Alma Mater Studiorum - Università di Bologna

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Informatica

Attività progettuale di Linguaggi e Modelli Computazionali M

Gestione del Class Loading di tuProlog

Candidato:
Michele Mannino

Professore:
Enrico Denti

Sommario

1. Introduzione

L'obiettivo principale dell'attività progettuale è quello di risolvere il problema di tuProlog riguardo il caricamento di librerie addizionali e oggetti Java utilizzati tramite `JavaLibrary`. Le librerie possono caricate (scaricate) dinamicamente nell'engine sia lato Java tramite il metodo `loadLibrary` (`unloadLibrary`) sia lato Prolog utilizzando il predicato `load_library/2` (`unload_library/2`).

Il problema è relativo al caricamento delle librerie lato Prolog: lanciando il GUI tuProlog (o la console CUI) con il doppio-click sul file 2p.jar, le librerie necessarie, essendo fuori dal JAR, non vengono caricate come desiderato. Questo è dovuto al comportamento del class loader che non permette il caricamento di classi che sono fuori dal JAR dell'applicazione stessa. Per evitare questo problema, si utilizza il seguente exploit per caricare il JAR come una libreria e non come un eseguibile:

```
java -cp MyLibrary.jar:2p.jar alice.tuprologx.ide.GUILauncher
```

```
java -cp MyLibrary.jar:2p.jar alice.tuprologx.ide.CUIConsole
```

In questo modo, l'opzione `-cp` permette al class loader di fare il possibile per caricare le librerie (in questo caso `MyLibrary`). Invece, se viene eseguito il JAR direttamente (`java -jar`), la clausola `-cp` viene ignorata portando al fallimento a runtime.

Per risolvere il problema appena visto si è deciso di modificare il class loading in modo tale da caricare le librerie fuori dal path locale all'applicazione tuProlog. Questo è stato possibile tramite utilizzo di `java.net.URLClassLoader`.

Una soluzione alternativa che prende spunto dall'URLClassLoader è stata adottata per modificare predicati esistenti come `java_object`, `java_call` ed introdurre nuovi predicati come `get/set_classpath`. E' stato implementato un class loader custom: `alice.util.DynamicURLClassLoader` (paragrafo 3.1).

Nei prossimi capitoli vengono analizzate le problematiche affrontate e le scelte progettuali adottate per risolvere quest'ultime.

2. Caricamento delle library

2.1. URLClassLoader

Per risolvere il problema del caricamento delle librerie è stata utilizzata la classe `java.net.URLClassLoader`. Questa permette di caricare classi sia da direttori che da file JAR. Ogni URL che finisce con un '/' fa riferimento ad un direttorio. Altrimenti, si assume che l'URL faccia riferimento ad un file JAR che verrà aperto quando necessario.

Il costruttore di cui sotto permette di creare un'istanza del loader che fa riferimento ad un array di `java.net.URL` ed utilizza per delega il costruttore padre in modo tale da verificare prima se esso abbia già caricato la classe desiderata.

```
public URLClassLoader(URL[] urls, ClassLoader parent)
```

Tramite il metodo di `java.lang.Class` di cui sotto è possibile utilizzare un'istanza dell'`URLClassLoader` per reperire l'istanza della classe desiderata.

```
public static Class forName(String name, boolean initialize, ClassLoader
loader)  throws ClassNotFoundException
```

2.2. `loadLibrary` e `load_library/2`

L'utilizzo dell'URLClassLoader è stato fondamentale per permettere il caricamento delle library sia lato Java (`loadLibrary`) che lato Prolog (`load_library/2`).

Nel primo caso è stato aggiunto un nuovo metodo ad `alice.tuprolog.LibraryManager` come segue:

```
public synchronized Library loadLibrary(String className, String[] paths)
throws InvalidLibraryException
```

Parametri:

> `className` – il nome della classe desiderata.
>
> `paths` – array di stringhe contenenti gli URL in cui l'`URLClassLoader` cercherà la classe desiderata.

Ritorna:

> l'oggetto `Class` per la classe specificata dal nome.

Lancia: `InvalidLibraryException` – Libreria desiderata non trovata.

Lato prolog è stato implementato il predicato `load_library/2` che ha accetta come il nome della libreria che si vuole caricare e la lista di URL a cui si vuole cercare. Il predicato utilizza il metodo del `LibraryManager.loadLibrary` una volta che ha verificato che i parametri siano conformi alle specifiche.

```
load_library(ClassName, [PathList])
```

Esempio di utilizzo di `load_library/2`:

```
demo_1(X,Y,R) :- load_library('TestLibrary',
      ['C:\', 'C:\Users\MyLibrary.jar']),
      R is sum(X, Y).

? - demo_1(3, 4, Res).
yes.
Res / 7.0
Solution: demo(3,4,7.0)
```

Es. 1

La libreria `TestLibrary` viene cercata nella lista di path passata al predicato, poi per verificarne il corretto caricamento si utilizza il predicato `sum/2` contenuto nella libreria stessa.

2.3. Modifica dell'IDE tuProlog

Una aggiunte le nuove funzionalità di class loading, è stato modificato l'IDE (Fig. 1) per permettere all'utente di caricare le proprie librerie in modo tale da poterle utilizzare.

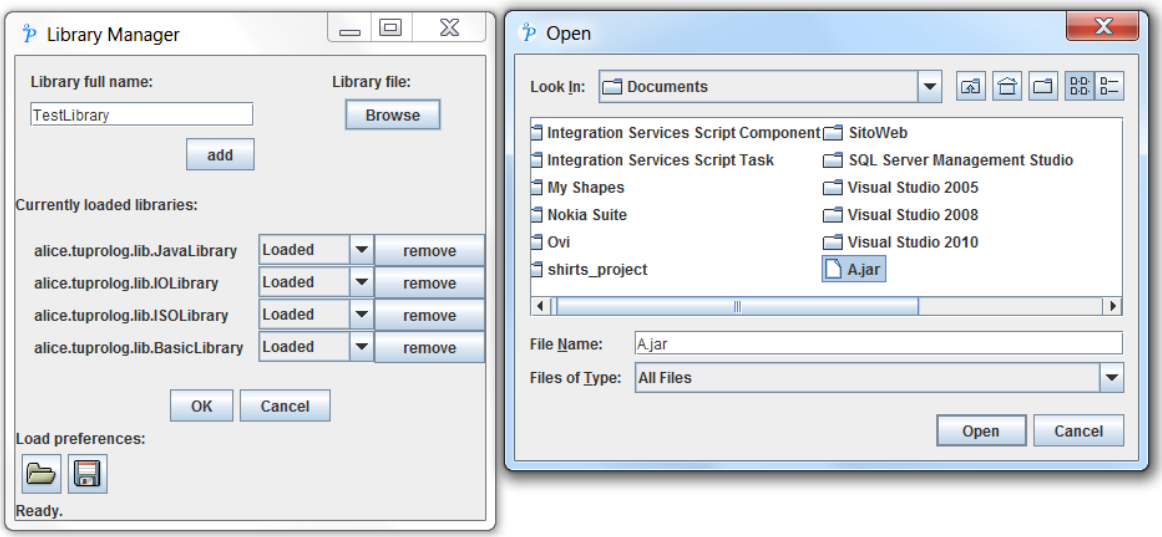

Fig. 1, tuProlog IDE con `javax.swing.JFileChooser`

E’ possibile caricare le librerie in due modi diversi:

- Utilizzando il pulsante *add* viene caricata la libreria, il cui nome è inserito nel TextField relativo, se essa è presente le path locale a 2p.jar;
- Utilizzando il pulsante *Browse* è possibile selezionare il jar in cui è presente la classe identificata dal nome sempre inserito nel TextField relativo.

Nel caso in cui il nome della libreria è invalido o non è contenuta nel jar allora viene restituito un warning sull’IDE come segue in figura (Fig. 2):

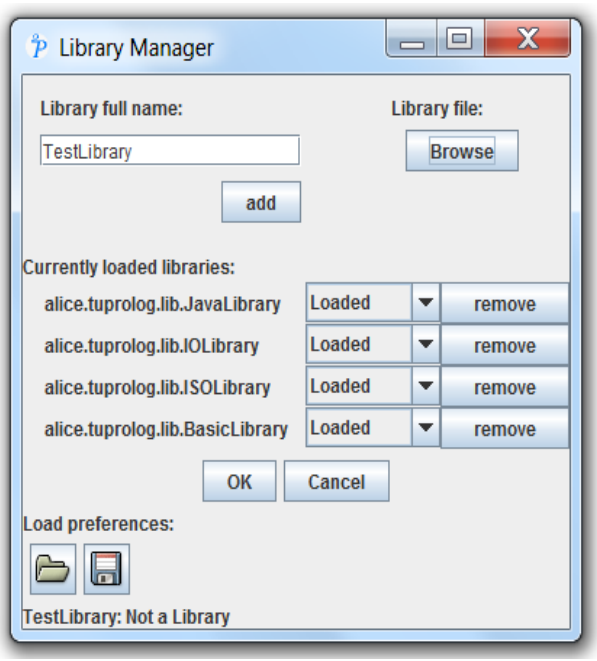

Fig. 3, Libreria non valida

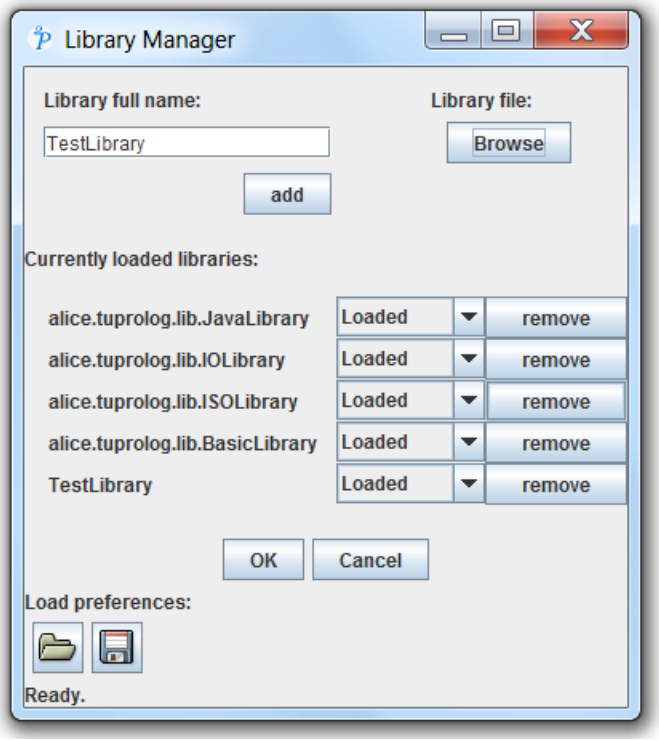

Fig. 2, Libreria valida

Altrimenti (Fig. 3) la libreria viene visualizzata tra quelle correnti caricate.

3. Estensione predicati tuProlog

Una volta assodate le capacità dell'URLClassLoader, sono stati inseriti nuovi predicati ed estesi quelli già esistenti nella JavaLibrary. Questi sono:

1. `java_object/4` – aggiunge all'esistente `java_object/3` il parametro della lista dei path a cui cercare l'oggetto desiderato.
2. Estensione del predicato `java_call/3` che permette l'accesso ai metodi e campi di oggetti caricati dall'utente da path diversi da quello locale a 2p.jar.
3. Modifica dei predicati già esistenti nella JavaLibrary (`java_set/get`, `java_object/3`,…) dovuto alla tecnica sostitutiva di class loading.

3.1. DynamicURLClassLoader

A differenza del caricamento delle library (`load_library/2`) che viene effettuato tramite `java.net.URLClassLoader`, il caricamento degli oggetti nella JavaLibrary necessitano di una soluzione alternativa. Il motivo principe è relativo ai vincoli di class loading di oggetti della stessa classe caricati da diversi class loader, quindi facenti parte a name space differenti.

Dati due oggetti dello stessa classe che vengono caricati da due ClassLoader differenti, se si effettua il cast tra i due allora la JVM lancia l'eccezione `java.lang.ClassCastException`. Questo perché l'oggetto per la JVM è identificato sia dalla propria classe che dal ClassLoader che l'ha caricato, si dice che fanno parte di due NameSpace diversi.

Il problema appena analizzato si è presentato nel caso seguente:

```
demo(Res) :- java_object('Counter', [], MyCounter, ['C:\Users\A.jar']),
      java_object('Counter[]', [10], ArrayCounters, ['C:\Users\A.jar']),
      MyCounter <- inc,
      java_array_set(ArrayCounters, 0, MyCounter),
      java_array_get(ArrayCounters, 0, C),
      C <- getValue returns Res.
```

Es. 2

Il predicato `java_array_set` cerca di inserire l'oggetto `MyCounter` all'interno di `ArrayCounters`, però la JVM verifica che i due oggetti hanno sì la stessa classe, ma sono caricati da URLClassLoader differenti. Quindi il risultato è che viene scatenata l'eccezione `ClassCastException`.

Quindi è stato implementato il `alice.util.DynamicURLClassLoader` (appendice [A1]). E' una versione rivisitata del `java.net.URLClassLoader` che permette anche di gestire in

maniera ottimizzata l'array di URL cui esso fa riferimento con l'obiettivo di ridurre l'utilizzo di risorse. Con questo si intende che l'utilizzatore può aggiungere/togliere URL (vedi appendice [A1]) a proprio piacimento, cosa che non è possibile nel caso del `java.net.URLClassLoader` che permette solamente di aggiungere URL. Segue un esempio di utilizzo:

```
DynamicURLClassLoader classLoader = new DynamicURLClassLoader(
new URL[]{},
this.getClass().getClassLoader());
// Update the list of paths of the DynamicURLClassLoader
classLoader.addURLs(urls);
…
// Reset the DynamicURLClassLoader at default configuration
classLoader.removeURLs(urls);
```

Es. 3

Come si nota dall'esempio sopra, è possibile aggiungere o rimuovere URL a cui il class loader verificherà la presenza della classe desiderata con l'obiettivo di evitare perdite di performance. E' stato utilizzato nell'ambito del caricamento di oggetti nella JavaLibrary (vedi predicato `java_object/4`) in modo tale che, una volta caricato un oggetto desiderato, l'URL utilizzato venga eliminato dalla lista in cui il loader, durante il suo funzionamento, andrà a cercare altri oggetti.

In definitiva la JavaLibrary ha un riferimento univoco al `DynamicURLClassLoader` e carica le classi a runtime tramite:

```
Class<?> cl = Class.forName(clName, true, classLoader);
```

dove classLoader è un'istanza del `DynamicURLClassLoader` generata dal costruttore della `JavaLibrary`.

3.2. Predicato `java_object/4`

Il predicato java_object/4 permette di caricare oggetti nella JavaLibrary specificando la lista di path a cui si fa riferimento:

```
java_object(ClassName, ArgList, ObjRef, [PathList])
```

In sostanza, il predicato controlla che il `PathList` sia una lista e poi delega il caricamento al predicato `java_object/3`. Segue una parte del codice implementativo:

```
if(!paths.isList())
  throw new IllegalArgumentException();

String[] listOfPaths = getStringArrayFromStruct((Struct) paths);
URL[] urls = getURLsFromStringArray(listOfPaths);

// Update the list of paths of the URLClassLoader
  classLoader.addURLs(urls);

// Delegation to java_object_3 method used to load the class
boolean result = java_object_3(className, argl, id);

// Reset the URLClassLoader at default configuration
classLoader.removeURLs(urls);

return result;
```

Es. 4

Esempio di utilizzo:

```
?- java_object('Counter', [], Obj,['C:\Users\A.jar']),
  Obj <- inc,
  Obj <- inc,
  Obj <- getValue returns Val.

Obj / '$obj_0'
Val / 2

yes.
```

Es. 5

3.3. Estensione del predicato `java_call/3`

Il predicato `java_call/3` è stato esteso aggiungendo anche il parametro opzionale relativo al path, in modo tale da chiamare metodi (ed accedere in set/get campi statici) di oggetti caricati sia dall'utente sia già presenti nella JavaLibrary. In questo caso il predicato riconosce se si vuole fare una call di un oggetto già caricato o di uno che è stato caricato dall'utente da uno specifico path. Quindi `java_call/3` riconosce:

- `(<className>) <- <methodName>`
- `([<pathList>], <className>) <- <methodName>`

Esempio di accesso ad un metodo:

```
?- class(['C:\ ', 'C:\Users\A.jar'], 'TestStaticClass') <-
echo('Message') returns Val.

Val / 'Message'

yes.
```

Es. 6

Esempio di accesso ad un campo statico:

```
?- class(['C:\', 'C:\Users\A.jar'], 'TestStaticClass').'id' <- get(Value)
returns Val.

Value / 0

yes.
```

Es. 7

Il predicato `java_call/3` viene indirettamente chiamato dalla JavaLibrary che permette sia l'accesso ad un campo statico che la chiamata ai metodi. Nel caso di accesso a campi statici, la JavaLibrary utilizza i metodi di `java_get/set` i quali anch'essi utilizzano il `DynamicURLClassLoader`.

Con questo si intende che è stato sostituito:

```
Class<?> cl = Class.forName(className);
```

con

```
Class<?> cl = Class.forName(className, true, classLoader);
```

3.4. Predicati `java_array_*`

I predicati per la gestione degli array, come già detto, permettono anch'essi l'utilizzo di oggetti java aggiunti alla JavaLibrary dall'utente. Eccone un esempio:

```
?- java_object('Counter', [], MyCounter, ['C:\Users\A.jar']),
java_object('Counter[]', [10], ArrayCounters, ['C:\Users\A.jar']),
java_array_length(ArrayCounters, Size).

MyCounter / '$obj_0'
ArrayCounters / '$obj_1'
Size / 10

yes.
```

Es. 8

3.5. Predicato `set/get_classpath`

Per evitare di riscrivere ogni volta i path nei predicati di `java_object`, ecc… è stato aggiunto il predicato di `set_classpath` e il relativo `get_classpath` in modo tale da aggiungere al DynamicURLClassLoader i path in cui cercare e conoscere a runtime il loro valore. Segue un esempio:

```
demo(List) :- set_classpath(['C:\', 'C:\Users\A.jar']),
              get_classpath(List).
```

```
?- demo(L).
yes.
L / ['C:\', 'C:\Users\A.jar']
```

Es. 9

3.6. Predicati `register/1` e `unregister/1`

Il tempo di vita del binding tra oggetti Java e termini Prolog è quello della query corrente. Per mantenere il binding in vita anche dopo la query, è stato aggiunto il predicato `register/1` (`unregister/1`) per permettere la registrazione (cancellazione) dagli oggetti statici. Segue un esempio:

```
demo(Obj) :- java_object('Counter', [], Obj, ['C:\Users\A.jar']),
   Obj <- inc,
   Obj <- inc,
   register(Obj).

demo2(Obj, Val) :- Obj <- inc, Obj <- getValue returns Val.

?- demo(Obj).
yes.
Obj / '$obj_0'

?- demo2(Obj, Val).
yes.
Val / 3.
```

Es. 10

4. tuProlog in .NET

L'URLClassLoader che viene tradotto da ikvm per la versione .NET di tuProlog non riesce a caricare assembly da path diversi da quello locale. Questo è invece effettuato dal class loader di sistema lo che lo stesso sviluppatore di ikvm, Jeroen Frijters, asserisce essere magicamente effettuato tramite un 'hack':

"Class.forName() has a special hack to allow loading fully qualified .NET types. This has nothing to do with class loaders." - Jeroen Frijters.

Per affrontare il problema è stato necessario usare le API di .NET per caricare le .dll da path diversi da quello locale, purtroppo solamente per quanto riguarda l'IDE. Quindi tutti i predicati introdotti in JAVA (`get/set_classpath`, `java_object_4`, ecc…) non possono essere utilizzati lato .NET perché manca la possibilità di gestire in maniera custom, come avviene per il `DynamicURLClassLoader`, il loading delle dll.

Ecco il workflow che è stato adottato:

1. `ikvmstub mscorlib` : Permette di generare, tramite lo stub di ikvm, il file `mscorlib.jar`.
2. Aggiungere al classpath:
   a. `mscorlib.jar`
   b. `ikvm-api.jar` – utilizzata per accedere al class loader `ikvm.runtime.AssemblyClassLoader`
3. Utilizzo `ikvm.runtime.AssemblyClassLoader` nel seguente modo:

```
// Caricamento della dll
Assembly asm = Assembly.LoadFrom(file.getPath());

// Reperimento istanza AssemblyCustomClassLoader
loader = new AssemblyCustomClassLoader(asm, new URL[]{url});

// Formato: cli.<namespace>.<className>
libraryClassname = "cli." + libraryClassname.substring(0,
      libraryClassname.indexOf(",")).trim();

// Caricamento della classe relative alla dll desiderata
lib = (Library) Class.forName(libraryClassname, true,
      loader).newInstance();

libraries.add(lib.getName());
```

Es. 11

Segue l'esempio di caricamento di una dll.

```
demo(X, Y, Res) :- load_library('TestLibraryCS.TestLibraryCS,
      TestLibraryCS',['C:\Users\TestLibraryCS.dll']),
      Res is sum(X, Y).

?- demo(4, 7, R).
yes.
R / 11.0
```

Es. 12

4.1. Differenze implementative tra Java e .NET di tuProlog

La seguente tabella mostra le differenze implementative tra la soluzione tuProlog per Java e quella contrapposta .NET:

	Java	**.NET**
Loading librerie dall'IDE	SI	NO
load_library/2	SI	SI
java_object/4	SI	NO
java_call_3 con path	SI	NO
get/set_classpath/1	SI	NO
java_object per array con path	SI	NO

Dato il minimo impatto dell'utilizzo dei tipi .NET sul codice Java si può accettare la convivenza dei due linguaggi. La parte relativa ai predicati che sono stati aggiunti/modificati lato Java può essere implementata direttamente nella libreria .NET `OOLibrary`.

Appendice

[A1] – Classe `alice.util.DynamicURLClassLoader.java`

```
public class DynamicURLClassLoader extends ClassLoader{
  private ArrayList<URL> listURLs = null;
  private Hashtable<String, Class<?>> classCache = new Hashtable<String,
    Class<?>>();

  public DynamicURLClassLoader()
  {
    super(DynamicURLClassLoader.class.getClassLoader());
    listURLs = new ArrayList<URL>();
  }

  public DynamicURLClassLoader(URL[] urls)
  {
    super(DynamicURLClassLoader.class.getClassLoader());
    listURLs = new ArrayList<URL>(Arrays.asList(urls));
  }

  public DynamicURLClassLoader(URL[] urls, ClassLoader parent)
  {
    super(parent);
    listURLs = new ArrayList<URL>(Arrays.asList(urls));
  }
  public Class<?> loadClass(String className) throws
    ClassNotFoundException
  {
    return findClass(className);
  }



/**
* Find class method specified by className parameter.
* @param className - The class name used to find the class needed.
*/

public Class<?> findClass(String className) throws ClassNotFoundException
{
  Class<?> result = null;
  String classNameReplaced = className.replace(".", File.separator);

  result = (Class<?>) classCache.get(className);
  if (result != null)
    return result;
  try {
    return findSystemClass(className);
  } catch (ClassNotFoundException e) {
  }
  for (URL aURL : listURLs) {
    try {
```

```
      InputStream is = null;
      byte[] classByte = null;

      if(aURL.toString().endsWith(".jar"))
      {
        aURL = new URL("jar", "", aURL + "!/" +
            classNameReplaced + ".class");
        is = aURL.openConnection().getInputStream();
      }

      if(aURL.toString().indexOf("/",
            aURL.toString().length() - 1) != -1)
      {
        aURL = new URL(aURL.toString() + classNameReplaced + ".class");
        is = aURL.openConnection().getInputStream();
      }

      classByte = getClassData(is);
      try {
        result = defineClass(className, classByte, 0,
                 classByte.length, null);

         classCache.put(className, result);

      } catch (SecurityException e) {
        result = super.loadClass(className);
      }
      return result;
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  throw new ClassNotFoundException(className);
}

/**
* Get data of the class to be loaded.
* @param is - InputStream
*/

private byte[] getClassData(InputStream is) throws IOException
{
  ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
  int nextValue= is.read();
  while (-1 != nextValue) {
  byteStream.write(nextValue);
  nextValue = is.read();
  }
  is.close();
  return byteStream.toByteArray();
}

/**
* Add array URLs method.
```

```
* @param urls - URLs array.
*/
public void addURLs(URL[] urls) throws MalformedURLException
{
  if(urls == null)
    throw new IllegalArgumentException("Array URLs must not be null.");
    for (URL url : urls) {
      if(!listURLs.contains(url))
        listURLs.add(url);
    }
}

/**
* Remove array URLs method.
* @param urls - URL to be removed.
*/
public void removeURL(URL url) throws IllegalArgumentException
{
  if(!listURLs.contains(url))
    throw new IllegalArgumentException("URL: " + url + "not found.");
  listURLs.remove(url);
}

/**
* Remove all URLs contained into URLs array param.
*
* @param urls - Array urls to be deleted.
*/
public void removeURLs(URL[] urls) throws IllegalArgumentException
{
  if(urls == null)
    throw new IllegalArgumentException("Array URLs must not be null.");
  for (URL url : urls) {
    if(!listURLs.contains(url))
      throw new IllegalArgumentException("URL: " + url + "not found.");
    listURLs.remove(url);
  }
}

/**
* Remove all URLs cached.
*/
public void removeAllURLs()
{
  if(!listURLs.isEmpty())
    listURLs.clear();
}

/**
* Get all URLs cached.
*/
public URL[] getURLs()
{
  URL[] result = new URL[listURLs.size()];
```

```
  listURLs.toArray(result);
  return result;
}

/**
* Get all loaded class stored into the class cache.
*/
public Class<?>[] getLoadedClasses()
{
  Class<?>[] result = new Class<?>[classCache.size()];
  int i = 0;
  for (Class<?> aClass : classCache.values()) {
    result[i] = aClass;
  }
  return result;
}

/**
* Clear all class cached.
*/
public void clearCache()
{
  classCache.clear();
}

/**
* Remove a Class from the cache named className.
* It does not unload the class, but it only remove it from the cache.
* @param className - Class name.
*/
public void removeClassCacheEntry(String className)
{
  classCache.remove(className);

}

/**
* Add class into cache.
* @param cls - Class instance.
*/

public void setClassCacheEntry(Class<?> cls)
{
  if(classCache.contains(cls))
    classCache.remove(cls.getName());
    classCache.put(cls.getName(), cls);
  }
}
```