JMLS – Multi Language System for Java

Informatik Projekt

in der Software Composition Group SCG
am Institut für Informatik und angewandte Mathematik IAM
der Univerität Bern

Author:
John M. Hutchison

Juni 2001

Betreut durch:

Prof. Oscar Nierstrasz
Inst. für Informatik (IAM)
Universität Bern
Neubrückstr. 10
3012 Bern

Benno Burkhardt
TeTrade AG für Informatik
Bahnhofstr. 4
3073 Gümligen

Einführung

Motivation

Die Firma TeTrade AG für Informatik in Bern vertreibt ein Produkt namens WMLS oder das Multi Language System for Windows. WMLS bietet die Möglichkeit ein Programm auf der Microsoft Win32 Plattform zu übersetzen, ohne das Programm zu verändern. Dies bedeutet, dass alle auf dem Bildschirm angezeigten Texte in eine andere Sprache als ursprünglich vorgesehen. Zudem hat man mit die Möglichkeit Texte aus einem Programm auszulesen, um die Übersetzung zu vereinfachen.

Um das Geschäftsbereich der Übersetzungssysteme zu erweitern, wollte die TeTrade die Möglichkeit eines WMLS ähnlichen Systems erörtern, welche aber Java Programme übersetzen könnte. Diesem Projekt wurde der Name JMLS, Multi Language System for Java, gegeben.

Zielsetzung

Ziel dieses Projektes war es ein Prototyp für JMLS zu erstellen, das alle Hauptprobleme der Übersetzung und der Erfassung von Texten, ohne die übersetzte Java Applikation zu verändern. Die tatsächliche Implementation eines Produktes JMLS liegt ausserhalb des Rahmens dieses Projekts.

Gliederung

Dieses Dokument erklärt die Technologien und Ansätze die in Java benutzt werden können, um eine Laufzeit Übersetzung von Java Applikationen zu erreichen. Dazu werden das Design und die wichtigsten Elemente des beiliegenden Prototyps beschrieben. Eine ausführliche Klassendokumentation liegt dem Prototyp im Java – Doc Format bei.

Im ersten Teil werden die benötigten Technologien und Begriffe eingeführt und kurz erklärt. Danach wird beschrieben, wie diese Technologien ausgenutzt werden können, um eine Echtzeitübersetzung zu erreichen. Darauf folgt das Design und eine Erklärung der wichtigsten Komponenten des Prototyps und abschliessend einige Ideen, wie man die Funktionalität des Prototyps erweitern und verbessern kann.

Anforderungen

Aus der Erfahrung mit WMLS ist bekannt, dass vor allem die Erweiterbarkeit ein wichtiges Element ist bei einem Übersetzungsprogramm, da immer neue GUI Elemente und Text-Darstellungsarten erfasst und übersetzt werden müssen. Um dies zu erreichen soll die Architektur von JMLS so gewählt werden, dass alle wichtigen Elemente in unabhängigen Komponenten liegen, die jeweils beliebig erweitert und ausgetauscht werden können. Nebst diesen allgemeinen Bedingungen soll JMLS noch folgenden Anforderungen genügen:

- Echtzeitübersetzung von Java Applikationen
  Eine beliebige Java Applikation soll, mit JMLS gestartet, in eine andere Sprache übersetzt werden können. Für dieses Projekt werden nur Applikationen, welche eine Darstellung mit der java.awt Bibliothek haben, übersetzt.
- Auswahl von verschiedenen Sprachmodulen pro Applikation
  Da nicht jede Applikation das gleich Vokabular benutzt, soll pro Applikation eine oder mehrere Sprachmodule ausgewählt werden. Damit müssen nicht alle Worte in eine zentrale.
- Einfache Einbindung von zusätzlichen Darstellungselementen
  Neue Klassen, welche eine Textdarstellung vornehmen, z.B. das Swing API, sollen einfach eingebunden werden können.
- GUI zum starten und zum anpassen der Konfiguration
- Statische Anpassung von Applikationen, damit diese auch ohne mit JMLS gestartet zu werden übersetzt werden können.

Technologie

In diesem Abschnitt werden die benutzten Technologien und Eigenschaften vorgestellt, die in der Lösung benutzt werden. Es wird erst später darauf eingegangen, wie diese Eigenschaften zusammenspielen, damit eine Echtzeitübersetzung möglich wird.

Java Version

Der Prototyp wurde mit der Java Version 1.2 erstellt. Eine Echtzeitübersetzung nach dem selben Prinzip wäre auch in den anderen Versionen möglich, aber man könnte das Starten der Applikationen nicht so elegant lösen.

Zudem ist die Version 1.2 die erste wirklich ausgereifte Java Version und der grösste Teil der Applikationen, die mit den früheren Versionen von Java geschrieben wurden, funktionieren tadellos mit einer 1.2 Virtual Machine. Nur Programme, die auf ein (meist fehlerhaftes) Verhalten einer bestimmten Klasse der früheren Versionen vertrauen, könnten Fehlerhaft arbeiten.

Ein weiterer Nachteil der Java Versionen vor 1.2 besteht darin, dass ein Applet keinen eigenen ClassLoader benutzen darf. Da ein ClassLoader für die Echtzeitübersetzung notwendig ist, könnten Applets nur statisch übersetzt werden. In

Java 1.2 können Applets, unter der Voraussetzung dass sie aus einer vertrauten Quelle stammen, einen eigenen ClassLoader verwenden.

Java .class Dateien

Der Bitcode jeder kompilierten Klasse in Java ist in einer eigenen Datei mit der Endung .class abgelegt.

Die Struktur dieser Datei ist sehr interessant. Sie ist darauf angelegt, die Informationen und den Code so platzsparend wie möglich abzulegen. Die .class Datei besteht aus mehreren Teilen:

Sehr skizzenhaft kann man den Aufbau einer .class Datei wie folgt beschreiben:

1. Versionen Information
2. Konstanten Pool
3. Klassen Informationen
4. Feld Informationen
5. Methoden Informationen
6. Attribut Informationen

Die Abschnitte 1 und 3 sind eher administrativer Natur. Man kann dort die Java Version des Compilers und die Namen der Klasse und Superklasse sowie die Zugriffparameter der Klasse auslesen.

Die Abschnitte 4 – 6 enthalten die Variabeln und Methoden Definitionen sowie den eigentlichen Bitcode.

Der wirklich interessante Teil ist Abschnitt 2. Um Platz zu sparen, wird jeder String, jede Nummer etc. genau einmal aufgelistet, im Konstanten Pool. Werden die Daten in einem anderen Teil der .class Datei benötigt, so wird durch eine Referenz in den Konstanten Pool auf die Daten verwiesen.

Für genauere Informationen zum Aufbau der .class Dateien verweise ich auf die Sun Java VM Spezifikation:
http://java.sun.com/docs/books/vmspec/html/ClassFile.doc.html

Der Java ClassLoader

In Java werden Klassen dynamisch geladen. Dies bedeutet, dass die Virtual Machine eine bestimmte Klasse erst dann in den Speicher lädt, wenn zum ersten Mal eine Methode oder ein Feld dieser Klasse benutzt wird. Das auffinden und laden einer bestimmten Klasse übernimmt der sogenannte ClassLoader.

Der gesamte Ladevorgang läuft folgendermassen ab:

1. ClassLoader findet die Daten der Klasse und lädt diese in den Speicher.
2. Die VM überprüft die neu geladene Klasse auf Richtigkeit hinsichtlich Opcodes, Sprungadressen und Methodenaufrufe.
3. Statische Variablen werden erstellt und mit den Standardwerten gefüllt.
4. Symbolische Referenzen werden durch direkte Referenzen ersetzt.

Verschiedene ClassLoader werden benötigt um z.B. Klassen über eine Netzwerkverbindung zu laden oder auf eine andere Art zu finden, als es der standard ClassLoader macht. Dieser durchsucht nur alle .class Dateien und .jar Archive, die in der Umgebungsvariable CLASSPATH angegeben werden.

Wird eine Klasse von einer anderen referenziert, so wird die referenzierte Klasse mit demselben ClassLoader geladen, wie die referenzierende Klasse. Konkret bedeutet dies, dass wenn man die Basisklasse einer Java Applikationen mit einem bestimmten ClassLoader lädt, so werden alle Klassen dieser Applikation mit demselben ClassLoader geladen.

Für genauere Informationen zum Thema Laden von Klassen verweise ich auf die Sun Java VM Spezifikation:
http://java.sun.com/docs/books/vmspec/html/Concepts.doc.html#22574

Java Namespaces

Java benutzt sogenannte Namespaces, um die Klassen im Speicher von einander zu trennen. Jeder ClassLoader bekommt sein eigener Namespace. Eine Klasse in einem Namespace kann nur solche Klassen kennen und folglich deren Funktionalität aufrufen, die im selben Namespace geladen wurden.

Werden verschiedene ClassLoader benutzt, können die statischen Variablen einer bestimmten Klasse in jedem Namespace andere Werte annehmen, da die Klassen zweimal separat geladen werden.

Introspektion von Klassen

Seit Version 1.1 von Java kann man Klassen und Objekte zu Methoden, Variabeln und Zugriffsbeschränkungen befragen. Dazu benutzt man die Klassen des java.lang.reflect Pakets.

Mit der Hilfe dieses Pakets ist es z.B. möglich über Namespaces hinweg Methoden aufzurufen.

Für Informationen zu den Klassen und der genauen Funktionalität verweise ich wiederum auf die Sun Dokumentation:
http://java.sun.com/products/jdk/1.2/docs/api/index.html

Lösungsansatz

Grundidee

Der erste Schlüssel zur Übersetzung ist der Konstanten Pool in der .class Datei. Hier werden alle Informationen gespeichert und danach nur darauf verwiesen. Tauscht man nun gewisse Informationen, so werden diese Klassen sich anders verhalten, als ursprünglich vorgesehen. Tauscht man z.B. den Namen einer Klasse im Konstanten Pool aus, so kann eine ganz andere Funktion aufgerufen werden, als ursprünglich vorgesehen. Hierzu ein (semi-)konkretes Beispiel:

Die Klasse foo Besitz eine Variable vom Typ java.awt.Button und ruft irgendwann die Methode setText() dieser Variable auf.
Ersetzt man nun im Konstanten Pool die konstante Zeichenkette java.awt.Button mit jmls.classes.Button, so wird die Variable vom Typ jmls.classes.Button sein und es wird die Methode setText() der Klasse jmls.classes.Button aufgerufen.

Der zweite Schlüssel zur Lösung liegt in der Vererbung von Java Klassen. Nimmt man z.B. an, dass im vorherigen Beispiel die Klasse jmls.classes.Button ein Erbe von java.awt.Button ist, so wird foo problemlos funktionieren. Man muss nur die Methoden der Elternklasse überschreiben, welche den Zugriff auf den sichtbaren Text regeln und schon kann jeder Text übersetzt werden, bevor es dargestellt wird, ohne die normale Funktionalität der Klasse zu stören.

Macht man jetzt für jedes GUI Element (Knopf, Beschriftung, Checkbox, etc.) ein Erbe und vertauscht im Konstanten Pool aller Klassen die Namen der jeweiligen Klassen, so hat man eine Applikation, die weiss wie sich selbst übersetzten.

Um alles zusammenzubinden muss man nur noch ein Singleton schreiben, der die Kommunikation mit den neuen GUI Elementen regelt und die Übersetzung vornimmt. Hinzu kommt als zentrales Element einen eigenen ClassLoader.

Der ClassLoader muss jede Klasse finden und der VM zum Linken übergeben. Nach dem Laden und bevor die geladene Klasse der VM übergeben wird, tauscht der JMLS ClassLoader die Klassennamen im Konstanten Pool. Die auf der Festplatte gespeicherten .class Dateien werden so nicht berührt, aber die Applikation kann dennoch zu Laufzeit übersetzt werden.

Grenzen der Übersetzung

Dank der Introspektion und der Tatsache dass alle Informationen in Bitcode vorliegen ist es in Java relativ leicht auch Klassen, von denen man nicht direkt den Source-Code besitzt, zu übersetzen.

Nur final Klassen können nicht direkt durch eine Subklasse ausgetauscht werden. Es wäre auch möglich, aber nicht sauber. Da aber nur sichtbare GUI Elemente ausgetauscht werden, ist dies weniger ein Problem. z.B. in den java.awt und

javax.swing Pakten gibt es keine einzige Klasse, die nicht ausgetauscht werden könnte.

Einzig Programme die schon laufen können nicht übersetzt werden. Man muss die Applikation anhalten und mit JMLS starten, um eine Übersetzung zu ermöglichen.

Applets könnten auch ein Problem darstellen, da das einbinden eines ClassLoaders bei Applets durch das Sicherheitskonzept erschwert wird. Kommt das Applet aus einer vertrauten Quelle, darf es ab Java Version 1.2 dennoch einen eigenen ClassLoader verwenden.

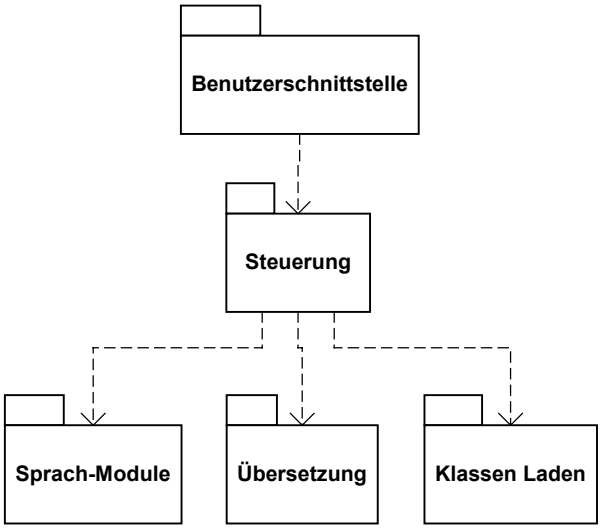
Sollte das Laden mit einem eigenen ClassLoader nicht möglich sein, z.B. wenn ein anderer, spezieller ClassLoader verwendet werden muss, kann man die Konstanten Pool Einträge statisch Austauschen. Dies bringt aber andere Probleme mit sich.

Design

Übersicht

Komponenten

Der JMLS Prototyp ist in 5 Hauptkomponenten gegliedert. Diese sind Steuerung, Module Laden, Übersetzung, Klassen Laden und Benutzereschnittstelle. Jede Komponente ausser der Steuerung und der Benutzerschnittstelle kann unabhängig von den anderen verwendet werden. Die Steuerung verbindet die anderen Komponenten zu einer Einheit und die Benutzerschnittstelle stützt sich ausschliesslich auf die Steuerung und bietet ausschliesslich eine Ansicht auf die anderen Komponenten.



Steuerung

Diese Komponente steuert wie der Name sagt die anderen Komponenten, damit diese einander unterstützen können. Hier ist keine zusätzliche Funktionalität, welche der Übersetzung dient, sondern vielmehr eine Abstraktion und Kapselung der ander Komponenten, damit eine einheitliche Schnittstelle gegen aussen geboten werden kann.

In dieser Schnittstelle wird die Konfiguration und das starten der Applikation geregelt. Dabei greift die Steuer-Komponente auf alle anderen Komponenten zu.

Die Steuerung wird ausschliesslich von der Klasse `jmls.Application` übernommen.

Klassen Laden

Um die .class Dateien zu finden, zu laden und deren Konstanten Tabelle anzupassen, wird diese Komponente benutzt. Sie ist passiv. D.h. sie wird von der Steuerung zu Beginn des Applikationstartes benutzt. Nachdem die Anpassungen gemacht wurden, wird die Funktionalität dieser Komponente nicht mehr benötigt.

Die gesamte Funktionalität wird auf verschiedene Klassen aufgeteilt, aber die Hauptschnittstellen sind `jmls.JMLSClassLoader` und `jmls.Exchanger`. `jmls.Exchanger` wiederum stützt sich sehr stark auf den ganzen `jmls.classfile` Paket, dessen Fassade `jmls.classfile.ClassFile` ist.

Sprach-Module

Eine weitere „passive“ Komponente ist die zum Laden der Sprachmodulen. Sie kann benutzt werden, um transparent die Texte in den verschiedenen Sprachen zu finden und zu laden. Hier können verschiedene Modul-Formate erkannt und gelesen werden, danach werden die gewonnen Daten in einer einheitlichen Form zurückgegeben.

Die `jmls.ModuleLoader` Klasse bietet die Schnittstelle zur Funktionalität und übergibt Informationen in der Form von `jmls.LanguageInfo` Instanzen, welche jeweils die Daten einer bestimmten Sprache kapseln und dem Übersetzer weitergeleitet werden.

Übersetzung

Im Gegensatz zu den andern Komponenten sind die Klassen welche für die Übersetzung zuständig sind nur lose miteinander verbunden. Dies kommt daher, dass die Übersetzung zu Laufzeit stattfindet und nur in bestimmten Situationen, wie bei einer Textänderung bei einer Beschriftung oder bei einem Sprachwechsel, aktiviert wird.

Die Klasse `jmls.Translator` übernimmt die eigentliche Übersetzung, aber den Auslöser geben die verschieden Klassen des jmls.classes Pakets, welche für ihre jeweiligen Elternklassen in den geladenen Applikationen eingewechselt wurden.

Benutzerschnittstelle

Die Benutzerschnittstelle gibt dem Anwender die Möglichkeit Applikationen zu konfigurieren und mit Übersetzung zu starten. Sie stütz sich ausschliesslich auf die Funktionalität der Steuerungs-Komponente.

Ablauf einer Übersetzung

Die Übersetzung läuft grob wie folgt ab:

1. Über die Benutzerschnittstelle wird eine Instanz von jmls.Application (der Steuerung) konfiguriert und mit allen nötigen Informationen zum Starten einer bestimmten Java Applikation gefüttert.
2. Wiederum über der Benutzerschnittstelle wird dieser Instanz den Befehl zum starten der Applikation gegeben.
3. Die jmls.Application Instanz initialisiert einen jmls.JMLSClassLoader.
4. Als nächstes werden die Sprachmodule ausgelesen.

5. Danach wird der Übersetzer konfiguriert und mit den Sprachmodulen gefüttert.
6. Die Basisklasse der Applikation wird mit dem neuen ClassLoader geladen und dadurch auch alle anderen Klassen.
7. Die Applikation wird gestartet.
8. Verändert sich der sichtbare Text, so wird zuerst der Übersetzer nach einer Übersetzung gefragt und dessen Antwort an Stelle des ursprünglichen Textes gesetzt.

Pakete

Der JMLS Prototyp kommt mit 5 Paketen, welche sich nicht ganz mit den 5 Komponenten decken, aber eine logische Gliederung der Klassen geben. Diese sind:

- `jmls`
- `jmls.classfile`
- `jmls.tools`
- `jmls.userinterface`
- `jmls.classes`

Einen genauen Überblick über die einzelnen Klassen der Pakete, sowie eine genaue Dokumentation aller Klassen kann in der HTML Dokumentation des JMLS Prototyps gefunden werden.

Detailansicht

Der nächste Abschnitt geht auf die Details der Implementation der verschiedenen Komponenten und wie die Klassen der verschiedenen Pakete darauf verteilt werden. Da alle Komponenten in sich geschlossen sind und nur auf eine oder zwei Klassen einer anderen Komponente zugreifen, werden die Komponenten jeweils gesondert betrachtet.

Es werden nur die wichtigsten Klassen und von diesen nur die wichtigsten Methoden beschrieben. Für eine genaue Dokumentation aller Klassen und Methoden verweise ich auf die HTML Dokumentation und den Source Code.

Steuerung

Die Klasse jmls.Application übernimmt die gesamte Steuerung der anderen Komponenten (ausser der Benutzerschnittstelle).

Alle Methoden sind Accessors für Member Variabeln oder Hilfsfunktionen, ausser der Methode:
`public void run()`

`jmls.Application` kapselt alle wichtigen Eigenschaften einer Java Applikation.

Eine Applikation kann aus einer Konfigurationsdatei ausgelesen werden, welche folgendes Format besitzen sollte:

<Basisklasse>
<Argument Liste>
<Sprachmodul Liste>

Die beiden Listen sind Strichpunkt getrennte Listen der Argumente, beziehungsweise der Module.

Beispiel:
Aus der folgenden Konfigurationsdatei würde eine `jmls.Application` erstellt, welche die Java Applikation mit Basisklasse `foo`, den Argumenten `2`,`3` und `4` mit den beiden Sprachmodulen `CommonStrings` und `RareStrings` starten würde.

```
foo
2;4;5;
/LanguageModules/CommonStrings;/LanguageModules/RareStrings
```

Klassen Laden

jmls.JMLSClassLoader

Diese Klasse ist ein Erbe von `java.lang.ClassLoader` und überlädt 2 Methoden:
`public Class loadClass(String className, boolean resolve)`
`public Class loadClass(String className)`
Wobei die zweite Methode nur ein Aufruf der ersten beinhaltet mit `resolve == false`.

`loadClass` sucht die Klasse className in der globalen Variable `CLASSPATH`. Dies bedeutet konkret, dass keine Klassen in .jar Archiven geladen werden können und der Versuch zu einem Exception und der Abbruch des Startvorgangs führen.

Falls die gewünschte Klasse gefunden wurde, wird sie als `jmls.classfile.ClassFile` eingelesen und mit der Hilfe einer Instanz von `jmls.Exchanger` abgeändert und danach als `byte[]` ausgelesen und dem Linker übergeben (dies nur, falls `resolve == true` gilt). Die resultierende `java.lang.Class` Instanz wird dann als Rückgabewert zurückgegeben.

Eine produktive Instanz von jmls.JMLSClassLoader müsste zusätzlich alle geladenen Klassen zwischenspeichern, um weitere Zugriffe zu beschleunigen. Information dazu kann der Sun Java Dokumentation zum Thema `java.lang.ClassLoader` entnommen werden.

jmls.Exchanger

`jmls.Exchanger` verlangt bei der Konstruktion eine Nx2 Matrize. Der Inhalt dieser Matrize ist eine Tabelle, wo die Klassennamen, welche ausgetauscht werden sollen, aufgeführt werden. Die erste Spalte ist für die ursprüngliche Klassennamen, die zweite Spalte für die neuen Klassennamen. Java benutzt dabei den ‚/' Zeichen, um Pakete zu trennen. Diese Schreibweise ist die sogenannte „fully classified class representation".

Beispiel:
Sollen die Klassen `java.awt.Button` und `java.awt.Label` ersetzt werden, so sieht die Tabelle so aus:

```
String[][] {{java/awt/Button, jmls/classes/jmlsButton},
           {[java/awt/Button, [jmls/classes/jmlsButton},
           {java/awt/Label, jmls/classes/jmlsLabel}
           {[java/awt/Label, [jmls/classes/jmlsLabel}}
```

Die Schreibweise `[java/awt/Button` bezeichnet dabei ein Array von `java.awt.Button`, `[[java/awt/Button` ein Array von Arrays usw. Im Prototyp ist es notwendig, alle anzugeben. Eine produktive Implementation von `jmls.Exchanger` müsste dies unnötig machen, indem die Klassenname beim Ersetzen aufgeschlüsselt würden.

Für ein produktives System müsste diese Tabelle in `jmls.Application` aus einer Konfigurationsdatei ausgelesen werden, damit man pro Applikation die Klassen bestimmen könnte. Im Prototyp wird die Tabelle statisch in `jmls.JMLSClassLoader` bzw. `jmls.StaticGrabber` erzeugt.

Um nun Anhand der oben beschriebenen Tabelle die Einträge des Konstanten Pools einer Klasse zu ersetzen, wird die folgende Methode aufgerufen:
`public ClassFile exchangeConstantPoolInfoIn(ClassFile classFile)`

Jeder Eintrag im Konstanten Pool von `classFile` wird darin ausgelesen und begutachtet. Ist ein Eintrag vom Typ `jmls.classfile.Utf8Info` wird nachgeschaut, ob es ein Klassenname ist, der ausgetauscht werden muss. Falls ja, wird der Text im diesem `Utf8Info` ausgetauscht, sonst wird der Eintrag ignoriert.

Um im ClassLoader eine Endlosschleife beim Laden der Klassen zu verhindern, werden die Klassen des Paketes `jmls.classes` nicht angeschaut. Diese sind darauf angewiesen, dass sie die ursprüngliche Implementationen ihrer Eltern verwenden können.

jmls.classfile

Dieses Paket basiert auf der Java Classfile Spezifikation. Alle Teile einer .class Datei werden abgebildet.

Die Klasse `jmls.classfile.ClassFile` dient als Schnittstelle zu allen anderen Klassen des Paketes. Die Grobstruktur ist:

- Magische Zahl
  0xCAFEBABE
- Major Version
  1 Byte
- Minor Version
  1 Byte

- Constant Pool
  Constant Pool Grösse N
  N-mal ein Constant Pool Info
- This Class
  1 Byte – Index in den Constant Pool
- Super Class
  1 Byte – Index in den Constant Pool
- Field Info Pool
  Field Info Pool Grösse N
  N-mal ein Field Info
- Method Pool
  Method Pool Grösse N
  N-mal ein Method Info
- Attribute Pool
  Attribute Pool Grösse N
  N-mal ein Attribute

Im Field Info Pool werden Informationen zu den Members der Klasse gespeichert, im Method Pool Informationen zu den Methoden und im Attribute Pool weitere Informationen und der eigentliche Bitcode. Field Info, Method Info und Attribute verweisen alle für Namen, Konstanten etc. in den Konstanten Pool. Der Attribute Pool verweist seinerseits auf die Field Info und Method Pools, um jegliche Redundanz zu verhindern.

Ein Constant Pool Info kann eines der folgenden Typen haben:

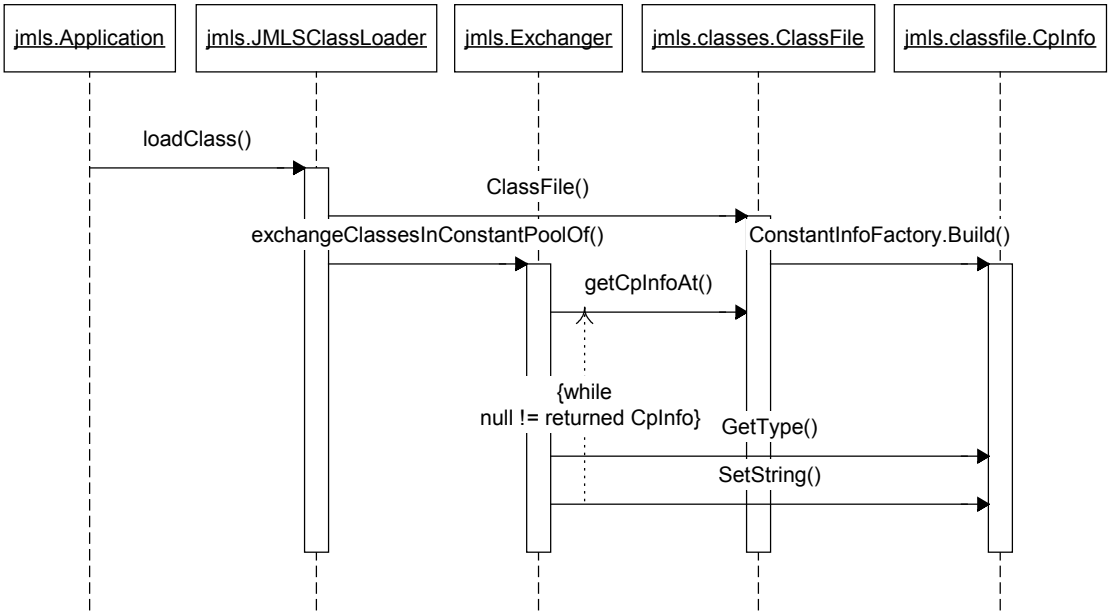
- Integer
- Long
- Float
- Double
- String
- Utf8
- NameAndType
- Class
- Methodref

Die Klasse `jmls.classfile.CpInfoFactory` stellt sicher, dass jeweils der richtige Erbe von `jmls.CpInfo` beim Einlesen die Informationen kapselt.

Für alle Details der Spezifikation verweise ich wiederum auf die Sun Dokumentation unter http://java.sun.com/docs/books/vmspec/html/ClassFile.doc.html und der beiliegenden HTML Dokumentation.

Ablauf beim Ersetzen

Dies ergibt den folgenden Ablauf, um eine Klasse zu laden und die Konstanten Pool Einträge zu ersetzen.



Module Laden

Die jmls.ModuleLoader Klasse kann im Prototyp nur sehr einfach formatierte Dateien einlesen. Ein java.util.Vector mit allen Sprachen, die in einer angegebenen Datei gefunden werden kann mit dieser Methode abgefragt werden:
`public Vector loadModule(String moduleName)`

Die Datei muss den Format haben:

<Anzahl Sprachen n>
<Name der 1. Sprache>
<1. Wortliste>
<Name der 2. Sprache>
<2. Wortliste>
...
<Name der n. Sprache>
<n. Wortliste>

Die Begriffe müssen jeweils auf einer Zeile geschrieben werden. Die entsprechenden Wörter der verschiedenen Sprachen müssen an gleicher Stelle in den Listen liegen.

Es wäre problemlos möglich, weitere Formate (z.B. WMLS .MLS Dateien) einzulesen, welche z.B. anhand der Dateiendung zu erkennen wären.

Übersetzung

Zur Übersetzung einer Applikation müssen alle Klassen, welche in dieser Applikation zur Darstellung von Text auf dem Bildschirm benutzt werden, abgeleitet werden. In den abgeleiteten Klasse werden alle Methoden, welche den dargestellten Text verändern nach dem folgenden Muster überladen:

1. Übersetzer nach der Übersetzung fragen.
2. Original Text zwischenspeichern, falls notwendig.
3. Die überladene Methode der Eltern Klasse mit der Übersetzung aufrufen.

Im Prototyp sind exemplarisch 4 Klassen des `java.awt` Paketes übersetzt. Diese sind:

```
java.awt.Button
java.awt.Checkbox
java.awt.Label
java.awt.Listbox
```

Diese werden abgeleitet und beim Laden der Klassen jeweils ersetzt durch:

```
jmls.classes.jmlsButton
jmls.classes.jmlsCheckbox
jmls.classes.jmlsLabel
jmls.classes.jmlsListbox
```

Die Kommunikation der abgeleiteten Klassen mit dem Übersetzer `jmls.Translator` ist sehr einfach. Da `jmls.Translator` keine Sprachauswahl zu Laufzeit zulässt, wird die Übersetzung mit dieser Funktion gemacht:
`public String translate(String oldText)`

`jmls.Translator` ist ein Singleton dessen Instanz mit dieser Funktion geholt wird:
`public static Translator getInstance()`

Um eine Sprachumschaltung zu Laufzeit zu ermöglichen, müsste der Übersetzer nur ein Fenster mit einer Sprachauswahl erzeugen und eine List führen mit allen Objekten, die eine Übersetzung verlangt haben. Wird die Zielsprache geändert, werden die zwischengespeicherten Objekten benachrichtigt und deren Text in die neue Sprache übersetzt. Eine Anpassung der abgeleiteten Klassen wäre auch notwendig. Diese müssten den Text in der Ursprungssprache zwischenspeichern, um eine später Übersetzung zu vereinfachen.

Benutzerschnittstelle

Die Benutzerschnittstellen sind in zwei Kategorien unterteilt: Graphische und Zeilenkommando basierte Schnittstellen.

`jmls.tools` bietet zwei Zeilenkommando basierte Programme:

- `jmls.tools.JMLS`
  Kann benutzt werden zum Starten einer bestimmten Applikation, welche im config file beschrieben wird..

  ```
  java jmls.tools.JMLS <config file>
  ```

- `jmls.tools.static.JMLS`
  Kann benutzt werden, um eine statisch übersetzte Applikation zu starten. Es kann ein Sprachmodul übergeben werden, um eine Übersetzung zu ermöglichen.

  ```
  java jmls.tools.static.JMLS <modulename> <classname>
  [<arguments>]
  ```

- `jmls.tools.StaticGrabber`
  Kann benutzt werden, um die Konstanten Tabelle einer Klasse statisch zu verändern.

  ```
  java jmls.tools.StaticGrabber <class file name> [<new
  class file name>]
  ```

  Wird nur der erste Dateiname angegeben, wird die abgeänderte Datei nach `System.out` geschrieben. Die ausgegebene Datei wird eine Klasse mit einem anderen Klassennamen enthalten als die Ursprüngliche.

`jmls.userinterface.JMLStheGUI` ist eine graphisch Benutzeroberfläche, um Applikationen zu konfigurieren und zu starten.

Man muss `JMLStheGUI` einen Pfad angeben, wo die Konfigurationsdateien der gewünschten Applikationen liegen oder abgespeichert werden sollen.

```
java jmls.userinterface.JMLStheGUI <configurationpath>
```

Mit „Add“ wird eine neue konfigurations Datei erstellt und das Editierfenster geöffnet. Mit „Edit“ kann eine ausgewählte, schon bestehende Applikation im Editierfenster geöffnet werden. Mit „Delete“ wird die Konfigurationsdatei der ausgewählte Applikation gelöscht und mit „Start“ die ausgewählte Applikation gestartet.

Erweiterungen

Nebenläufigkeit

Im Moment kann man in JMLStheGUI mehrere Applikationen gleichzeitig starten. Dies funktioniert sehr gut, da jede Applikation den eigenen ClassLoader und somit den eigenen Namespace, den eigenen Übersetzer und den eigenen Speicher erhält.

Beendet jedoch eine Applikation mit einem Aufruf der Funktion `System.exit(int state)`, so wird auch die JMLStheGUI Applikation, und somit alle Applikationen, gestoppt.

Eine produktive JMLS Lösung müsste mit echten Threads arbeiten oder eine andere Regelung der nebenläufigen Ausführung von Applikationen finden.

Sprachauswahl zu Laufzeit

Ist in diesem Projekt nicht implementiert. Eine möglich Lösung dieses Problems wird weiter oben beschrieben.

Konfiguration der auszutauschenden Klassen

Dies ist ein weiteres Problem, das durch eine weiter Konfigurationsdatei gelöst werden könnte. Man müsste aber einige Anpassungen and die `jmls.Exchanger`, `jmls.JMLSClassLoader` und `jmls.Application` Klassen machen. Auch der Format der .jcf Dateien müsste angepasst werden, damit jede Applikation die eigene „Austauschliste“ führen könnte.

Module laden auf Basis von WMLS Dateien

WMLS hat ein eigenes Dateiformat, um Texte in mehreren Sprachen abzulegen. Diese Dateien werden mit der Extension .MLS gekennzeichnet. Eine sinnvolle Erweiterung wäre die Möglichkeit .MLS Dateien als Quelle für Sprachen einzulesen. Damit könnten bisher erstellte Module benutzt werden.

Man könnte auch den WMLS Editor benutzen, um neue Module für JMLS zu erstellen.

Aufnahmemodus beim Übersetzer

Um das Auffinden der Texte zu erleichtern, könnte man den Übersetzer mit einer Aufnahmemoduls ausstatten. In diesem Modus würde der Übersetzer anstatt die ihm zur Übersetzung übergebenen Texte abspeichern. Aufgrund dieser Aufnahmen könnte man danach die Sprachmodule machen.

Editor

Für eine vollständige JMLS Implementation wäre auch ein Editor notwendig, um Übersetzungsmodule zu erstellen.

Abschiessende Bemerkungen

Abschliessend möchte ich noch eine kurze Beurteilung meines Projektes machen. Ich werde dabei anschauen was gut gelaufen ist, was ich dabei gelernt habe und was ich anders machen würde.

In meinen Augen ist meine grundsätzliche Vorgehensweise richtig gewesen. Ich habe zuerst die theoretischen Grundlagen angeschaut, mich mit der Programmiersprache Java und der Virtual Machine auseinandergesetzt. Dabei habe ich Lösungswege gesucht, welche den Anforderungen genügen. Nach der Einschränkung auf ein paar Lösungswege habe ich die Besten ausprobiert, danach ein Design erstellt und erst zum Schluss mit der Implementation begonnen. Dies hat mir viel Zeit gespart und hat zu einer kompakten und erweiterbaren Lösung geführt.

Die zeitliche Einteilung meines Projektes ist auf der anderen Hand nicht sehr gut gegangen. Während ich das Projekt bearbeitet habe, habe ich auch in einer Firma gearbeitet, für die Fachschaft ein Projekt bearbeitet und dazu Vorlesungen und Praktika besucht. Dies führte zu grossen Verzögerungen und langen Pausen zwischen den einzelnen Arbeitsschritten. Durch diese Verzögerungen und der Bearbeitung von bis zu vier Projekten gleichzeitig ist für keine der Projekte genug Zeit geblieben. Ein anderes Mal würde ich jedes Projekt nach Wichtigkeit zu Ende bringen und jeweils 100% an einem Projekt arbeiten. Auf diese Weise spart man in meiner Erfahrung sehr viel Zeit, da die Wiedereinarbeitung in ein Projekt immer eine unproduktive Phase ist. Mehr als zwei Projekte gleichzeitig bearbeiten zu wollen führt dazu, dass man mehr Zeit mit „Task-Switching“ verbringt und nur sehr wenig Zeit damit, etwas Produktives zu machen. Dies ist wahrscheinlich die wichtigste Lehre, die ich aus der Bearbeitung dieses Projektes ziehen werde.

Eine weitere Änderung, die ich an meiner Arbeitsweise machen werde, ist das erstellen von realistischen Meilensteinen. Anhand solcher Meilensteine ist es dann einfacher den eigenen Fortschritt zu verfolgen und auch Auskunft über den Status des Projektes zu geben.

Im Ganzen bin ich zufrieden mit meinem Projekt, obwohl es einen sehr lange gedauert hat, haben andere Teile des Projektes gut geklappt. Mit einer besseren Einteilung meiner anderen Projekte wäre dies nicht passiert.