz. jar datoteka vsebuje vstopni vtičniški razred (razred, označen z zaznambo (angl. annotation) **@Plugin()**), dodamo vstopni vtičniški razred v seznam naloženih vtičnikov. V nasprotnem primeru izpišemo sporočilo, da nalaganje določenega razreda ni uspelo in nadaljujemo z nalaganjem naslednjega vtičnika.

```
/* Check for failure */
```

```
    if (pluginLoadFailed || pluginClass == null) {
        logger.error("Error loading the plugin {}. See the debug
 log for details.", dirOrJar);
        return false;
    }

    synchronized (loadedPluginClasses) {
        loadedPluginClasses.put(pluginClass, classLoader);
    }

    logger.info("Successfully loaded the plugin {}", pluginClass
);
    return true;
```

Do trenutka incializacije ima ogrodje le reference na vtičniške razrede, ki so shranjene v seznamu *loadedPluginClasses*. Šele ko uporabnik pokliče metodo *getPluginsOfType(...)*, ogrodje naredi instanco določenega razreda in vrne instanco uporabniku.

```
Set<Class<?>> loadedClasses = pluginClassLoaderWorker.
 getLoadedPluginClasses();

...

/* Loop through all the loaded plugin classes and check for
 the type */
 for (Class<?> theClass : loadedClasses) {
     if (type.isAssignableFrom(theClass)) {
         Object instance = instanceManager.newInstance(
 theClass);
         if (instance != null) {
             instances.add((T) instance);
         }
     }
 }
```

Vredno je omeniti, da pri inicializaciji vtičnika preverimo, če ima vtičnik odobrene vse dovolilnice, ki jih zahteva. [1]

```
/* First check if the permissions are accepted */
  if (!securityManager.isApproved(type)) {
      logger.error("The plugin has not been approved yet!
 Aborting...");
      return null;
  }
```

4.2.1 Zaznavanje sprememb v vtičniškem direktoriju

Želeli smo narediti proces dodajanja/odstranevanja vtičnikov, ki bi bil čimbolj enostaven za uporabnike. Zaradi tega smo se odločili, da bomo naredili čuvaja, ki bo zaznaval spremembe v vtičniškem direktoriju in bo to sporočil gostiteljski aplikaciji. V ta namen smo uporabili funkcionalnost, ki jo ponuja Java platforma [8].

Čuvaj dela tako, da v ločeni niti čaka na spremembe, ki jih sistem sporoči. Trenutna implementacija se zaveda samo dveh tipov sprememb: da je bil nov vtičnik dodan v vtičniški direktorij, in da je obstoječi vtičnik zbrisan iz vtičniškega direktorija.

```
/* The available event types */
  public enum EventType {
      ADDED,
      REMOVED;
  }
```

Ko čuvaj zazna spremembo, to sporoči poslušalcu sprememb, ki je nastavljen preko klica *JPluggyManager.setDirectoryWatcher()*. Poslušalec sprememb je lahko vsak razred, ki implementira vmesnik *PluginsDirectoryWatcher*.

[1] Dovolilnice so podrobno opisane v poglavju 5

```
public interface PluginsDirectoryWatcher {
    /**
     * Called when a change happens in the plugins directory
     *
     * @param event
     */
    public void onChange(PluginsDirectoryChangeEvent event);
}
```

Poslušalec sprememb lahko reagira na to informacijo po želji.

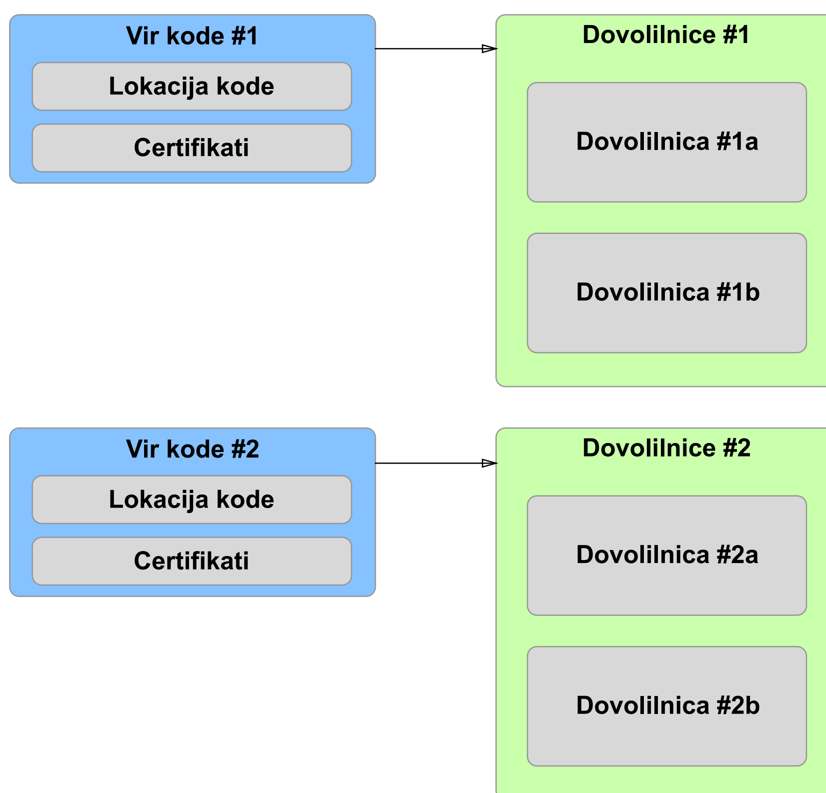
Poglavje 5

Varnost pri dinamičnem izvajanju kode

Glavni problem pri dinamičnem izvajanju kode je varnost. Pri nalaganju kode vedno obstaja nevarnost, da lahko ta naredi škodo. Na primer določena koda lahko pri nalaganju zbriše celotni disk. Drugi, bolj nevarni primer, je koda, ki pošilja celotni vpis, ki ga uporabnik vtipka preko tipkovnice na nek strežnik. Takšna koda lahko "ukrade" občutljive podatke kot so uporabniška imena in gesla in lahko dostopa do kritičnih storitev kot je elektronsko bančništvo in podobno. Zaradi tega je varnost pri dinamičnem nalaganju kode ključnega pomena.

Platforma Java je bila od samega začetka zasnovana s ciljem zmožnosti vzpostavljanja varnosti [1] [2]. Ena od glavnih karakteristik Jave na začetku so bili Java Appleti. Appleti so omogočali interaktivnost spletnih strani, takrat ko se je HTML uporabljal le za prezentacijo besedila in slik. Appleti so delovali tako, da je uporabniški brskalnik pri nalaganju spletne strani naložil še kodo appleta v Java virtualni stroj in stroj je začel izvajati to kodo. Ker je koda appleta naložena preko spleta in lahko prihaja iz nepreverjenega vira, lahko rečemo, da je nevarna.

Prav zaradi tega ima Java platforma dober varnostni mehanizem. V prvi verziji Jave je bil ta mehanizem zelo enostaven. Obstajala sta dva načina



Slika 5.1: Dovolilnice v Javi

delovanja. Razredi, ki so bili naloženi iz lokalnega diska računalnika, so imeli popoln dostop do vseh sistemskih virov. Na drugi strani so bili razredi, ki so bili naloženi preko spleta (Java Appleti), ki pa so bili "zaprti" v peskovniku (angl. sandbox). Ti razredi so lahko le risali na zaslon in brali uporabniški vhod (na primer tipkovnica).

To se je spremenilo v drugi verziji Jave, ko je platforma dobila bolj kompleksen mehanizem varnosti.

Od te verzije naprej ima vsak vir kode (angl. Code base) svoje dovolilnice, kot je prikazano na sliki 5.1. Vir kode je definiran z lokacijo kode, ki je URL v primeru razredov appleta in lokacija na disku v primeru lokalnih razredov. Dodatno je še seznam certifikatov, s katerimi je koda podpisana. Slednji del je opcijski, ker podpis kode ni obvezen. Dovolilnice kode pa povejo, do katerih sistemskih virov lahko koda dostopa in so vezane na en vir

kode. Te dovolilnice preveri vgrajeni sistemski varnostni upravljalec (angl. SecurityManager). Na podlagi teh dovolilnic, upravljalec dovoli ali prepove dostop do enega sistemskega vira, kot je na primer dostop do datoteke na disku.

Primer dovolilnice, ki omogoča bralno-pisalni dostop do vseh datotek v '/tmp' mapi:

```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

V najnovejši verziji Jave je privzeto delovanje varnostnega mehanizma zelo podobno delovanju istega mehanizma prve verzije. Vsa koda, ki je naložena iz spleta (sem spadajo vsi razredi, ki jih en applet uporablja), je zaprta v peskovniku (angl. sandbox) in dostop do sistemskih virov je zelo omejen oz. prepovedan. Na drugi strani je koda, ki je naložena iz lokalnega diska, ki pa ima vsa dovoljenja, in lahko dostopa do vseh sistemskih virov brez omejitev.

5.1 Izolacija kode s pomočjo peskovnika

Če želimo varno izvajati tudi kodo, ki jo naložimo iz lokalnega diska, ji moramo omejiti dostop do sistemskih virov na isti način, kot to naredi sama platforma kodi, ki je bila naložena iz spleta. Ves ta mehanizem je znan pod imenom "Zapiranje kode v peskovniku" (angl. Sandboxing). Da bi to dosegli, bomo uporabili že obstoječe močne varnostne mehanizme Jave, kot so: Upravljalec varnosti in dovolilnice.

Kot je bilo že omenjeno, je privzeto delovanje tako, da je lokalni kodi omogočen popoln dostop do vseh sistemskih virov. To lahko spremenimo tako, da nastavimo upravljalca varnosti na sistemski ravni z naslednjim ukazom:

```
System.setSecurityManager(new SecurityManager());
```

Od te vrstice naprej nastavljeni upravljalec varnosti preverja dovolilnice za vsako vrstico kode, ki želi dostopati do kakršnegakoli sistemskega vira. Ne da bi naredili karkoli drugega, s tem preprečimo dostop do sistemskih virov celotni kodi trenutnega procesa. Število primerov, v katerem je ta način dela koristen, je zelo majhen.

Da bi dobili koristen varnosti sistem, moramo poskrbeti, da je naša koda dobro struktuirana oz. razdeljena na ločene vire kode (angl. Code base), in da ima vsak tak vir kode svoje dovolilnice, ki jih bo uporabnik eksplicitno omogočal.

Podoben varnosti sistem ima operacijski sistem Android. Vsaka Android aplikacija zahteva neko množico dovolilnic. Pri namestitvi aplikacije je uporabnik prisiljen pogledati seznam dovolilnic in odobriti ali zavrniti uporabo teh dovolilnic s strani aplikacije. Če uporabnik zavrne uporabo, aplikacije ni mogoče namestiti. S tem Android zagotovi, da aplikacije dostopajo samo do majhne podmnožice sistemskih virov, in da je uporabnik seznanjen s tem.

Poskusili smo narediti podoben sistem. Vsak vtičnik se naloži preko ločenega nalagalnika razredov. S tem zagotovimo izoliranost kode in to, da je vsak vtičnik v ločenem viru kode (angl. Code base). Dodatno ima še vsak vir kode oz. vtičnik svojo množico dovolilnic, ki mu omogoča dostop do sistemskih virov. Pred tem pa mora uporabnik pogledati zahtevane dovolilnice vtičnika in jih odobriti ali zavrniti. Enako kot pri Androidu, vtičnika z zavrnjenimi dovolilnicami ni mogoče izvajati.

5.2 Nalaganje razredov

Prvi korak do peskovnika je izolacija potencialno nevarnih razredov. S tem bi dosegli želeno varnost, ker omejimo *vidljivost* razredov. Na ta način lahko kontroliramo, do katerih razredov potencialno nevarna koda lahko dostopa.

Da bi razumeli, kako lahko to dosežemo, moramo najprej razumeti, kako poteka nalaganje razredov. Java v ta namen uporablja nalagalnike razredov (angl. *Class loaders*). Nalagalnik razredov prebere razred (datoteka s

končnico *.class*) z diska ali spleta v primeru applet-a in naloži nabor ukazov tega razreda v navidezni stroj. Vsaka Java aplikacija ima vsaj tri nalagalnike razredov:

- Nalagalnik Bootstrap
- Nalagalnik Extension
- Nalagalnik System/Application

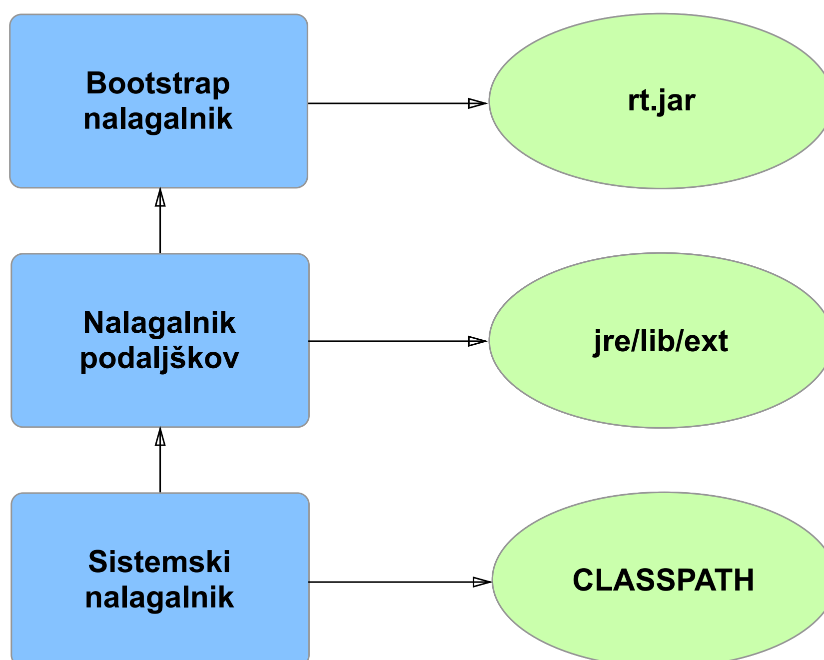
Nalagalnik Bootstrap naloži sistemske razrede in je del Java navideznega stroja. Vsi osnovni podatkovni tipi (String, Float, Integer,...) v Javi so naloženi preko tega nalagalnika. Njegova značilnost je to, da nima svojega razreda v Javi. Če pokličemo metodo *getClassLoader()* za razred, ki je bil naložen preko bootstrap nalagalnika, bomo dobili vrednost *null*.

Nalagalnik Extension naloži standardne knjižnice iz mape *jre/lib/ext*.

Nalagalnik System/Application naloži razrede aplikacije. Poišče vse razrede, ki se nahajajo v direktorijih in *jar* datotekah, ki so nastavljeni v spremenjivki *CLASSPATH*.

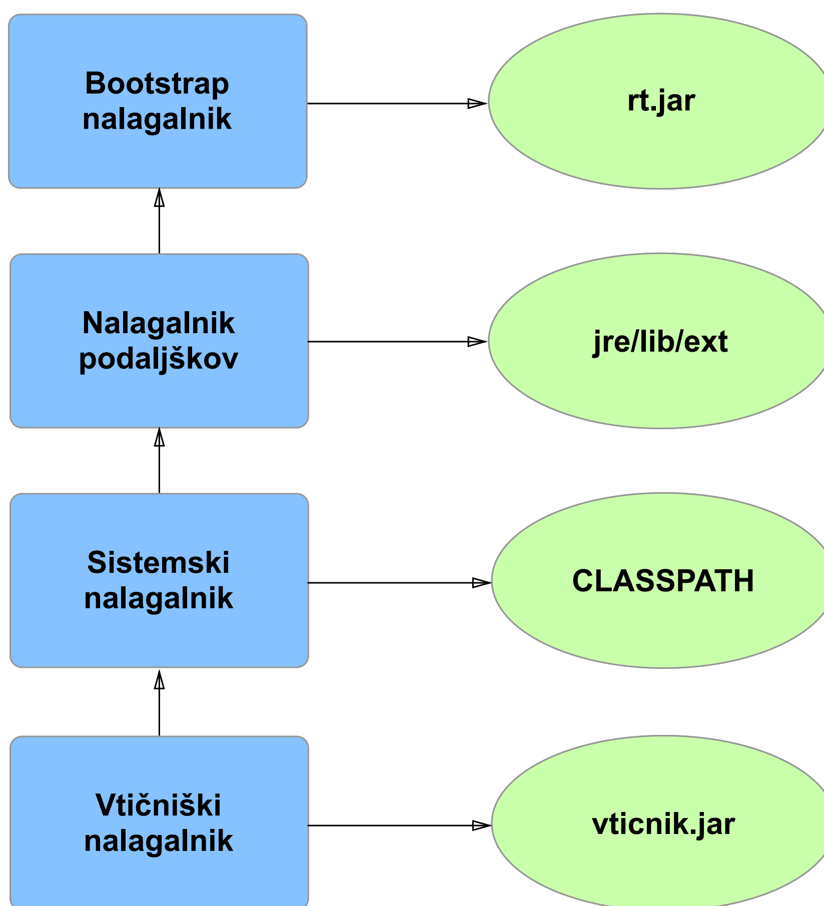
Nalagalniki so v zvezi starš/otrok (angl. *parent/child*). Vsak nalagalnik, ki je bil narejen iz navideznega stroja, z izjemo Bootstrap nalagalnika, ima svojega starša. Ko nalagalnik dobi ukaz, da mora naložiti nek razred, najprej posreduje svojemu staršu (če obstaja), naj naloži ta razred. Samo v primeru, da dobi odgovor, da starš ne more naložiti razreda, je nalagalnik dolžan to storiti sam. Tako delujejo vse standardne implementacije nalagalnikov. Če želi uporabnik napisati svojo implementacijo, mora slediti tem pravilom.

Takšna hierarhija nalagalnikov omogoča željeno izolacijo razredov. Kar moramo narediti, je to, da za nalaganje dinamične kode uporabljamo ločen nalagalnik razredov in ne sistemskega. Ker je bila želja, da imamo popolnoma kontrolirano okolje, smo se odločili za izolacijo vsakega vtičnika posebej. To pomeni, da je vtičnik, ki vsebuje potencialno nevarno kodo, naložen preko svojega nalagalnika in ima **omejen** dostop le do razredov, ki so bili naloženi preko nalagalnikov, ki so v hierarhiji nad nalagalnikom tega vtičnika.



Slika 5.2: Hierarhija nalagalnikov razredov v Javi

Na tem mestu pojasnimo besedo *omejen*. Dostop do razredov namreč omejimo s pomočjo sistema dovolilnic v Javi, ki pa ga bomo bolj podrobno obdelali v delu 5.3. Če ne bi nastavili ustreznih dovolilnic, dostop do razredov, ki so bili naloženi preko ostalih nalagalnikov, ne bi bil omejen. Uporaba obeh sistemov skupaj (razdelitev kode s pomočjo uporabe ločenih nalagalnikov razredov in nastavljanje ustreznih dovolilnic te kode) je "orodje", ki nam omogoča željeno izolacijo (angl. sandboxing).



Slika 5.3: Hierarhija nalagalnikov razredov v jPluggy

Pri izdelavi ogrodja jPluggy smo se odločili, da bomo napisali svojo implementacijo nalagalnika razredov, ki je razširitev razreda *URLClassLoader*, ki pa je že v platformi Java. Naš nalagalnik samo nastavi okolje, ki kasneje omogoča dodajanja novih dovolilnic. Vse ostalo posreduje svojemu nadrazredu (URLClassLoader).

Definicija *PluginClassLoader* zgleda:

```
public class PluginClassLoader extends URLClassLoader {
  /* A set of protectionDomains for this class loader. */
  private Set<ProtectionDomain> protectionDomains = new
 HashSet<>();
  /* The plugin permission collection */
```

```
 private JPluggyPermissionCollection permissions = new
JPluggyPermissionCollection();
 /* A flag indicating that the permissions from the super
class were added to the collection from above */
 boolean superPermissionsAdded = false;

 public PluginClassLoader(URL[] urls, ClassLoader parent) {
     super(urls, parent);
 }

 public void setPluginPermissions(List<JPluggyPermission>
jPluggyPermissions) {
     for (JPluggyPermission jPluggyPermission :
jPluggyPermissions) {
         permissions.add(jPluggyPermission.getJavaPermission
());
     }
 }

 @Override
 protected PermissionCollection getPermissions(CodeSource
codesource) {
     if (!superPermissionsAdded) {
         /* Add the existing permissions */
         Enumeration<Permission> enum_ = super.getPermissions
(codesource).elements();
         while(enum_.hasMoreElements()) {
             permissions.add(enum_.nextElement());
         }
         superPermissionsAdded = true;
     }
     return permissions;
 }

 public void addProtectionDomain(ProtectionDomain
protectionDomain) {
     protectionDomains.add(protectionDomain);
 }
```

```
    public Set<ProtectionDomain> getProtectionDomains() {
        return protectionDomains;
    }
}
```

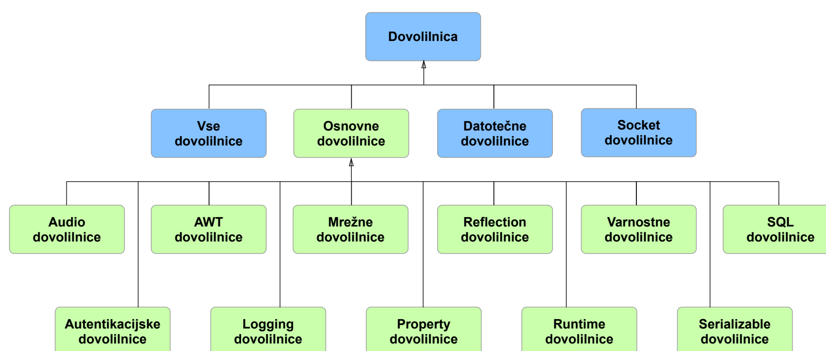
5.3 Dovolilnice v Javi

Kot smo že omenili, ima Java od svoje druge verzije zelo dober in fleksibilen varnosti sistem, ki temelji na njenih dovolilnicah. Dovolilnice se v Javi uporabljajo za preverjanje, če ima koda, ki želi dostopati do enega sistemskega vira, dovoljenje za to. Kot je razvidno iz slike 5.1, enemu viru kode pripada en seznam dovolilnic.

Za preverjanje teh dovolilnic skrbi upravljalec varnosti, če je nameščen. Pri vsakem poskusu dostopa do sistemskega vira, kot je na primer bralni dostop do ene datoteke na disku, upravljalec preveri seznam dovolilnic vseh razredov, ki so v hierarhiji klica. Če vsaj en razred v tej hierarhiji nima ustrezne dovolilnice, je dostop zavrnjen.

Primer hierarhije klica v operacijskem sistemu Android:

```
dime.android.ui.fragments.CentraFragment
android.support.v4.app.Fragment
android.support.v4.app.FragmentManagerImpl
android.support.v4.app.FragmentActivity
android.app.Instrumentation
android.app.Activity
android.app.ActivityThread
android.os.Handler
android.os.Looper
android.app.ActivityThread
java.lang.reflect.Method
com.android.internal.os.ZygoteInit
```



Slika 5.4: Tipi dovolilnic v Javi

```
dalvik.system.NativeStart
```

Če upravljalca varnosti ni, se vsa ta preverjanja ignorirajo, in kodi se dovoli dostop do vseh sistemskih virov.

Java ima dovolilinice za vse sistemske klice. Celoten seznam dovolilnic se nahaja na [5]. Mi si bomo pogledali samo najbolj osnovne.

5.3.1 FilePermission

FilePermission je dovolilnica, ki jo mora imeti vsak kos kode, ki želi dostopati do datotečnega sistema. Konstruktor dovolilnice zahteva naslednje podatke:

- Pot do datoteke/direktorija za katero/ga konstruiramo dovolilnico
- Način dostopa [read, write, execute, delete]

Koda, ki bo poskušala dostopati do datoteke, za katero nima dovolilnice, bo dobila SecurityException.

Vsi dostopi do datotečnega sistema, tudi v standardni Java knjižnici, gredo preko celotnega varnostnega preverjanja. Kot primer lahko vzamemo metodo createNewFile() iz razreda File:

```
public boolean createNewFile() throws IOException {
        SecurityManager security = System.getSecurityManager();
```

```
        if (security != null) security.checkWrite(path);
        if (isInvalid()) {
            throw new IOException("Invalid file path");
        }
        return fs.createFileExclusively(path);
    }
```

Iz zgornjega dela kode lahko ugotovimo, da če imamo aktivni SecurityManager, sistem najprej preveri, če imamo write dovolilnico za pot, kjer želimo ustvariti novo datoteko. Brez te dovolilnice metoda checkWrite() iz razreda SecirutyManager vrže SecurityException in koda se takoj neha izvajati.

5.3.2 ReflectPermission

Reflection mehanizem v Javi omogoča veliko stvari. Prek njega lahko beremo in spreminjamo atribute objekta, tudi če je atribut definiran kot privatni (angl. private). Včasih je to zelo korisno, v našem primeru pa je to zelo nevarno. S pomočjo tega mehanizma nam lahko zlonamerna koda zapiše napačne podatke ali pa prebere občutljive podatke.

Zaradi tega obstaja dovolilnica, ki to omogoča. Brez te dovolilnice noben razred ne more dostopati do nobenih atributov prek zrcalnega mehanizma Jave (angl. Java reflection).

Primer uporabe:

```
ReflectPermission refperm = new ReflectPermission("
    suppressAccessChecks", "");
```

Z zgornjo vrstico kode dovolimo kodi dostop do vseh atributov prek reflection mehanizma. Pri uporabi te dovolilnice moramo biti zelo pazljivi, ker naenkrat dovolimo dostop do **vseh** atributov - tudi do atributov, ki so definirani kot privatni.

Primer uporabe lahko pogledamo v metodi getField() iz razreda Class:

```
    public Field getField(String name)
        throws NoSuchFieldException, SecurityException {
        // be very careful not to change the stack depth of this
        // checkMemberAccess call for security reasons
        // see java.lang.SecurityManager.checkMemberAccess
        checkMemberAccess(Member.PUBLIC, Reflection.
getCallerClass(), true);
        Field field = getField0(name);
        if (field == null) {
            throw new NoSuchFieldException(name);
        }
        return field;
    }
```

5.3.3 RuntimePermission

RuntimePermission je zelo obsežna in zelo pomembna dovolilnica. Lahko si jo predstavljamo kot več dovolilnic v eni. Kaj nam dovoli, je odvisno od tega, kako jo zgradimo. Njen konstruktor zahteva en parameter in to je target name.

Celoten seznam target imen se nahaja na [5]. Našteli bomo samo najbolj pomembne:

- createClassLoader - dovolimo kreiranja novega nalagalnika razredov
- setSecurityManager - dovolimo nastavljanja novega SecurityManager-ja
- exitVM - dovolimo možnost "ugašanja" aplikacije
- getenv - dovolimo branja okoljskih spremenjivk

Primer uporabe:

```
RuntimePermission rntpermission = new RuntimePermission("
  exitVM")
```

Primer uporabe lahko pogledamo v metodi getenv() iz razreda System:

```
public static String getenv(String name) {
      SecurityManager sm = getSecurityManager();
      if (sm != null) {
          sm.checkPermission(new RuntimePermission("getenv."+
  name));
      }

      return ProcessEnvironment.getenv(name);
 }
```

Ta dovolilnica je zelo pomembna, ker brez nje nihče ne more zamenjati že nastavljenega upravljalca varnosti (angl. SecurityManager) in na ta način spremeniti naša varnostna pravila. Če bi potencialno nevarna koda imela to dovolilnico, bi lahko z lahkoto zamenjala sistemskega upravljalca varnosti s svojim in s tem bi si omogočila popoln dostop do vseh sistemskih virov.

5.3.4 SocketPermission

SocketPermission nam dovoli dostop do omrežja preko vtičnikov (angl. socket). Konstruktor zahteva dva parametra: gostitelj (angl. host), do katerega želimo dostopati, in način dostopa [accept, connect, listen, resolve].

Primer uporabe:

```
new SocketPermission("192.168.1.123:2303", "accept, resolve,
  listen");
```

Brez te dovolilnice koda ne more komunicirati preko omrežja (tukaj je vključen tudi svetovni splet). Danes si zelo težko predstavljamo aplikacijo, ki ne more komunicirati s svetovnim spletom, in zaradi tega je smiselno to

dovolilnico skoraj vedno vključiti. Lahko pa omejimo, do katerih strežnikov ali spletnih mest lahko koda dostopa.

5.4 Dovolilnice v jPluggy

Pri izdelavi ogrodja smo se odločili, da bomo naredili fleksibilen sistem dovolilnic. V ta namen smo razdelili dovolilnice na dva dela. Prvi del je zaznamba (angl. annotation), ki se uporablja za to, da bi označili, da naš vtičnik potrebuje to dovolilnico. Drugi del pa je razred, ki deluje kot povezovalnik med našo zaznambo (angl. annotation) in domorodno (angl. native) Java dovolilnico.

Takoj po nalaganju vtičničnega razreda v virtualni stoj naš upravljalec varnosti pogleda, s katerimi jPluggyPermission zaznambami (angl. annotations) je razred označen. To pomeni, da ta vtičnik potrebuje te dovolilnice. Do tega trenutka so dovolilnice le shranjene v seznamu dovolilnic in so vse označene kot nepregledane. Nato jih je uporabnik dolžan pregledati in jih mora tudi potrditi. Po potrditvi sistem doda dovolilnice v seznam potrjenih dovolilnic za ta vtičnik. Šele takrat lahko sistem naredi instanco vtičnika in uporabnik ga začne uporabljati.

5.4.1 Osnovne jPluggy dovolilnice

Za poenostavitev dela z ogrodjem smo napisali nekaj osnovnih dovolilnic. Vse ostale se lahko napišejo kasneje in dodajo v sistem.

FileIOPermission

Je dovolilnica, ki omogoča dostop do ene same mape na disku. Aplikacija, ki uporablja ogrodje, je dolžna nastaviti mapo, v katero bodo vsi vtičniki shranjevali svoje podatke. V tem direktoriju potem vsak vtičnik, ki ima to dovolilnico, dobi svojo mapo in lahko nad njo izvaja zahtevane operacije.

Definicija zaznambe (angl. annotation) zgleda:

```
public @interface FileIOPermission {
    boolean read() default true;
    boolean write() default false;
    boolean delete() default false;
}
```

Uporablja se na naslednji način:

```
@Plugin("hello")
@FileIOPermission()
public class HelloPlugin {...}
```

Dovolilnica nam omogoča bralno-pisalni dostop in možnost brisanja datotek/map. V ozadju ta dovolilnica uporablja domorodno *FilePermission* dovolilnico.

InternetPermission

Dovolilnica, ki omogoča dostop do spleta. Brez te dovolilnice se vtičnik ne more povezati na splet. Definicija te dovolilnice je zelo enostavna, ker nima nobenih dodatnih parametrov.

```
public @interface InternetPermission {
}
```

Uporablja se na zelo enostaven način:

```
@Plugin("hello")
@InternetPermission()
public class HelloPlugin { ... }
```

V ozadju ta dovolilnica uporablja domorodno *SocketPermission* dovolilnico.

SocketPermission

Je dovolilnica, ki omogoča različne operacije nad eno vtičnico (angl. Socket). Uporaba je razvidna iz definicije dovolilnice:

```
public @interface SocketPermission {
        String ip() default "127.0.0.1";
        int port() default 1234;

        boolean accept() default true;
        boolean connect() default true;
        boolean listen() default true;
        boolean resolve() default true;
}
```

Uporaba te dovolilnice je bolj kompleksna od ostalih, ker ima več parametrov.

```
@Plugin("hello")
@SocketPermission(ip = "10.0.0.1", port = 8080, resolve = true,
    connect = true, accept = true)
public class HelloPlugin { ... }
```

V ozadju ta dovolilnica uporablja domorodno *SocketPermission* dovolilnico.

5.4.2 Razširitev seznama dovolilnic

Ker smo se zavedali, da to niso edine dovolilnice, ki jih realni sistem potrebuje, smo naredili sistem, ki je zelo enostavno razširljiv in hkrati varen za uporabo. “Razširljiv” pomeni, da lahko uporabniki ogrodja dodajajo svoje implementacije dovolilnic. Za varnost smo poskrbeli tako, da smo preprečili vtičnikom dodajanja svojih dovolilnic. Če bi lahko vtičniki dodajali svoje implementacije, bi si dodali vse željene dovolilnice, in s tem bi se izognili peskovniku. Zaradi tega smo naredili sistem tako, da lahko registrirajo nove

dovolilnice le razredi, ki so bili naloženi preko istega nalagalnika razredov kot gostiteljska aplikacija in ogrodje.

Uporabnik, ki želi razširiti sistem in napisati svojo dovolilnico, mora napisati dva razreda. Najprej mora napisati svojo zaznambo (angl. annotation). Primeri teh zaznamb (angl. annotation) so v delu 5.4.1. Drugi razred, ki mora bit napisan, je implementacija vmesnika JPluggyPermission:

```
public interface JPluggyPermission<A> {
    /**
     * Initializes the permission. In this step - the manager
    passes the annotation instance and any needed additional
    parameters.
     *
     * @param annotationInstance
     * @param params
     * @throws PermissionInitializationException
     */
    public void init(A annotationInstance, Object... params)
    throws PermissionInitializationException;

    /**
     * Returns the java permission class.
     *
     * @return
     */
    public Permission getJavaPermission();

    /**
     * Returns the description of this permission.
     *
     * @return
     */
    public String getDescription();

    /**
     * Returns the annotation class that belongs to this
    permission.
```

```
     *
     * @return
     */
    public Class<A> getAnnotationClass();
}
```

Ta implementacija je dolžna povezati zaznambo z domorodno Java dovolilnico. S pomočjo te povezave bo sistem vedel, katero domorodno dovolilnico je vtičnik zahteval. Primer take implementacije je dovolilnica InternetPermission:

```
package si.dime.jpluggy.permissions.implementations;

import si.dime.jpluggy.exceptions.
    PermissionInitializationException;
import si.dime.jpluggy.permissions.JPluggyPermission;
import si.dime.jpluggy.permissions.annotations.
    InternetPermission;

import java.net.SocketPermission;
import java.security.Permission;

/**
 * Created by dime on 3/21/14.
 */
public class InternetPermissionImpl implements JPluggyPermission
    <InternetPermission> {

    @Override
    public void init(InternetPermission annotationInstance,
    Object... params) throws PermissionInitializationException {
        /* Do nothing */
    }

    @Override
    public Permission getJavaPermission() {
        return new SocketPermission("*", "resolve");
```

```
    }

    @Override
    public String getDescription() {
        return "Gives the plugin access to the internet.";
    }

    @Override
    public Class<InternetPermission> getAnnotationClass() {
        return InternetPermission.class;
    }
}
```

Iz tega primera je razvidno, da je *InternetPermission* le lepše ime za *SocketPermission*, ki dovoli operacijo **resolve** za vse IP naslove.

5.5 Upravljalec varnosti v jPluggy

Na začetku poglavja 5 smo povedali, kako lahko začnemo uporabljati obstoječi varnosti mehanizem, ki ga Java platfoma ponuja v naši aplikaciji.

```
System.setSecurityManager(new SecurityManager());
```

S pomočjo zgornje vrstice kode nastavimo upravljalca varnosti na aplikacijskem nivoju. Od te vrstice naprej bo vsak klic, ki želi dostopati do sistemskih virov, preverjen s strani upravljalca varnosti. Če direktno skopiramo zgornjo vrstico v našo aplikacijo, bomo hitro ugotovili, da se bo naša aplikacija ustavila takoj, ko bo poskusila dostopati do nekega sistemskega vira. Vzrok za to je, da koda v Javi privzeto ne vključuje nobenih dovolilnic.

Primer neuporabne, a zelo varne aplikacije:

```
System.setSecurityManager(new SecurityManager());
File file = new File("/tmp");
```

Zgornja koda bo sprožila *RuntimeException* in naša aplikacija bo nehala delovati.

Ta problem reši naša implementacija upravljalca varnosti, ki deduje od privzete implementacije in na novo implementira samo en del starševskega razreda.

Glavni nalogi tega upravljalca sta:

1. Da dovoli popoln dostop do vseh sistemskih virov, vseh razredov starševske aplikacije

2. Da omeji dostop do sistemskih virov vtičnikov

Za prvo nalogo smo se odločili, da bomo uporabili pristop s takoimenovanim belim seznamom (angl. white list). Beli seznam v našem primeru pomeni seznam nalagalnikov razredov, katerim lahko zaupamo. Pri inicializaciji upravljalnik varnosti doda svoj nalagalnik razredov in vse njegove starše v beli seznam. Praviloma bomo s tem pokrili vse nalagalnike, ki so bili uporabljeni pri nalaganju razredov ogrodja in starševske aplikacije.

Pri procesu preverjanja dovolilnic najprej preverimo, če je nalagalnik razredov določenega razreda na belem seznamu - če je, takoj omogočimo dostop. Povedano z drugimi besedami, to pomeni, da bodo vsi razredi ogrodja in vsi razredi starševske aplikacije dobili vse dovolilnice oz. bodo imeli neomejeni dostop do sistemskih virov.

Takoj po nalaganju razredov vtičnika nalagalnik razredov prebere vse varnostne zaznambe (angl. annotation) glavnega razreda tega vtičnika in seznam dovolilnic posreduje našemu upravljalcu varnosti. Upravljalec varnosti nastavi dane dovolilnice temu vtičniku tako, da pokliče ustrezno metodo starševskega razreda.

Kasneje, ko uporabniška aplikacija pokliče metodo tega vtičnika, sam varnostni sistem Java platforme pokliče upravljalca varnosti, ki je nastavljen na nivoju aplikacije, da preveri celotno hierarhijo razredov, če imajo ustrezne dovolilnice. Naša implementacija v tem primeru samo posreduje klic starševskemu razredu.

Na ta način rešimo drugo nalogo našega upravljalca varnosti. Vse ostalo prepustimo že preverjenemu sistemu varnosti Java platforme. In na ta način zmanjšamo verjetnost napake.

Poglavje 6

Primer uporabe ogrodja jPluggy

Za boljše razumevanje smo se odločili, da si bomo bolj podrobno ogledali en enostaven primer uporabe ogrdoja jPluggy.

Kot primer bomo uporabili spletno aplikacijo, ki uporablja vtičnike za zagotavljanje različnih metod za preverjanje verodostojnosti uporabnika. Zaradi poenostavitev primera, bomo obravnavali le dva vtičnika. Prvi vtičnik bo imel vse podatke o uporabnikih shranjene v kodi, drugi pa bo vse te podatke prebral iz datoteke.

6.1 Definicija vmesnikov

Kot smo omenili v poglavju 2.3, mora vsaka gostiteljska aplikacija omogočati določene storitve (angl. Services) preko vmesnikov. V našem primeru mora gostiteljska aplikacija omogočiti pridobitev informacije o poti, kamor lahko vtičniki shranijo svoje datoteke. To informacijo bo potreboval vtičnik, ki bo bral podatke o uporabnikih iz datoteke. V ta namem napišemo vmesnik, ki omogoča to storitev.

```
public interface PluginContext {
```

```
    /**
     * Returns the path of the dir where the plugin can store it
    's data (if such permission is asked from the plugin).
     * If no such permission is asked - null is returned.
     *
     * @return
     */
    public String getStoragePath();
}
```

V istem poglavju (2.3) smo povedali, da gostiteljska aplikacija in vtičniki komunicirajo preko vtičniških vmesnikov. Naša aplikacija potrebuje vtičnike le za preverjanje verodostojnosti uporabnika. Zato potrebujemo le en vtičniški vmesnik.

```
public interface AuthenticationPlugin {
    /**
     * Initializes the plugin. The context is passed as
    parameter.
     *
     * @param context
     */
    public void init(PluginContext context);

    /**
     * Returns an User object if exists for the given username.
    Otherwise returns null.
     *
     * @param username
     * @return
     *       the user object for the given username.
     */
    public UserInfo findUser(String username);
}
```

Vmesnik je sestavljen iz dveh metod. Prva metoda služi kot vstopna točka in preko nje vtičnik dobi dostop do storitev, ki jih ponuja gostiteljska aplikacija. Druga metoda je storitev, ki jo je vtičnik ponuja. Gostiteljska aplikacija pokliče to metodo, da dobi informacijo, če je uporabnik verodostojen.

S tem smo zaključili definicijo vmesnikov in lahko nadaljujemo z razvojem vtičnikov.

6.2 Razvoj vtičnikov

Razvili bomo dva vtičnika. Prvi je zelo enostaven vtičnik, ki bere informacije o uporabnikih iz kode in zaradi tega ne potrebuje nobenih dovolilnic. Je enorazredni vtičnik, ki implementira metode vmesnika *AuthenticationPlugin*.

```
@Plugin("SimpleAuthenticator")
public class SimpleAuthenticator implements AuthenticationPlugin
    {
    /* The context */
    private PluginContext context;

    /* The user informations */
    private Map<String, String> users = new HashMap<String,
   String>();

    public SimpleAuthenticator() {
      /* Add a user 'root' with password 'root' */
      users.put("root", "root");
    }

    @Override
    public void init(PluginContext context) {
        this.context = context;
    }

    @Override
    public UserInfo findUser(String username) {
```

```
    /* Check if the user exists */
    if (users.containsKey(username)) {
      return new UserInfo(username, new SimpleUserCredentials(
   users.get(username)), Arrays.asList(new String[]{
   AuthenticationUserRoles.USER.name().toLowerCase()}));
    }

    /* No such user found */
    return null;
    }
}
```

Drugi vtičnik je razširitev prvega. Edina razlika je v tem, kako dobi informacije o uporabnikih (metoda *init(PluginContext context)*). Ta vtičnik prebere informacije iz določene datoteke. Zaradi tega potrebuje dovolilnico za branje iz datotečnega sistema. Lokacija datoteke je določena s strani gostiteljske aplikacije, ker ima vtičnik dovolilnico brati le določeno mapo na datotečnem sistemu (glej 5.4.1).

```
@Plugin("FileBasedAuthenticator")
@FileIOPermission(read = true, write = false, delete = false)
public class FileBasedAuthenticator implements
   AuthenticationPlugin {
    /* The context */
    private PluginContext context;

    private Map<String, String> users = new HashMap<String,
   String>();

    @Override
    public void init(PluginContext context) {
        this.context = context;

        File inputFile = new File(context.getStoragePath() + "/
   users.txt");
```

```
        try {
            BufferedReader br = new BufferedReader(new
    FileReader(inputFile));
            String line;
            while ((line = br.readLine()) != null) {
                String[] split = line.split(";");
                users.put(split[0], split[1]);
            }
            br.close();
        } catch (IOException e) {
            System.err.println("Error while reading the input
    file. No users available.");
        }
    }

    @Override
    public UserInfo findUser(String username) {

        /* Try to find the user */
        if (users.containsKey(username)) {
            return new UserInfo(username, new
    SimpleUserCredentials(users.get(username)), Arrays.asList(new
     String[]{AuthenticationUserRoles.USER.name().toLowerCase()})
    );
        }

        /* No such user found */
        return null;
    }
}
```

6.3 Uporaba vtičnikov

Prvi korak, ki ga moramo narediti v gostiteljski aplikaciji, je inicializacija ogrodja jPluggy. Pri inicializaciji je nujno poslati dva parametra, pot do

mape, kjer so shranjeni vsi vtičniki, in pot do mape, kamor lahko vtičniki shranjujejo svoje datoteke. Pri tem shranimo referenco na ogrodje.

```
/* Initialize the framework & load the plugins */
  jPluggy = new JPluggyManager("/var/myserver/plugins", "/var/
 myserver/plugins_storage");
  jPluggy.loadPlugins();
```

Naslednji korak je preverjanje in odobritev dovolilnic vtičnikov. Ogrodje ne shranjuje ničesar, ker je povezano z preverjanjem in odobritvijo dovolilnic, in zaradi tega mora za to poskrbeti gostiteljska aplikacija in eksplicitno odobriti dovolilnice po vsaki inicializaciji ogrodja.

```
/* Get the classes of the AuthenticationPlugin implementations
   */
Set<String> pluginClasses = jPluggy.getLoadedPluginNames(
 AuthenticationPlugin.class);

/* Check the permission and the approval of all of them */
for (String className : pluginClasses) {

  /* Get the permissions */
  List<JPluggyPermission> permissions = jPluggy.
 getSecurityManager().getPermissionsForPlugin(className);

  /* Check if the permisisons set is the same as the last time
  we checked it */
  if (permissionSetIsSame(className, permissions)) {

    /* If we are here, that means the user already approved
 the permissions so it's save to mark them as approved */
    jPluggy.getSecurityManager().approvePlugin(className);
  } else {
    /* Display some info the the user and prompt him to check
 the permissions */
  }
```

```
}
```

Metoda *permissionSetIsSame(className, permissions)* preveri v svoji bazi podatkov, ali je uporabnik že odobril dovolilnice za podani vtičnik in, če se te dovolilnice ujemajo.

Zadnji korak je pridobitev instance vtičnikov.

```
List<AuthenticationPlugin> plugins = jPluggy.getPluginsOfType(
  AuthenticationPlugin.class);

/* Get the currently active authenticator */
AuthenticationPlugin provider = getActiveAuthenticator(plugins
  );

/* Build the plugin context */
PluginContext context = new PluginContextImpl(jPluggy.
  getSecurityManager().getStorageDirForPlugin(provider.getClass
  ()));

/* Initialize the plugin */
provider.init(context);
```

Metoda *getActiveAuthenticator(plugins)* preveri v bazi podatkov, katera metoda za preverjanje verodostojnosti uporabnikov je trenutno aktivna, in vrne ustrezni vtičnik za to metodo.

Referenca vtičnika je zdaj shranjena v spremenljivki *provider*, in vtičnik lahko uporabljamo kot vsak navadni Java razred.

```
/* Try to authenticate a user */
String user = getUserInput("user");
String pass = getUserInput("password");

UserInfo foundUser = provider.findUser(user);
if (foundUser != null) {

  /* Check the password */
```

```
    if (foundUser.checkCredential(pass)) {

      /* Authentication successful */
      System.out.println("Authentication successful!");
    } else {

      System.err.println("User authentication failed!");
      /* Display an error message to the user */
    }
  } else {

    System.err.println("User authentication failed!");
    /* Display an error message to the user */
  }
```

Poglavje 7

Zaključek

V poslovnem IT svetu je poleg kakovosti programske opreme glavna skrb cena izdelave le-te, cena izdelave morebitnih razširitev in vzdrževanje. Prav zaradi tega se večina razvijalcev pri izdelavi arhitekture aplikacije odloča za vtičniško arhitekturo, ki omogoča enostavno dinamično razširljivost aplikacije.

V okviru diplomske naloge smo najprej preučili že obstoječa ogrodja za delo z vtičniki. Po ugotovitvi, da v tem trenutku obstaja le eno tako napredno ogrodje, ki pa je za večino uporabnikov preveč kompleksno in se zato odloča za razvoj lastnega, pri čemer pa se večinoma ne upošteva nobenih varnostnih pravil, smo se odločili za razvoj enostavnega ogrodja s poudarkom na varnosti.

Najbolj pomembna naloga, ki smo si jo zadali pri izdelavi ogrodja, je bila varnost pri izvajanju kode vtičnikov. Pri tem pa nam je uspelo doseči tudi, da naše ogrodje na noben način ne omejuje razvijalcev vtičnikov in uporabnikov, ker ne uporablja svojih pravil, proces razvoja pa ostane isti. Razvijalci bodo novo razvite vtičnike videli kot vsako navadno Java knjižnico s par dodatnimi zaznambami.

V bodoče bi bilo zanimivo pogledati, kako se ogrodje obnaša v produkcijskem okolju. Prav tako bi bilo zanimivo opazovati, v katero smer se bo ogrodje razvijalo in kakšne nove fukncionalnosti bi bilo treba razviti za po-

trebe realnega sistema.

Literatura

[1] Horstmann, Cay S. & Cornell, Gary, "Core Java Volume II 9E", poglavje 9, marec 2013.

[2] Core Java security: Class Loaders, Security Managers, and Encryption, http://www.informit.com/articles/article.aspx?p=1187967&seqNum=3, april 2008.

[3] Software framework, http://en.wikipedia.org/wiki/Software_framework, julij 2014.

[4] Software development process, http://en.wikipedia.org/wiki/Software_development_process, avgust 2014.

[5] Permissions in Java, http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html, avgust 2014.

[6] OSGi http://searchnetworking.techtarget.com/definition/OSGi, avgust 2014.

[7] Plug-in (computing) http://en.wikipedia.org/wiki/Plug-in_(computing), julij 2014.

[8] Java - Watching a Directory for Changes http://docs.oracle.com/javase/tutorial/essential/io/notification.html, avgust 2014.

[9] OSGi? No Thanks http://blogs.mulesoft.org/osgi-no-thanks/, november 2009