Diplomarbeit

Konfiguration von Java-Applikationen durch Abhängigkeitsanalyse

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

Angefertigt von: **Christoph Stoike**

Betreuung durch: Prof. Dr. Michael Hanus

Dr. Christian Friberg,
PPI AG Informationstechnologie, Kiel

Vorgelegt: November 2007

Eidesstattliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt zu haben.

....
Christoph Stoike

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Betriebliches Umfeld . . . 1
1.2. Motivation und Aufgabenstellung . . . 1
1.3. Verfügbare Produkte . . . 3
1.4. Aufbau der Arbeit . . . 4

2. Grundlagen **5**
2.1. Java Archive . . . 5
2.2. Java Enterprise Edition . . . 5
2.3. JNDI . . . 9
2.4. Reflection . . . 9
2.5. XML . . . 10
2.6. Ant . . . 10

3. Analyse von Abhängigkeiten zwischen JavaEE-Komponenten **11**
3.1. Konzept . . . 11
3.1.1. Ausgangssituation . . . 11
3.1.2. Vorgehensweise . . . 11
3.1.3. Einschränkungen . . . 12
3.2. Vorbereitungen . . . 13
3.3. Bestimmung der Komponenten der Applikation . . . 14
3.4. Bestimmung der direkten Abhängigkeiten . . . 17
3.5. Erstellung eines Abhängigkeitsgraphen . . . 20
3.6. Aktualisierung der Deployment-Deskriptoren . . . 21
3.7. Übergang zur Klassenebene . . . 22

4. Analyse der Abhängigkeiten zwischen Java-Klassen **23**
4.1. Unterschiedliche Analyseebenen . . . 23
4.1.1. Herausforderungen der Analyse auf Methodenebene . . . 24
4.2. Parsen von `class`-Dateien . . . 26
4.2.1. Der Konstanten-Pool . . . 27
4.2.2. Access-Flags . . . 28
4.2.3. Superklasse und Interfaces . . . 28
4.2.4. Felder . . . 28
4.2.5. Methoden . . . 29
4.2.6. Attribute . . . 32
4.3. Bestimmung der benötigten Klassen . . . 33
4.3.1. Datenstrukturen . . . 33
4.3.2. Der Algorithmus . . . 34
4.3.3. Eingabe . . . 35
4.3.4. Vorbereitung der Analyse . . . 36

4.3.5. Einschränkungen bei der Analyse . . . 38
4.3.6. Bestimmung der abhängigen Klassen . . . 39
4.3.7. Behandlung von Interfaces und Klassenerweiterungen . . . 44
4.4. Behandlung von Spezialfällen . . . 47
4.4.1. Reflection . . . 48
4.4.2. „Version“-Klassen . . . 50
4.4.3. Automatisch generierte Klassen . . . 50
4.5. Erstellung der neuen Applikation . . . 51
4.6. Konfigurationsmöglichkeiten . . . 52
4.7. Vergleich Klassen- und Methodenebene . . . 53

5. Visualisierung im Rahmen einer graphischen Benutzeroberfläche 56
5.1. Visualisierung der Abhängigkeiten zwischen JavaEE-Komponenten . . . 56
5.1.1. Beschreibung der Visualisierung . . . 56
5.1.2. Graph-Layout . . . 57
5.1.3. Zeichnen der Knoten . . . 60
5.2. Beispiel einer kompletten Analyse-Session . . . 61

6. Service Provider Interfaces 65
6.1. Zielsetzung . . . 65
6.1.1. Definition und Ziel . . . 65
6.1.2. Anforderungen . . . 65
6.2. Implementierung . . . 68
6.2.1. Eingabe und Vorbereitungen . . . 68
6.2.2. Bestimmung der direkten Abhängigkeiten . . . 68
6.2.3. Rekursive Bestimmung indirekter Abhängigkeiten . . . 69
6.2.4. Abschluss der Analyse . . . 69
6.3. Eine graphische Oberfläche . . . 69

7. Signatur-Matching 71
7.1. Ziel . . . 71
7.2. Matching-Arten . . . 72
7.2.1. Definition: Subtyp und Supertyp . . . 72
7.2.2. Exact Match . . . 73
7.2.3. Reorder Match . . . 74
7.2.4. Generalized Match . . . 74
7.2.5. Specialized Match . . . 74
7.2.6. Rename Match . . . 75
7.3. Implementierung . . . 75
7.3.1. Eingabe . . . 75
7.3.2. Bestimmung von Supertypen . . . 77
7.3.3. Bestimmung der Signaturen . . . 77
7.3.4. Durchführen des Matchings . . . 78
7.3.5. Ergebnis der Analyse . . . 81

8. Zusammenfassung und Ausblick 82

A. Instruktionssatz der `Java Virtual Machine` 84

1. Einleitung

In diesem Kapitel soll eine Einführung in die Thematik dieser Diplomarbeit gegeben werden. Bevor auf die genaue Zielsetzung eingegangen wird, erfolgt eine kurze Vorstellung der Firma *PPI AG Informationstechnologie*, die diese Arbeit unterstützt hat.

1.1. Betriebliches Umfeld

Die *PPI AG Informationstechnologie* ist ein Beratungs- und Softwareunternehmen, das auf Finanzdienstleister spezialisiert ist. Es besitzt Geschäftsstellen in Hamburg, Kiel und Frankfurt.

PPI ist Hersteller der Produktfamilie TRAVIC (Transaction Services), deren Komponenten den elektronischen Zahlungsverkehr für Firmen- und Privatkunden unterstützen. Eines dieser Produkte ist TRAVIC Retail, in dessen Umfeld diese Diplomarbeit Anwendung findet. Hierbei handelt es sich um ein auf der JavaEE-Technologie basierendes Bankgeschäftssystem, das unter anderem als Banking-Server für Browser- und Homebanking mit HBCI oder FinTS eingesetzt wird.

TRAVIC Retail setzt sich aus mehreren modularen Komponenten zusammen, die insgesamt aus bis zu 16.000 Java-Klassen bestehen. Welche dieser Komponenten die fertige Anwendung bilden, hängt von den Bedürfnissen des Kunden habt. Eine solche Berücksichtigung unterschiedlicher Auslieferungsumfänge ist der Ausgangspunkt dieser Arbeit.

1.2. Motivation und Aufgabenstellung

Die Handhabung großer Softwareprojekte gestaltet sich oft schwierig, da bei steigendem Volumen die Transparenz schnell verloren geht.

Einige dabei auftretende Fragestellungen sind:

- *Komplexität:*
  Wie hängen die einzelnen Komponenten eines Programms untereinander zusammen?
- *Wartung:*
  Gibt es Komponenten, die nicht mehr verwendet werden?
- *Wiederverwendbarkeit:*
  Können bei der Erweiterung der Software bestehende Komponenten wiederverwendet werden?

Die Beantwortung dieser Fragen in Bezug auf Java-Applikationen ist der Kern dieser Arbeit.

Es ist also eine Analyse der Struktur jener Anwendungen durchzuführen. Diese bildet die Grundlage für die Realisierung der primären Aufgabenstellung:

Gegeben ist eine JavaEE-Applikation, aus der eine neue Anwendung erzeugt werden soll. Dazu werden gewisse Komponenten der bestehenden selektiert, die die gewünschte Funktionalität definieren. Diese sowie alle von ihnen benötigten Programmeinheiten bilden schließlich die neue JavaEE-Anwendung. Insbesondere wird es damit möglich sein, aus einer vollständigen Distribution des PPI-Produkts TRAVIC Retail unterschiedliche Auslieferungsumfänge zu generieren.

Die Bestimmung der Abhängigkeitsstruktur dient somit dem Ziel, ein lauffähiges Programm zu erzeugen. Die Konsequenz davon ist, dass es bei der Analyse nicht ausreicht, nur statische Abhängigkeiten zu betrachten. Es müssen gerade auch dynamische, d.h. solche, die erst zur Laufzeit auftreten, berücksichtigt werden.

Eine JavaEE-Anwendung lässt sich in zwei Schichten unterteilen. Die obere wird durch die technischen Komponenten der Anwendung definiert, im Folgenden *JavaEE-Komponenten* genannt. Deren Umsetzung erfolgt durch Java-Klassen. Sie bilden die zweite Schicht.

In beiden Fällen können Abhängigkeiten zwischen den jeweiligen Programmeinheiten bestehen. Im Fall der Java-Klassen können diese erneut auf zwei Ebenen betrachtet werden, die aber im Gegensatz zur zuvor angegebenen Unterteilung alternativ zueinander sind. Es werden hier eine Analyse auf Klassen- und eine auf Methodenebene unterschieden. Letztere ist genauer, aber auch deutlich komplexer.

Abbildung 1.1 gibt noch einmal einen Überblick über die verschiedenen Ebenen der Abhängigkeitsanalyse bei einer JavaEE-Applikation:

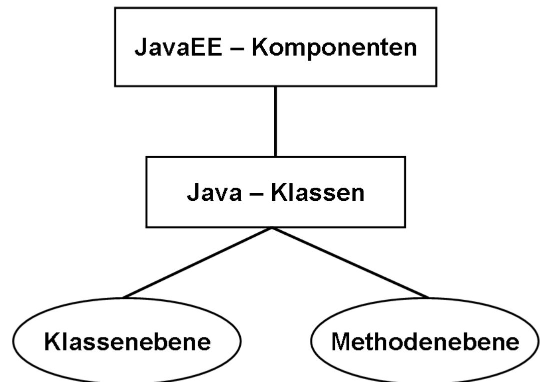


Abbildung 1.1.: Ebenen der Abhängigkeitsanalyse

Die in dieser Arbeit vorgestellte Analyse umfasst alle hier aufgeführten Levels. Bei der Bestimmung der Klassenabhängigkeiten auf Methodenebene wird auf Grund der Komplexität ein approximierender Ansatz verwendet.

An Hand des Ergebnisses dieser Analyse wird es dann möglich sein, eine neue JavaEE-Anwendung zu erzeugen, die in Bezug auf die an sie gestellten Anforderungen eine (nahezu) minimale Teilmenge der Ausgangsanwendung darstellt.

Die Erstellung neuer Applikationen soll von einer graphischen Benutzeroberfläche aus erfolgen. In diese soll auch eine Visualisierung der Abhängigkeitsstruktur der analysierten Anwendung integriert werden.

Eine weitere Problemstellung, die zu der eben geschilderten sehr verwandt ist, betrifft die Behandlung von so genannten *Service Provider Interfaces (SPIs)*. Diese Schnittstellen dienen der Definition von austauschbaren Komponenten innerhalb einer Java-Anwendung. Im Rahmen von TRAVIC Retail werden SPIs dazu verwendet, es dem Kunden zu ermöglichen, eigene Implementierungen für bestimmte Komponenten zu verwenden. Diese werden dann über ein solches SPI mit der Anwendung verknüpft.

Die Aufgabe ist es nun, eine minimale Menge an Java-Klassen zu bestimmen, die zur Implementierung eines gegebenen Service Provider Interfaces erforderlich ist. Dadurch wird es möglich sein, dem Kunden das SPI inklusive dieser Klassen noch vor der Auslieferung des finalen Produkts bereitzustellen. Er kann so bereits frühzeitig mit der Implementierung des Interfaces beginnen.

Die Wiederverwendbarkeit von Softwarekomponenten steht im Mittelpunkt des dritten Teils dieser Arbeit. Hier soll ein Werkzeug erstellt werden, das zu einem gegebenen Interface nach Klassen sucht, die dieses implementieren könnten. Grundlage dieser Suche ist ein Vergleich der Signaturen von Interface und potentieller Implementierung.

Als theoretische Basis für ein solches *Signatur-Matching* dient eine Arbeit von Amy Moormann Zaremski und Jeannette M. Wing [1], in der auch ein Signatur-Matcher für StandardML implementiert wurde. Darin werden verschiedene Arten von Matchings eingeführt, die in dieser Diplomarbeit in Bezug auf Java-Applikationen umgesetzt werden.

1.3. Verfügbare Produkte

Es gibt zahlreiche Produkte zur Analyse von Java-Programmen, von denen viele auch die Abhängigkeitsstruktur untersuchen. Einige davon sollen hier kurz vorgestellt werden.

JarAnalyzer [1], JDepend [2] und Classcycle [3] sind Programme, die statische Abhängigkeiten zwischen Java-Klassen ermitteln, d.h. es werden Abhängigkeiten bestimmt, die bereits zur Compile-Zeit bekannt sind. Die Auswertung der Analyseergebnisse erfolgt bei den einzelnen Werkzeugen auf unterschiedlichen Ebenen.

JarAnalyzer und JDepend führen eine Messung der Design-Qualität in Hinsicht auf Wiederverwendbarkeit, Erweiterbarkeit und Wartbarkeit durch. Ersterer auf Ebene von Java-Archiven, letzterer auf Paketebene. Als Basis dafür dienen ein- und ausgehende Abhängigkeiten der betrachteten Archive bzw. Pakete. So können unter anderem auch zyklische Abhängigkeiten ermittelt werden. Dies ist auch das primäre Ziel des Programms Classcycle. Der Unterschied zu den zuvor genannten ist, dass hier die Auswertung auf Klassenebene erfolgt.

Ein Werkzeug, das die Analyse auf allen drei Ebenen kombiniert, ist der Class Dependency Analyzer [4]. Er ermöglicht die Bestimmung von statischen Abhängigkeiten einer einzelnen Klasse, eines ganzen Pakets oder eines Archivs. Auch die Auswertung der Ergebnisse kann auf allen drei Ebenen erfolgen. Ferner wird eine Visualisierung in Form eines Abhängigkeitsgraphen angeboten.

Das kommerzielle Programm Structure101 [5] ist ein sehr umfangreiches Werkzeug zur Strukturanalyse von Software. Es erlaubt unter anderem eine Bestimmung der Abhängigkeiten bis auf Methodenebene und die Visualisierung der Abhängigkeitsstruktur auf allen Hierarchieebenen. Ebenso bietet es die Möglichkeit, die Ursache für eine bestimmte Abhängigkeit über alle Schichten hinweg nachzuvollziehen.

Alle hier vorgestellten Produkte haben eine Gemeinsamkeit: Sie bestimmen direkte, statische Abhängigkeiten. Um die Zielsetzung der Diplomarbeit erreichen zu können, eine neue Java-Applikation mit Hilfe einer Abhängigkeitsanalyse zu erstellen, ist es aber auch notwendig, dynamische zu berücksichtigen. Ferner ist bei einer Analyse auf Methodenebene auf Grund der Konzepte von Interfaces und Vererbung in Java auch hier eine speziellere Vorgehensweise anzuwenden als es bei den vorgestellten Programmen der Fall ist.

Wie in Abschnitt 1.2 erläutert, sollen auch die Beziehungen zwischen einzelnen JavaEE-Komponenten dargestellt werden. Die dazu bereits verfügbaren Werkzeuge beschränken sich jedoch auf Entwicklungs-

[1] http://www.kirkk.com/main/Main/JarAnalyzer
[2] http://www.clarkware.com/software/JDepend.html
[3] http://classycle.sourceforge.net/
[4] http://www.dependency-analyzer.org/
[5] http://www.headwaysoftware.com/products/structure101/

umgebungen wie IBMs RATIONAL APPLICATION DEVELOPER FOR WEBSPHERE SOFTWARE [6], wo lediglich eine textuelle Auflistung direkter Abhängigkeiten angeboten wird.

1.4. Aufbau der Arbeit

Nachdem in diesem Kapitel bereits auf die Zielsetzung der Arbeit und ähnliche, auf dem Markt verfügbare Produkte eingegangen wurde, werden in Kapitel 2 zunächst grundlegende Technologien erläutert, auf denen diese aufbaut.

Die Kapitel 3 und 4 beschreiben die Bestimmung der Programmabhängigkeiten und wie darauf aufbauend eine neue Applikation generiert werden kann. In Ersterem erfolgt die Analyse der Beziehungen zwischen JavaEE-Komponenten. Die untergelagerte Schicht der Java-Klassen ist Gegenstand des folgenden Kapitels. Hier werden die Abhängigkeiten in beiden bereits erwähnten Varianten bestimmt, auf Klassen- und auf Methodenebene. Die jeweiligen Resultate werden an Hand praktischer Beispiele miteinander verglichen, ehe schließlich die Zusammenstellung der neuen JavaEE-Anwendung erfolgt.

In Kapitel 5 wird dann eine Visualisierung der Beziehungen zwischen den JavaEE-Komponenten im Rahmen einer graphischen Benutzeroberfläche vorgestellt, welche eine interaktive Konfiguration der zu erstellenden Applikation erlaubt.

Anschließend wird im 6. Kapitel auf die Bestimmung der zur Implementierung eines Service Provider Interfaces notwendigen Klassen eingegangen. Hier wird gezeigt, wie der zuvor in Kapitel 4 entwickelte Algorithmus durch geringfügige Änderungen an die neuen Anforderungen angepasst werden kann.

In Kapitel 7 wird die Implementierung eines Signatur-Matchings präsentiert. Zunächst werden dort verschiedene Matching-Arten vorgestellt, die als Basis dazu dienen, potentielle Implementierungen für Java-Interfaces zu ermitteln.

Kapitel 8 fasst diese Arbeit zusammen und gibt einen Ausblick auf Erweiterungsmöglichkeiten des hier vorgestellten Programms.

[6] `http://www-306.ibm.com/software/awdtools/developer/application/`

2. Grundlagen

In diesem Kapitel werden die Technologien genauer erläutert, auf denen diese Arbeit aufbaut bzw. die zur Erstellung dieser Arbeit verwendet wurden.

2.1. Java Archive

Es existieren einige Archivformate für Java-Anwendungen, die in diesem Abschnitt kurz vorgestellt werden.

JAR

Als grundlegendes Format zur Archivierung von Java-Anwendungen dient das *Java Archive (JAR)*-Format [2]. Hierbei handelt es sich um eine ZIP-Datei mit der Endung `.jar`, die um zusätzliche Metadaten erweitert wird. Diese werden innerhalb des Archivs in der Datei `META-INF/MANIFEST.MF` gespeichert und enthalten Informationen der zu Grunde liegenden Java-Applikation wie den Namen der Hauptklasse oder den Klassenpfad, in dem andere JARs referenziert werden können.

WAR

Ein *Web Archive (WAR)* [3] ist ein Format zur Archivierung von Java-Web-Anwendungen, welches die Dateiendung `.war` besitzt. Es erweitert das JAR-Format und zeichnet sich durch eine spezielle Verzeichnisstruktur aus. So enthält es ein Verzeichnis `WEB-INF`, in dem neben kompilierten Java-Klassen und benötigten Hilfsbibliotheken ein so genannter *Deployment-Deskriptor* namens `web.xml` liegt. Hierbei handelt es sich um eine XML-Datei, die zur Konfiguration der zu Grunde liegenden Anwendung dient. In ihr werden zum Beispiel die Servlet-Klassen der Anwendung definiert.

EAR

Auch das *Enterprise Archive (EAR)* [4] ist eine Erweiterung des JAR-Formats. Es trägt die Endung `.ear` und enthält eine JavaEE-Applikation. Analog zum WAR-Format enthält auch eine EAR-Datei einen Deployment-Deskriptor, der hier den Namen `application.xml` trägt und unter anderem Informationen über die einzelnen Komponenten der Anwendung enthält. Eine genauere Beschreibung des Aufbaus einer EAR-Datei erfolgt im nächsten Abschnitt.

2.2. Java Enterprise Edition

Bei den Anwendungen, die in dieser Diplomarbeit analysiert werden, handelt es sich um JavaEE-Applikationen. Daher wird diese Technologie im Folgenden auf Basis des JavaEE-Tutorials [4] genauer vorgestellt.

Die *Java Platform, Enterprise Edition* [1], kurz *JavaEE* oder früher *J2EE*, ist der industrielle Standard zur Entwicklung von transaktionsbasierten, serverseitigen Java-Anwendungen in einer verteilten Umgebung. Wichtige Merkmale sind dabei Portabilität, Sicherheit, Robustheit und Geschwindigkeit.

JavaEE benutzt eine mehrschichtiges Anwendungsmodell, wie in Abbildung 2.1 dargestellt wird.

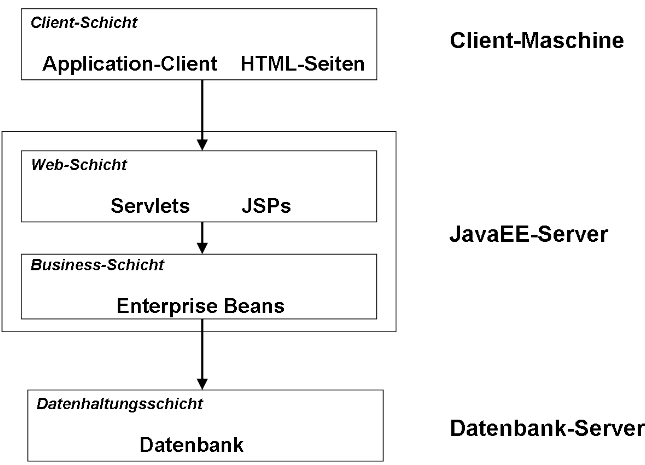


Abbildung 2.1.: Mehrschichtige Anwendungen

Obwohl hier vier Schichten zu erkennen sind, werden JavaEE-Applikationen in der Regel als dreischichtige Anwendungen betrachtet, da sie über drei Orte verteilt sind: Client-Maschine, JavaEE-Server-Maschine und eine Datenbank.

Eine solche Anwendung setzt sich aus drei Arten von Komponenten zusammen:

- Application-Clients
- Web-Komponenten, wie Java Servlets oder JavaServer Pages (JSP)
- Enterprise JavaBeans (EJB)

Application-Clients laufen auf der Client-Maschine und ermöglichen dem Nutzer den Zugriff auf die JavaEE-Anwendung in der Regel über eine graphische Benutzeroberfläche. Eine Alternative hierzu, die in ihren Möglichkeiten etwas limitierter ist, sind Web-Clients. Hierbei erfolgt der Zugriff mit Hilfe eines Browsers über dynamische, von Web-Komponenten generierte HTML-Seiten.

Diese Web-Komponenten laufen, genauso wie Enterprise Beans, auf Seiten des Servers. Sie stellen die Präsentationsschicht der Anwendung dar, d.h. sie dienen der Darstellung der Inhalte und der Entgegennahme von Benutzereingaben, während Enterprise Beans die Geschäftslogik bereitstellen. Der Datenaustausch zwischen diesen beiden Schichten kann unter anderem über *Java-RMI*, *CORBA* oder *HTTP* erfolgen.

Die vorgestellten Komponenten benötigen eine spezielle Laufzeitumgebung, einen so genannten JavaEE Application Server. Dieser stellt Funktionalitäten wie die Kommunikation zwischen einzelnen JavaEE-Komponenten, Transaktionsmanagement, Sicherheit oder Namens- und Verzeichnisdienste zur Verfügung. Es existieren zahlreiche verschiedene Implementierungen für solche Server. Im Rahmen dieser Arbeit werden nur die für das PPI-Produkt TRAVIC Retail relevanten behandelt: der *IBM Websphere Application Server* [2] und der *JBoss Application Server* [3].

[1] `http://java.sun.com/javaee/`
[2] `http://www-306.ibm.com/software/webservers/appserv/was/`
[3] `http://www.jboss.org/products/jbossas`

Die Auslieferung einer JavaEE-Anwendung erfolgt in Form einer EAR-Datei. Ihren wesentlichen Aufbau beschreibt Abbildung 2.2.

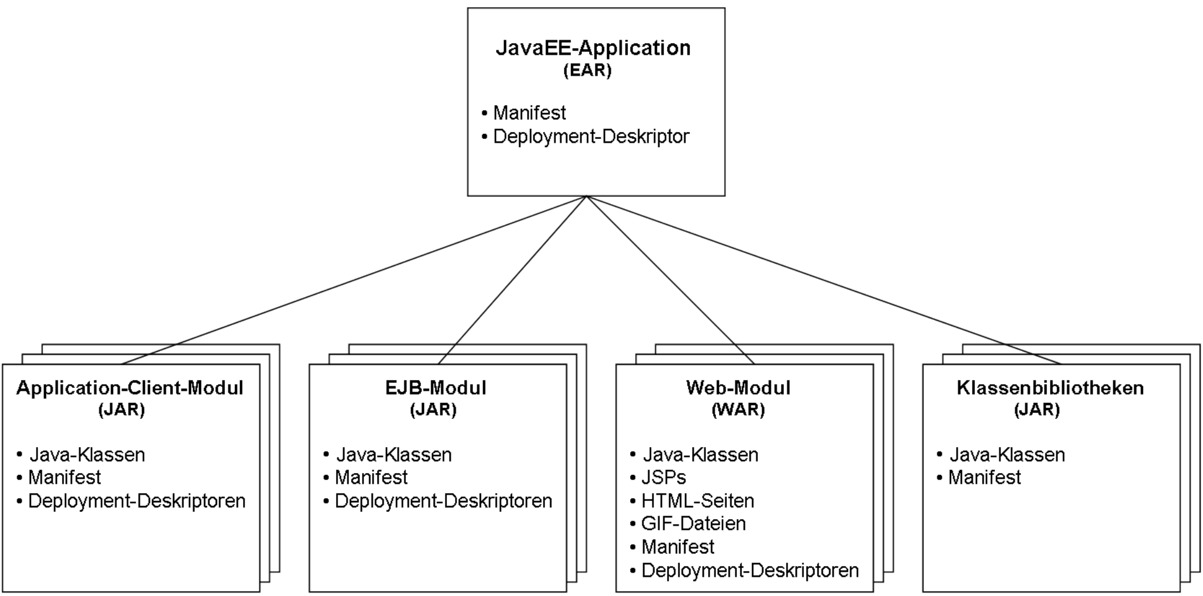


Abbildung 2.2.: Aufbau einer EAR-Datei

Die bereits erläuterten drei JavaEE-Komponentenarten spiegeln sich auch im Aufbau der Applikation wieder. In einem Enterprise Archiv sind beliebig viele Application-Client-, Web- und EJB-Module enthalten. Diese liegen wiederum in Form von Java Archiven (JARs) oder Web Archiven (WARs) vor.

Darüber hinaus enthält eine EAR-Datei auch weitere Java-Klassenbibliotheken (als JARs), die von den verschiedenen Modulen benötigt werden.

Ein viertes JavaEE-Modul ist das Connector- oder Resource-Adapter-Modul, was in den hier betrachteten Applikationen jedoch nicht vorkommt und daher auch nicht weiter behandelt wird.

Da der Aufbau der drei anderen Modularten für diese Arbeit eine wichtige Rolle spielt, werden deren Bestandteile im Folgenden noch etwas genauer beschrieben:

- *Application-Client-Modul:*
  - eine Main-Klasse
  - weitere Java-Klassen
  - Manifest zur Definition der Main-Klasse und des Class-Paths
  - Deployment-Deskriptor `application-client.xml`
  - (optionale) Application-Server-spezifische Deployment-Deskriptoren:
    IBM Websphere Application Server:
    * `ibm-application-bnd.xmi`
    * `ibm-application-ext.xmi`

    JBoss Application Server:
    * `jboss-client.xml`
- *EJB-Modul:*
  - mehrere Enterprise JavaBeans, die jeweils aus einer EJB-Klasse sowie einem Home- und einem Remote-Interface bestehen

  - weitere Java-Klassen
  - Manifest zur Definition des Class-Paths
  - Deployment-Deskriptor `ejb-jar.xml`
  - (optionale) Application-Server-spezifische Deployment-Deskriptoren:
    IBM Websphere Application Server:
    * `ibm-ejb-jar-bnd.xmi`
    * `ibm-ejb-jar-ext.xmi`

    JBoss Application Server:
    * `jboss.xml`
    * `jbosscmp-jdbc.xml`
- *Web-Modul:*
  - Java-Servlet-Klassen
  - JavaServer Pages
  - weitere Java-Klassen
  - Manifest zur Definition des Class-Paths
  - HTML- und GIF-Dateien
  - Deployment-Deskriptor `web.xml`
  - (optionale) Application-Server-spezifische Deployment-Deskriptoren:
    IBM Websphere Application Server:
    * `ibm-web-bnd.xmi`
    * `ibm-web-ext.xmi`

    JBoss Application Server:
    * `jboss-web.xml`

Jedes Modul besitzt mindestens einen so genannten Deployment-Deskriptor. Dies sind Dateien im XML-Format, die Informationen über den Aufbau des Moduls enthalten.

Neben den vorgeschriebenen (`application-client.xml`, `ejb-jar.xml` bzw. `web.xml`) können auch zusätzliche Deskriptoren enthalten sein, in denen Parameter konfiguriert werden, die von der verwendeten Implementierung des JavaEE-Application-Servers benötigt werden. In obiger Auflistung sind beispielhaft die Deployment-Deskriptoren der in dieser Arbeit betrachteten Server IBM Websphere und JBoss angegeben.

Auf den Inhalt dieser Dateien wird in Kapitel 3 noch genauer eingegangen.

Zum Abschluss dieses Abschnitts werden die Enterprise JavaBeans noch etwas näher betrachtet. Es existieren drei unterschiedliche Typen:

- *Entity-Beans*, die die persistenten Daten des Systems modellieren
- *Session-Beans*, die zur Repräsentation eines Clients innerhalb des JavaEE-Servers verwendet werden
- *Message-Driven-Beans*, die eine asynchrone Kommunikation ermöglichen

Jede Enterprise Bean besteht aus einem *Remote-Interface*, einem *Home-Interface* und der *Enterprise-Bean-Klasse*. Das Remote-Interface definiert die Business-Methoden, die ein Client aufrufen kann, während das Home-Interface Methoden definiert, mit denen es dem Client ermöglicht wird, eine Bean zu erzeugen, zu finden oder zu löschen. Diese beiden Interfaces bilden die Basis für einen Remote-Zugriff auf die Beans. Sie können also auch von anderen Virtuellen Maschinen aus angesprochen werden, so zum Beispiel auch von einem entfernten Rechner.

Die Enterprise-Bean-Klasse stellt nun eine Implementierung dieser Methoden zur Verfügung. Sie implementiert die Interfaces allerdings nicht direkt, d.h. diese Klasse enthält kein Statement der Form `BeanClass` **`implements`** `Interface`. Stattdessen wird an Hand dieser Klasse eine direkte Implementierung der zwei Interfaces automatisch vom System generiert, wodurch sichergestellt ist, dass der Client nicht direkt mit der Bean-Klasse kommuniziert.

2.3. JNDI

Das *Java Naming and Directory Interface (JNDI)* [5] ist eine Programmierschnittstelle für Namens- und Verzeichnisdienste. Über diese werden Bindungen von Objekten an Namen erzeugt, die an einer zentralen Stelle gespeichert und von dort abgerufen werden können.

Im Umfeld der JavaEE-Technologie wird JNDI dazu verwendet, verteilte Objekte, wie zum Beispiel Enterprise Beans, innerhalb des Netzwerkes zu registrieren und diese somit den einzelnen JavaEE-Komponenten für Remote-Aufrufe zur Verfügung zu stellen.

2.4. Reflection

Das Reflection-Modell in Java erlaubt es dem Programmierer, Informationen über die Struktur von Klassen und Objekten abzurufen, die zur Laufzeit von der JVM im Speicher gehalten werden. Dabei können zum Beispiel Erkenntnisse über die Methoden einer Klasse wie deren Argument- und Rückgabetypen gewonnen werden.

Außerdem ermöglicht es dieses Konzept, dass Programme Objektinstanzen von für sie zur Compilezeit unbekannten Klassen erzeugen können. Genau dies findet Anwendung innerhalb von TRAVIC Retail. Daher soll im Folgenden die prinzipielle Vorgehensweise einer solchen Instanziierung veranschaulicht werden:

1. Bestimmung eines Objekts vom Typ `java.lang.Class`, das Strukturinformationen über die gewünschte Klasse enthält. Dazu wird der Methode `forName` einfach der Name der gesuchten Klasse in Form eines `java.lang.String` übergeben, wie hier am Beispiel einer Klasse `myPackage.MyClass` verdeutlicht wird:

   ```
   Class c = Class.forName(„myPackage.MyClass“);
   ```

2. Erzeugung eines Objekts vom Typ `java.lang.Object` aus dem `java.lang.Class`-Objekt durch Aufruf der Methode `newInstance()`.

   ```
   Object o = c.newInstance();
   ```

3. Dieses Objekt `o` ist eine Instanz der Klasse `myPackage.MyClass`, sofern diese Klasse zur Laufzeit im Speicher der JVM gehalten wird. Nun braucht nur noch ein Type-Casting auf die entsprechende Klasse durchgeführt werden:

   ```
   myPackage.MyClass myClass = (myPackage.MyClass) o;
   ```

2.5. XML

Wie in Abschnitt 2.2 erläutert, handelt es sich bei den Deployment-Deskriptoren der JavaEE-Module um Dateien im XML-Format. Daher werden im Folgenden einige grundlegende Elemente der XML-Technologie vorgestellt, die für diese Arbeit von Bedeutung sind.

XML (Extensible Markup Language) ist eine Auszeichnungssprache zur Darstellung strukturierter Daten innerhalb von Textdateien. Die Grammatik eines XML-Dokuments kann mit Hilfe von Schemasprachen wie *DTD (Document Type Definition)* oder *XML Schema* [6] angegeben werden. Letztere ist selbst in XML geschrieben und wesentlich leistungsfähiger als die DTD. Um hinsichtlich eines solchen XML Schemas als korrekt akzeptiert zu werden, muss ein XML-Dokument den vom Schema definierten Regeln genügen. Das hierbei überprüfte XML-Dokument wird als Instanzdokument bezeichnet.

JAXB

JAXB (Java Architecture for XML Binding) [7] ist eine Schnittstelle, die eine Bindung zwischen XML Schemas und Java-Klassen ermöglicht. So ist es hierdurch auf einfache Art und Weise möglich, XML-Instanzdokumente einzulesen und daraus einen Java-Objekte-Baum zu erzeugen, ohne dass der Programmierer dabei selbst das XML-Dokument parsen muss.

Der typische Ablauf eines JAXB-Prozesses sieht wie folgt aus:

1. Generierung von Java-Klassen aus einem XML-Schema
2. Erzeugung von Java-Objekten aus einem XML-Instanzdokument
3. gegebenenfalls Modifikation der erzeugten Objekte
4. Speichern der veränderten Java-Objekte in neuem XML-Instanzdokument

2.6. Ant

Apache Ant ist auf Java basierendes, systemunabhängiges Werkzeug zum automatisierten Erzeugen von Programmen aus Quelltext.

Als Basis dient die XML-Datei `build.xml`, die so genannte Build-Datei. In dieser können verschiedene Targets definiert werden, die mit Funktionen in Programmiersprachen vergleichbar sind. Diese enthalten wiederum einzelne Tasks, die nacheinander abgearbeitet werden. So zum Beispiel `javac` zum Kompilieren von Java-Quellcode, `copy` zum Kopieren von Dateien oder `zip` zum Erstellen einer ZIP-Datei.

Letzteres ist der Hauptanwendungspunkt von Ant innerhalb dieser Diplomarbeit.

3. Analyse von Abhängigkeiten zwischen JavaEE-Komponenten

In diesem und dem folgenden Kapitel wird vorgestellt, wie mit Hilfe einer Analyse der Abhängigkeitsstruktur einer JavaEE-Applikation aus dieser eine neue Anwendung generiert werden kann. Die praktische Umsetzung erfolgt dabei selbst in Form eines Java-Programms.

3.1. Konzept

Bevor das entwickelte Analysewerkzeug präsentiert wird, soll dieser Abschnitt zunächst noch einmal die Ausgangssituation und die genaue Zielsetzung präzisieren.

3.1.1. Ausgangssituation

Den Ausgangspunkt der Analyse bildet eine JavaEE-Anwendung, genauer eine Auslieferung des PPI-Produkts TRAVIC Retail, die in Form einer EAR-Datei vorliegt. Der Nutzer wählt nun eine Teilmenge der JavaEE-Komponenten dieser Applikation aus, die in die neu zu erstellende übernommen werden sollen. Diese Komponenten können Enterprise JavaBeans, Application-Clients oder Web-Komponenten sein. Ebenso soll es möglich sein, anzugeben, ob bestimmte Komponenten auf keinen Fall übernommen werden dürfen.

An Hand dieser Informationen soll dann eine JavaEE-Anwendung erzeugt werden, die aus eben diesen JavaEE-Komponenten sowie allen davon benötigten besteht. Dabei ist es das Ziel, auch nur gerade diejenigen Java-Klassen in die neue Applikation zu übernehmen, die von diesen Komponenten gebraucht werden.

Das Ergebnis wird wieder ein EAR sein, das einem Auslieferungsumfang von TRAVIC Retail entspricht, der gerade die Funktionalität zur Verfügung stellt, die der Nutzer durch seine Eingabe gewünscht hat.

Zusammengefasst:

Gegeben:

- EAR-Datei, die eine JavaEE-Applikation enthält
- Auswahl von JavaEE-Komponenten dieser Applikation

Gesucht:

- Minimale Menge der von der Auswahl benötigten Programmkomponenten

3.1.2. Vorgehensweise

Zur Erzeugung einer neuen JavaEE-Applikation ist eine genaue Analyse der Abhängigkeitsstruktur der Originalanwendung notwendig. Diese erfolgt auf zwei Ebenen, die weitestgehend unabhängig voneinander behandelt werden können. Abbildung 3.1 veranschaulicht die verschiedenen Ebenen der Abhängigkeiten.

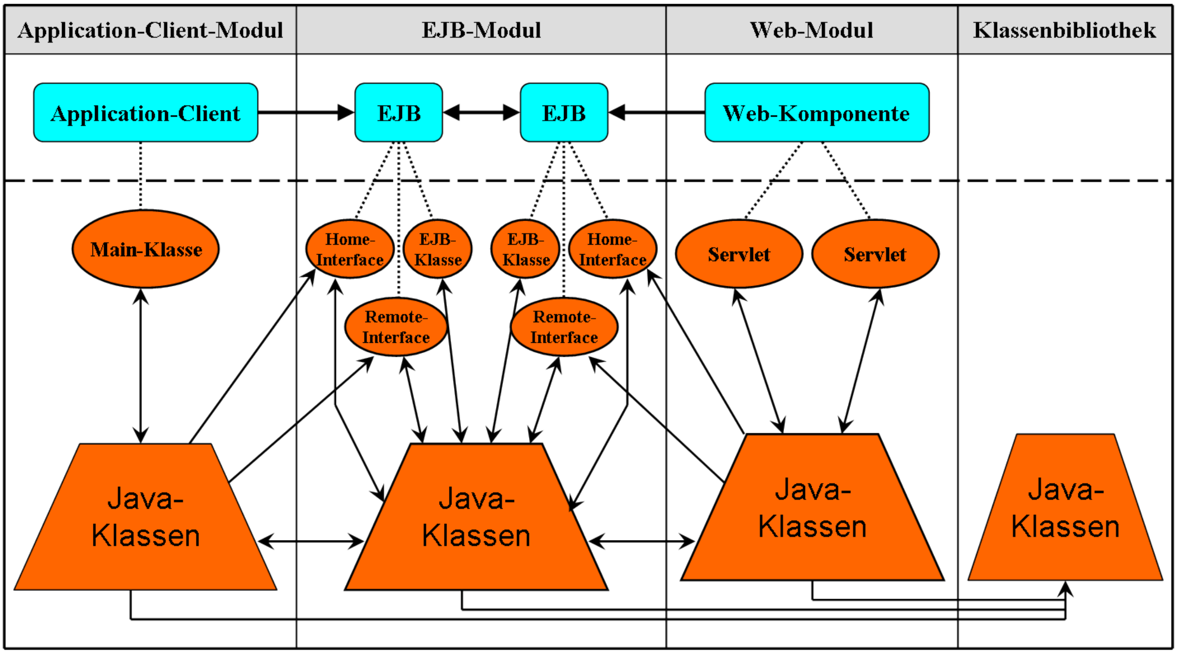


Abbildung 3.1.: Ebenen der Abhängigkeiten

Die obere Schicht zeigt das Zusammenspiel der JavaEE-Komponenten. Auffällig hierbei ist, dass Abhängigkeiten stets zu Enterprise Beans hinführen, sei es von einem Application-Client, einer Web-Komponente oder einer anderen EJB aus.

In einem ersten Schritt gilt es also, die Abhängigkeiten der JavaEE-Komponenten aufzulösen. Als Zwischenergebnis erhält man so all diejenigen Komponenten, die am Ende die JavaEE-Anwendung bilden werden.

Anschließend wird die untergeordnete Ebene der Java-Klassen betrachtet. Dazu ist es zunächst erforderlich, die Klassen zu bestimmen, über die die JavaEE-Komponenten angesteuert werden. Wie Abbildung 3.1 zu entnehmen ist, sind dies im Falle einer Enterprise Bean zum Beispiel die EJB-Klasse, das Home- und das Remote-Interface.

Im zweiten Teil der Analyse werden dann die von diesen Klassen benötigten Java-Klassen bestimmt. Solche Abhängigkeiten können innerhalb desselben Moduls wie die analysierte Klasse auftreten, aber auch zu Klassen aus anderen Archiven, sei es ein anderes Modul oder eine Klassenbibliothek. Entscheidend dafür ist, dass das entsprechende Archiv im Klassenpfad des aufrufenden enthalten ist. Dieser Klassenpfad dient somit als Suchraum für abhängige Klassen.

Sind alle Abhängigkeiten bestimmt, so können die benötigten Java-Archive gemäß der aus ihnen erforderlichen Klassen aktualisiert und zu einer neuen Anwendung zusammengesetzt werden.

3.1.3. Einschränkungen

Zum Abschluss dieses Abschnitts werden nun noch einige einschränkende Anforderungen an die zu analysierende Applikation gestellt. Die meisten davon sind notwendig, da es sich hier um eine statische Analyse handelt. Sie orientieren sich in erster Linie an TRAVIC Retail, da die hier vorgestellte Arbeit dort Anwendung findet.

Die Anforderungen an die untersuchten Applikationen sind:

- *Der Klassenpfad eines Java-Archivs wird durch einen Eintrag im Manifest des Archivs definiert.*

  Im Allgemeinen wäre es auch möglich, diesen zum Beispiel als Kommandozeilenparameter beim Aufruf des Archivs zu übergeben, womit der Klassenpfad erst zur Laufzeit bekannt wäre. Da er aber zur Definition des Suchraums für abhängige Klassen gebraucht wird, muss er schon vorher verfügbar sein.
- *Die Verwendung von Reflection erfolgt nach speziellen Mustern.*

  Reflection erlaubt einen hohen Grad an Dynamisierung, weshalb bei einer statischen Analyse eine gewisse Problematik entsteht. Daher ist es erforderlich, dass die Nutzung von Reflection nach bekannten Schemata abläuft. Wie diese im Detail aussehen, wird in Abschnitt 4.4.1 erläutert.
- *Als Laufzeitumgebung der zu analysierenden JavaEE-Anwendung dienen nur der IBM Websphere Application Server oder der JBoss Application Server*

  Bei TRAVIC Retail werden nur die zwei oben genannten Application-Server verwendet, weshalb in dieser Arbeit auch nur diese behandelt werden. Die Erweiterung auf andere ist aber durchaus möglich.

3.2. Vorbereitungen

Bevor mit der eigentlichen Analyse begonnen werden kann, müssen noch einige Vorbereitungen getroffen werden. Diese stehen in Zusammenhang mit den Deployment-Deskriptoren der Anwendung, d.h. den XML-Dateien, über die eine JavaEE-Anwendung konfiguriert wird. Um den Inhalt dieser Dateien auszulesen, zu verändern und wieder in eine neue Datei zu schreiben, soll das Programm *JAXB* verwendet werden, welches in Abschnitt 2.5 bereits vorgestellt wurde.

Die Definitionen der Grammatiken der Deskriptoren liegen zunächst in Form von DTDs vor (vgl. [8]). Sie können aber leicht in XML-Schemata transformiert werden, welche JAXB als Eingabe erwartet.

Nun werden einmalig aus den Schemata Java-Klassen zu deren Repräsentation erzeugt. Das Ergebnis dieses Prozesses soll nun am Beispiel des Schemas zum Deployment-Deskriptor `application.xml` einer EAR-Datei veranschaulicht werden. Dabei wird jedoch aus Gründen der Übersichtlichkeit nur ein Ausschnitt dieses Schemas betrachtet:

```
<?xml version="1.0" encoding="UTF-8">
<xs:schema
   xmlns:xs="http://www.w3.org/
   2001/XMLSchema"
   elementFormDefault="qualified">
  <xs:import
     namespace="http://www.w3.org/
     XML/1998/namespace"/>

  <xs:complexType name="application">
    <xs:sequence>
      <xs:element ref="icon"
          minOccurs="0"/>
      <xs:element
          ref="display-name"/>
      <xs:element ref="description"
          minOccurs="0"/>
      <xs:element ref="module"
          maxOccurs="unbounded"/>
      <xs:element
          ref="security-role"
          minOccurs="0"
          maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id"
       type="xs:ID"/>
  </xs:complexType>

  <xs:element name="application"
     type="application"/>

  <xs:complexType name="connector"
     mixed="true">
    <xs:attribute name="id"
       type="xs:ID"/>
  </xs:complexType>

  <xs:element name="connector"
     type="connector"/>
```

```
<xs:complexType
    name="context-root"
    mixed="true">
  <xs:attribute name="id"
      type="xs:ID"/>
</xs:complexType>

<xs:element name="context-root"
    type="context-root"/>

<xs:complexType name="ejb"
    mixed="true">
  <xs:attribute name="id"
      type="xs:ID"/>
</xs:complexType>

<xs:element name="ejb" type="ejb"/>

<xs:complexType name="java"
    mixed="true">
  <xs:attribute name="id"
      type="xs:ID"/>
</xs:complexType>

<xs:element name="java"
    type="java"/>

<xs:complexType name="module">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="connector"/>
      <xs:element ref="ejb"/>
      <xs:element ref="java"/>
      <xs:element ref="web"/>
    </xs:choice>
    <xs:element ref="alt-dd"
        minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="id"
      type="xs:ID"/>
</xs:complexType>

<xs:element name="module"
    type="module"/>

<xs:complexType name="web">
  <xs:sequence>
    <xs:element ref="web-uri"/>
    <xs:element
        ref="context-root"/>
  </xs:sequence>
  <xs:attribute name="id"
      type="xs:ID"/>
</xs:complexType>

<xs:element name="web" type="web"/>

<xs:complexType name="web-uri"
    mixed="true">
  <xs:attribute name="id"
      type="xs:ID"/>
</xs:complexType>

<xs:element name="web-uri"
    type="web-uri"/>

</xs:schema>
```

Das mit Hilfe von *Altova UModel* [1] erzeugte UML-Diagramm in Abbildung 3.2 zeigt die Struktur der von JAXB erzeugten Klassen. Auf die Darstellung der Methoden der Klassen wird hierbei verzichtet, da sich diese nur auf Getter und Setter zu den angegebenen Feldern beschränken, um die dort gehaltenen Werte zu lesen oder zu setzen.

Es ist zu erkennen, dass für jeden im XML-Schema definierten `complexType` eine eigene Klasse erzeugt wurde. Die Attribute und Elemente des XML-Dokuments finden sich in den Feldern der Klasse wieder.

Analog werden auch Datenstrukturen für die Deskriptoren `application-client.xml`, `web.xml` und `ejb-jar.xml` generiert.

3.3. Bestimmung der Komponenten der Applikation

Gegeben sei nun eine JavaEE-Applikation in Form einer EAR-Datei. Der erste Schritt ist es, den Deployment-Deskriptor `application.xml` der Anwendung zu analysieren, um zu bestimmen, aus welchen Modulen diese Applikation besteht. Dieser befindet sich innerhalb der EAR-Datei, muss also zunächst entpackt werden.

[1] `http://www.altova.com/products/umodel/uml_tool.html`

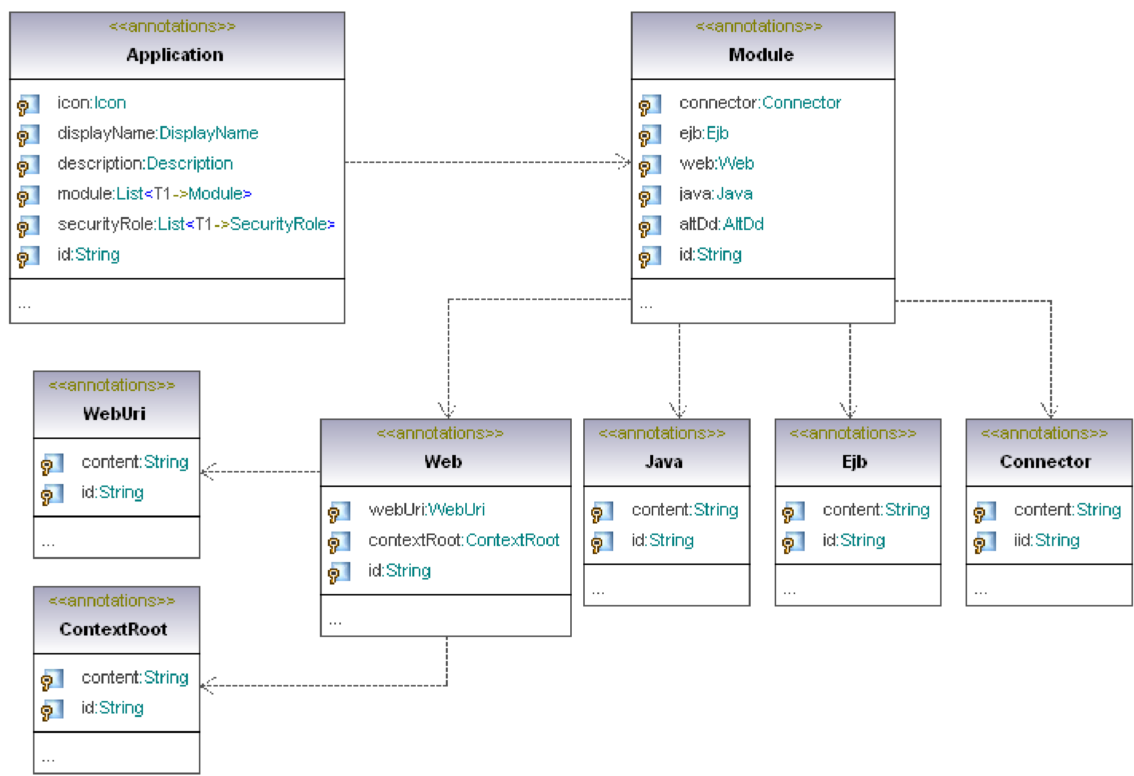


Abbildung 3.2.: Diagramm der von JAXB aus dem Schema zu `application.xml` erzeugten Klassen

Dazu wird die Klasse `java.util.jar.JarFile` verwendet, mit deren Hilfe Einträge von JAR-Dateien oder Dateien, die auf diesem Archiv aufbauen wie eben EARs oder WARs, gelesen werden können.
Folgendes Code-Fragment liefert so den gewünschten Deployment-Deskriptor in Form eines `java.io.InputStream`:

```
JarFile earFile = new JarFile("NameOfEarFile");
ZipEntry entry = earFile.getEntry("META-INF/application.xml");
InputStream in = earFile.getInputStream(entry);
```

Dieser `InputStream` kann nun in eine XML-Datei ausgegeben werden.
Es folgt ein Beispiel für eine solche `application.xml`-Datei.

```
<?xml version="1.0" encoding="UTF-8"?>
<application id="Application_ID">
  <display-name>travicEnterpriseAppA4</display-name>
  <module id="EjbModule_1106298014376">
    <ejb>Server.jar</ejb>
  </module>
  <module id="EjbModule_1106298014377">
    <ejb>StatusProtocol.jar</ejb>
  </module>
  <module id="JavaClientModule_1106298014376">
    <java>TravicAdmin.jar</java>
  </module>
  <module id="WebModule_1106298211594">
    <web>
      <web-uri>PinTanBroker.war</web-uri>
```

```
        <context-root>travic/pintan</context-root>
      </web>
    </module>
  </application>
```

In den <module>-Tags werden die Module definiert, aus denen sich die Anwendung zusammensetzt. In der nächsten Hierarchieebene des Dokuments wird die Art des Moduls festgelegt, sei es ein EJB-, Web- oder Java-Application-Modul. Ebenso ist hier der Name der Archivdatei gespeichert, in der das entsprechende Modul enthalten ist.

Unter Verwendung von JAXB wird der Inhalt der XML-Datei in die zuvor generierte Datenstruktur eingelesen, sodass die Information, welche Module in der Applikation enthalten sind, nun leicht abrufbar und auch veränderbar ist.

Anschließend müssen diese Module genauer untersucht werden. Die JARs aus der EAR-Datei, die lediglich Klassenbibliotheken enthalten, werden in dieser Phase der Analyse zwar noch nicht benötigt, dennoch werden auch sie hier schon behandelt. Das Ziel dabei ist es, alle wichtigen Informationen eines jeden Java-Archivs in einer Datenstruktur `datastructures.Archive` zu speichern. Diese dient als Grundlage der Analyse von Abhängigkeiten aller Ebenen, weshalb es sich anbietet, sie schon an dieser Stelle so weit wie möglich mit Inhalt zu befüllen.

Die Informationen, die in der Klasse `datastructures.Archive` gespeichert werden, sind:

- Name des Archivs
- Speicherposition des Archivs
- Typ des Archivs (bestimmtes Modul oder Klassenbibliothek)
- enthaltene Java-Klassen
- enthaltene Java-Archive
- Klassenpfad

Die Klasse `datastructures.Archive` dient dabei als Basis zur Darstellung von Archiven. Für die unterschiedlichen Modularten und auch die EAR-Datei selbst müssen zusätzlich Informationen über die Deployment-Deskriptoren gespeichert werden. Daher wird für jede Modulart eine Klassen-Erweiterung von `datastructures.Archive` erstellt, die unter anderem auch die per JAXB erstellte Datenstruktur für diesen Deskriptor umfasst.

In einem ersten Schritt müssen die Archive zunächst aus der EAR-Datei entpackt werden. Dies geschieht, wie oben bereits geschildert, über die Klasse `java.util.jar.JarFile`. Diese bietet noch einen weiteren Vorteil: Sie ermöglicht den direkten Zugriff auf das Manifest der Archivdatei, sodass der für die spätere Analyse der Klassenabhängigkeiten benötigte Klassenpfad schon hier leicht bestimmt werden kann:

```
JarFile jarFile = new JarFile("NameOfJarFile");
Manifest manifest = jarFile.getManifest();
Attributes attrs = manifest.getMainAttributes(); // Einträge im Manifest
String classPath = attrs.getValue("Class-Path");
```

Für die Module der JavaEE-Anwendung ist jetzt eine weitergehende Analyse der Deployment-Deskriptoren notwendig. Es werden zunächst die vorgeschriebenen betrachtet, d.h. `ejb-jar.xml`, `web.xml` und `application-client.xml`.

EJB-Module setzen sich aus mehreren einzelnen Enterprise JavaBeans zusammen. Diese werden in der Datei `ejb-jar.xml` definiert. Ferner werden hier, wie auch in den anderen beiden Deskriptoren, Referenzen zu anderen benötigten Beans angegeben.

In Abbildung 3.3 ist der Ausschnitt aus einem Deployment-Deskriptor für ein EJB-Modul illustriert.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar id="ejb-jar_ID">

  <enterprise-beans>
    <session id="MessageController">
      <ejb-name>MessageController</ejb-name>
      <home>messagecontroller.api.MessageControllerHome</home>
      <remote>messagecontroller.api.MessageController</remote>
      <ejb-class>messagecontroller.internal.MessageControllerEJB</ejb-class>
      <ejb-ref id="EjbRef_1058867825157">
        <ejb-ref-name>ejb/travic/retail/server/DialogContext</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>dialogcontext.common.DialogContextHome</home>
        <remote>dialogcontext.common.DialogContext</remote>
      </ejb-ref>
      [...]
    </session>

    <entity id="DialogContext">
      <ejb-name>DialogContext</ejb-name>
      <home>dialogcontext.common.DialogContextHome</home>
      <remote>dialogcontext.common.DialogContext</remote>
      <ejb-class>dialogcontext.internal.DialogContextEJB</ejb-class>
      [...]
    </entity>
  </enterprise-beans>

</ejb-jar>
```

Abbildung 3.3.: Beispiel: ejb-jar.xml

Innerhalb des Tags <enterprise-beans> werden die EJBs dieses Moduls definiert. Die Bezeichnungen der Elemente richten sich dabei nach dem Typ der Beans, d.h. Entity-, Session- oder Message-Driven-Bean. Jede Bean bekommt einen eindeutigen Namen zugewiesen, der im Tag <ejb-name> gespeichert wird. Ferner werden die drei Java-Klassen, die die Bean bilden (EJB-Klasse, Home- und Remote-Interface), in entsprechenden Elementen angegeben.

Damit sind nun alle Komponenten der JavaEE-Anwendung bestimmt und es kann damit begonnen werden, deren Abhängigkeiten untereinander zu ermitteln.

3.4. Bestimmung der direkten Abhängigkeiten

Für die Bestimmung der Beziehungen zwischen den einzelnen JavaEE-Komponenten sind wieder die Deployment-Deskriptoren von entscheidender Bedeutung.

Zur Veranschaulichung kann hier noch einmal Abbildung 3.3 herangezogen werden. Das entscheidende XML-Element ist hier <ejb-ref>, wo eine Referenz zu einer benötigten Bean definiert wird. Wichtig ist hierbei <ejb-ref-name>, wo die Bezeichnung der referenzierten Bean vermerkt ist. Sowohl im <ejb-ref>- als auch im Bean-Element kann zusätzlich ein ID-Attribut angegeben werden, das im Folgenden noch eine wesentliche Rolle spielen wird. Dieses Attribut muss aber nicht zwingend existieren. In den beiden anderen Deskriptoren application-client.xml und web.xml erfolgt die Definition von Abhängigkeiten auf analoge Art und Weise.

Die Hauptproblematik besteht nun darin, eine Beziehung dieser Referenzen zu den Beans herzustellen, auf die sich diese beziehen. Denn wie im Beispiel erkennbar ist, muss dies weder über den Bean-Namen noch über die ID direkt möglich sein. So wird in der Session-Bean „MessageController“ eine Referenz zu der Entity-Bean „DialogContext“ definiert. Der im `<ejb-ref>`-Element angegebene Name „ejb/travic/retail/server/DialogContext“ stimmt aber nicht dem Inhalt des Tags `<ejb-name>` des `<entity>`-Elements zur Definition der „DialogContext“-Bean überein. Gleiches gilt für die jeweiligen IDs „EjbRef_1058867825157“ und „DialogContext“.

Die Lösung, die auch innerhalb des Application Servers, also der Laufzeitumgebung, verwendet wird, heißt JNDI (vgl. 2.3). Dieser Namensdienst erlaubt es, die EJBs in einem Netzwerk zu registrieren, sodass alle JavaEE-Komponenten auf diese zugreifen können. Die Bereitstellung dieser Funktionalität ist in der JavaEE-Umgebung die Aufgabe des Application-Servers. In den unterschiedlichen Implementierungen solcher Server erfolgt die Bindung der EJBs an eindeutige JNDI-Namen jedoch nicht immer auf die gleiche Art und Weise. Im Folgenden wird die Realisierung beim *IBM Websphere Application Server* und beim *JBoss Application Server* vorgestellt. Eine Erweiterung auf andere Application Server ist aber durchaus möglich.

Bei IBM Websphere gibt es einen zusätzlichen Deskriptor für jedes Modul, in dem die JNDI-Bindungen definiert werden. Diese sind: `ibm-ejb-jar-bnd.xmi`, `ibm-application-client-bnd.xmi` und `ibm-web-bnd.xmi`. Wie man erkennt, handelt es sich hierbei um *XMI (XML Metadata Interchange)*-Dateien, die aber auf der XML-Syntax aufbauen und somit genauso behandelt werden können. Sie liegen im selben Verzeichnis wie die zuvor betrachteten, allgemeingültigen Deskriptoren.

Es folgt ein Auszug aus dem Bindings-Deskriptor für das EJB-Modul aus Beispiel 3.3.

```
<ejbbnd:EJBJarBinding [...]>
  <ejbJar href="META-INF/ejb-jar.xml#ejb-jar_ID"/>

  <ejbBindings xmi:id="MessageController_Bnd"
      jndiName="ejb/travic/retail/server/MessageController">
    <enterpriseBean xmi:type="ejb:Session"
        href="META-INF/ejb-jar.xml#MessageController"/>
    <ejbRefBindings xmi:id="EjbRefBinding_1058867825157"
        jndiName="ejb/travic/retail/server/DialogContext">
      <bindingEjbRef href="META-INF/ejb-jar.xml#EjbRef_1058867825157"/>
    </ejbRefBindings>
  </ejbBindings>

  <ejbBindings xmi:id="DialogContext_Bnd"
      jndiName="ejb/travic/retail/server/DialogContext">
    <enterpriseBean xmi:type="ejb:ContainerManagedEntity"
        href="META-INF/ejb-jar.xml#DialogContext"/>
  </ejbBindings>
</ejbbnd>
```

Innerhalb des Elements `<ejbBindings>` wird zunächst über das Attribut `jndiName` der JNDI-Name einer EJB des Moduls angegeben. Welche Bean damit gemeint ist, steht im Tag `<enterpriseBean>`. Damit kann diese nun von allen Komponenten der JavaEE-Applikation über diesen JNDI-Namen referenziert werden. Ebenfalls wird hier der JNDI-Name für die Referenz zur Bean „DialogContext“ angegeben. Dieser muss natürlich identisch zu dem sein, der bei der Definition der Bindungen dieser Bean im zweiten `ejbBindings`-Element gewählt wurde.

Die Verbindung der beiden vorgestellten Deskriptoren, genauer die Verbindung zwischen den darin definierten Beans, erfolgt über das schon erwähnte ID-Attribut. In `ejb-jar.xml` enthalten die Tags

<ejb-ref>, <entity> und <session> eine ID, die sich innerhalb des Attributs href der Tags <enterpriseBean> und <bindingEjbRef> wiederfindet (der Teil nach „#“).

Die Deskriptoren ibm-application-client-bnd.xmi und ibm-web-bnd.xmi sind analog zu dem hier beschriebenen aufgebaut. Sie beschränken sich allerdings auf die Definition der Bindungen von EJB-Referenzen.

Problematisch ist jedoch, dass weder DTDs noch XML-Schemata für all diese Deskriptoren verfügbar sind. Ebenso spärlich ist die Dokumentation von deren Grammatik. Der Grund dafür mag sein, dass zur Erstellung der Deskriptoren ein Editor bereitgestellt wird, sodass der Programmierer die genaue Syntax gar nicht kennen muss. Aus diesem Grund ist auch ein Einsatz von JAXB hier nicht möglich.

Für diese Arbeit ist es aber ausreichend zu wissen, wo die JNDI-Bindungen definiert werden, d.h. in erster Linie in welchen XML-Elementen.

Die Gewinnung der benötigten Informationen erfolgt hier durch Parsen des Deskriptors mittels JDOM [9], einer Java-API zur Manipulation von XML-Dokumenten. Hierbei wird das Dokument eingelesen und in einer entsprechenden Datenstruktur abgelegt. Der Unterschied zu JAXB ist, dass sich diese Datenstruktur auf beliebige XML-Dokumente bezieht, die keinem besonderen XML-Schema genügen müssen.

Der Zugriff auf das JNDI-Attribut des ersten XML-Elements <ejbBindings> aus obigem Beispiel erfolgt dann so:

```
Document doc = new SAXBuilder().build("NameOfDocument"); // Parsen
Element element = doc.getRootElement();                  // Wurzel-Element
List list = element.getChildren("ejbBindings");          // <ejbBindings>-Elemente
Element first = (Element) list.get(0);
String jndiName = first.getAttributeValue("jndiName");
```

Das erzeugte Objekt des Typs org.jdom.Document kann genauso wie bei JAXB verändert und als neues XML-Dokument exportiert werden.

Beim JBoss-Application-Server erfolgt die Definition der JNDI-Bindungen ebenfalls über spezielle Deskriptoren: jboss.xml für EJB-Module, jboss-web.xml für Web- und jboss-client.xml für Application-Client-Module (siehe auch [10]).

Ein Auszug des JBoss-Bindings-Deskriptors zu dem bisher betrachteten Beispiel 3.3 ist hier zu sehen:

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>MessageController</ejb-name>
      <jndi-name>ejb/travic/retail/server/MessageController</jndi-name>
      <ejb-ref>
        <ejb-ref-name>ejb/travic/retail/server/DialogContext</ejb-ref-name>
        <jndi-name>ejb/travic/retail/server/DialogContext</jndi-name>
      </ejb-ref>
    </session>
    <entity>
      <ejb-name>DialogContext</ejb-name>
      <jndi-name>ejb/travic/retail/server/DialogContext</jndi-name>
    </entity>
  </enterprise-beans>
</jboss>
```

Wie leicht zu erkennen ist, werden die JNDI-Namen durch ein gleichnamiges Tag definiert. Die Referenzierung zur zugehörigen Bean erfolgt dabei über den Beannamen und nicht wie bei IBM Websphere über die ID. Die XML-Elemente dafür sind <ejb-name> und <ejb-ref-name>.

In `jboss-web.xml` und `jboss-client.xml` wird sich analog zu oben auf `<ejb-ref>`-Tags beschränkt.

In beiden beschriebenen Fällen ist es nun aber auch möglich, dass zu einigen Beans keine Bindungen definiert wurden oder die Bindings-Deskriptoren gänzlich fehlen. In diesem Fall verwendet der Application-Server Default-Namen, die gerade den angegebenen Namen in den Tags `<ejb-name>` bzw. `<ejb-ref-name>` entsprechen.

Weiterhin kann es auch sein, dass Referenzen zu EJBs existieren, die in der Anwendung gar nicht enthalten sind. Solche Einträge können einfach aus den jeweiligen Datenstrukturen gelöscht werden.

Noch nicht erklärt wurde, wie festgestellt wird, für welchen Application-Server die untersuchte Anwendung konfiguriert wurde. Dies erfolgt gerade über die Dateinamen der Application-Server-spezifischen Deployment-Deskriptoren.

Auf die hier vorgestellte Art und Weise ist es damit möglich, alle notwendigen Informationen über die Abhängigkeitsstruktur zwischen den einzelnen Komponenten einer JavaEE-Applikation zu bestimmen. Die direkten Beziehungen und natürlich auch die JNDI-Bindungen werden in der schon vorgestellten Datenstruktur `datastructures.Archive` gespeichert, sodass sie später leicht abrufbar sind. Wichtig ist hierbei, dass man sich merkt, auf welche Weise die Bindungen definiert wurden, d.h. ob über die ID oder den Namen der EJBs.

Zusätzlich wird an zentraler Stelle für jeden definierten JNDI-Namen eine Referenz zu dem Java-Archiv gespeichert, in dem die entsprechende Bean gespeichert ist. Dies hat den Vorteil, dass Beziehungen über Modulgrenzen hinweg unmittelbar aufgelöst werden können.

3.5. Erstellung eines Abhängigkeitsgraphen

Bislang wurden lediglich direkte Abhängigkeiten bestimmt. Zur Ermittlung der indirekten bietet es sich an, die bisher ermittelte Struktur in Form eines Graphen darzustellen, gerade auch in Bezug auf die Zielsetzung einer späteren Visualisierung (vgl. Kapitel 5).

Zur Realisierung wird eine Klasse `Node` als Datenstruktur für einen Graphknoten erstellt, die folgenden Inhalt umfasst:

- Name der zu Grunde liegenden JavaEE-Komponente
- Typ der zu Grunde liegenden JavaEE-Komponente
- Liste der benötigten EJBs (die Nachfolgerknoten)

Diese Datenstruktur wird später bei der Visualisierung noch erweitert werden. Für den Moment reicht sie in dieser Form aber aus.

An Hand dieser Knotenmenge ist nun eine komplette Bestimmung der gemäß einer Nutzereingabe benötigten Komponenten einer JavaEE-Anwendung möglich. Dies wird durch eine Tiefensuche auf dem zu Grunde liegenden Graphen realisiert, die in Pseudo-Code wie folgt abläuft:

Algorithmus zur Bestimmung der benötigten JavaEE-Komponenten

Gegeben: Knotenmenge V, Kantenmenge E in Form von Nachfolgerlisten $suc(v)$ für $v \in V$

Eingabe: Knotenmenge $I \subseteq V$

Gesucht: Menge B der von I benötigten JavaEE-Komponenten

1. $B := \emptyset$

2. $for\ all\ v \in I$
   $DFS(v);$

DFS(v):

$if(v \in B)$
   $return;$
$else$
   $B := B \cup v;$
   $for\ all\ s \in suc(v)$
      $DFS(s);$

Die Resultatsmenge B enthält am Ende die von der Nutzereingabe I aus erreichbaren Knoten, die den entsprechenden Tiefensuchenwald bilden. Dies sind genau die benötigten JavaEE-Komponenten.

Was bislang nicht berücksichtigt wurde, ist die Anforderung, dass es dem Nutzer auch möglich sein soll, Komponenten anzugeben, die nicht mit in die zu erstellende Anwendung übernommen werden dürfen. Dies wird dadurch umgesetzt, dass die jeweiligen Komponenten vor der Tiefensuche komplett aus dem Graphen entfernt werden, d.h. sowohl die Knoten selbst, als auch deren Einträge in den Nachfolgerlisten.

3.6. Aktualisierung der Deployment-Deskriptoren

Nachdem nun untersucht wurde, welche JavaEE-Komponenten der Anwendung benötigt werden, müssen die Deployment-Deskriptoren der einzelnen Module angepasst werden.

Zuvor werden jedoch die Einträge der nicht benötigten Module aus der Datei `application.xml` entfernt. Die Module werden in der mit JAXB erzeugten Klassen-Repräsentation in einer Liste gehalten, sodass das entsprechende XML-Element einfach aus dieser entfernt werden kann (vgl. Abbildung 3.2). Die geänderte Datenstruktur kann dann wieder mit Hilfe von JAXB in ein neues XML-Dokument geschrieben werden, was der Deployment-Deskriptor der neu zu erstellenden JavaEE-Applikation sein wird.

Die Aktualisierung der Deskriptoren der Module lässt sich in zwei Bereiche aufteilen:

1. Aktualisierung der EJB-Definitionen in einem EJB-Modul
2. Aktualisierung der EJB-Referenzen

Werden aus einem EJB-Modul nur einige Beans gebraucht, so müssen die Deklarationen der nicht mehr benötigten aus allen Deployment-Deskriptoren entfernt werden. Dies betrifft in `ejb-jar.xml` die jeweiligen XML-Elemente `<entity>`, `<session>` und `<message-driven>` (vgl. Abbildung 3.3), welche komplett aus dem Dokument gelöscht werden. Dies kann analog zu oben wieder durch Entfernen der Einträge aus der Liste in der zugehörigen Java-Klasse erfolgen.

Es existieren noch weitere Stellen in `ejb-jar.xml`, wo eine nicht mehr benötigte Bean vorkommen kann. Hier kann ganz analog vorgegangen werden.

Bei allen drei Modularten müssen zusätzlich noch Referenzen zu gelöschten EJBs aus den Deskriptoren entfernt werden, indem die betroffenen `<ejb-ref>`-Elemente herausgenommen werden.

Die veränderten Datenstrukturen können wieder über JAXB in neue XML-Instanzdokumente exportiert werden.

Bei den speziellen Deskriptoren für den JBoss Application Server kann auf gleiche Art und Weise verfahren werden. Anzumerken ist hier noch, dass das EJB-Modul einen weiteren Deskriptor besitzt, der hier noch nicht aufgeführt wurde: `jbosscmp-jdbc.xml`.

Für die Deployment-Deskriptoren des IBM Websphere Application Servers muss hingegen erneut JDOM benutzt werden, was aber ähnlich abläuft. Auch hier gibt es noch zwei bislang nicht erwähnte, die ebenfalls zu aktualisieren sind: `ibm-ejb-jar-ext.xmi` und `ibm-application-ext.xmi`.

Die Problematik, dass hierfür weder DTDs noch Schemata bekannt sind, tritt an dieser Stelle etwas mehr in den Vordergrund. Denn nun reicht es nicht mehr aus, nur zu wissen, wo die für die Abhängigkeitsanalyse wichtigen Informationen gespeichert sind. Es ist auch notwendig, über alle möglichen Vorkommen der nicht mehr benötigten Komponenten in den Deskriptoren informiert zu sein, um sie dann dort zu entfernen. Durch Analyse von Beispiel-Instanzen der XMI-Dokumente kann zwar ein bestimmter Teil dieses Wissens gewonnen werden, doch die bleibenden Lücken bergen die Gefahr, dass die neu erstellten serverspezifischen Deskriptoren inkonsistent zu den übrigen sind. Dies ist natürlich nicht wünschenswert, hat aber auf die Applikation selbst keinen Einfluss, da hier schlimmstenfalls einige überflüssige Informationen gespeichert sind.

3.7. Übergang zur Klassenebene

Zum Abschluss dieses Abschnitts muss noch der Übergang von der Ebene der JavaEE-Komponenten zur Klassenebene vollzogen werden. Es wird daher für jedes Modul die Menge an Klassen bestimmt, durch die die aus ihm benötigten Komponenten angesprochen werden.

Im Falle eines Application-Clients ist der Einstiegspunkt auf Klassenebene die Main-Klasse des JARs. Diese kann leicht aus dem Eintrag im Manifest bestimmt werden:

```
JarFile jarFile = new JarFile("NameOfJarFile");
Manifest manifest = jarFile.getManifest();
Attributes attrs = manifest.getMainAttributes();
String mainClass = attrs.getValue("Main-Class");
```

Eine Web-Komponente wird durch eine oder mehrere Servlet-Klassen angesprochen. Diese sind im Deployment-Deskriptor `web.xml` definiert:

```
<?xml version="1.0" encoding="UTF-8"?>
  <web-app id="WebApp">
    <servlet>
      <servlet-name>HbciPinTanHttpGate</servlet-name>
      <servlet-class>hbcibroker.pintan.HbciPinTanHttpGate</servlet-class>
    </servlet>
</web-app>
```

Die Servlet-Klassen können dann über die von JAXB erzeugten Klassen ermittelt werden:

```
List<String> result = new LinkedList<String>();
List<Servlet> servlets = webArchive.getWebApp().getServlet();
for(Servlet servlet : servlets)
  result.add(servlet.getServletClass().getContent());
```

Hier lässt sich auch gut erkennen, wie leicht über die Getter-Methoden für `web-app`, `servlet` und `servlet-class` durch den XML-Baum navigiert werden kann.

Auf die gleiche Weise werden die entsprechenden Klassen für Enterprise JavaBeans, die EJB-Klasse sowie das Home- und das Remote-Interface bestimmt.

Alle hier bestimmten Klassen werden nun mit Verweis auf das Java-Archiv, in dem sie liegen, in einer speziellen Datenstruktur gespeichert, die als Eingabe für das nächste Analyselevel dient.

Somit ist die Analyse auf der Ebene der JavaEE-Komponenten abgeschlossen und der Übergang zur Bestimmung der Abhängigkeiten zwischen Java-Klassen geschaffen.

4. Analyse der Abhängigkeiten zwischen Java-Klassen

Die Analyse der Beziehungen zwischen Java-Klassen ist Gegenstand dieses Kapitels. Dazu wird die Implementierung eines Algorithmus vorgestellt, der aufbauend auf einem Parser für `.class`-Dateien die Klassenabhängigkeiten analysiert. Dies erfolgt in zwei verschiedenen Varianten: auf Klassen- und auf Methodenebene. An Hand des Ergebnisses dieser Analyse wird dann eine neue Java-Applikation erzeugt. Den Abschluss dieses Abschnitts bildet ein Vergleich der beiden Analyseebenen an Hand praktischer Beispiele.

4.1. Unterschiedliche Analyseebenen

Bevor mit der Bestimmung der benötigten Klassen begonnen wird, werden in diesem Abschnitt zwei Alternativen für die prinzipielle Vorgehensweise der weiteren Analyse gegenübergestellt.

An Hand dieses Beispiels sollen die beiden Varianten nun miteinander verglichen werden:

```
class A {

    void methodA() {
        B.methodB1();
    }
}


class B {

    void methodB1() {
        C.methodC();
    }

    void methodB2() {
        D.methodD();
    }
}
```

Angenommen, in der zu erstellenden Anwendung wird die komplette Klasse `A` benötigt. Dann stellt sich die Frage, welche anderen Klassen von `A` gebraucht werden. Offensichtlich ist, das `B` dazuzählt, da in der Methode `methodA` `B`'s Methode `methodB1` aufgerufen wird. Nun kann aber differenziert werden, was weiterhin noch gebraucht wird. Dies führt zu den zwei verschiedenen Analyseebenen.

Bei einer Analyse auf *Klassenebene* würden alle Klassen, die innerhalb von `B` verwendet werden, in die Abhängigkeitsmenge eingefügt werden. Im Beispiel wären dies die Klassen `C` und `D` inklusive aller Klassen, die wiederum von ihnen benötigt werden.

Eine genauere Betrachtung des obigen Beispiels lässt aber erkennen, dass Klasse `A` nie die Methode `methodB2` aufruft, d.h. es wird auch nie `methodD` der Klasse `D` angesprochen und `D` somit eigentlich

gar nicht gebraucht. Analysiert man die Abhängigkeiten also auf *Methodenebene*, so wird weder Klasse `D` mit in die Abhängigkeitsmenge übernommen, noch diejenigen Klassen, die `D` selbst benötigt.

Die Analyse auf Methodenebene liefert somit als Ergebnis die minimale Menge an Klassen, die zur Laufzeit innerhalb der Anwendung verwendet werden könnten. Dies war gerade das Ziel dieser Arbeit. Betrachtet man die Abhängigkeiten hingegen auf Klassenebene, so erhält man diejenigen Klassen, die zum Kompilieren des entsprechenden Quellcodes gebraucht werden.

Es ist nicht schwer zu erkennen, dass die letztere Variante die deutlich einfachere und auch schnellere ist, jedoch die ermittelte Resultatsmenge von Java-Klassen in der Regel größer sein wird als eigentlich nötig. Wie die zusätzlichen Herausforderungen bei einer Analyse auf Methodenebene im Detail aussehen, soll nun geklärt werden.

4.1.1. Herausforderungen der Analyse auf Methodenebene

Die Hauptschwierigkeit liegt dabei in der Behandlung von Interfaces und Klassenerweiterungen.

Man betrachte folgendes Beispiel:

```
void myMethod (MyInterface interf) {
    interf.method();
}
```

Eine Methode bekommt hier als Argument ein Interface übergeben und ruft von diesem eine Methode auf. De facto ist es aber so, dass beim Aufruf von `myMethod` eine Implementierung dieses Interfaces übergeben wird, d.h. insbesondere wird nicht die Methode `method` aus dem Interface, sondern selbige aus dessen Implementierung aufgerufen.

Dies führt zu folgender Problematik: Woher weiß man, welche Implementierung sich hinter dem Interface verbirgt? Ferner kommt noch hinzu, dass bei jedem Aufruf von `myMethod` eine unterschiedliche Implementierung verwendet worden sein kann. Um dies herauszufinden, wäre es notwendig, den kompletten Kontrollfluss von dem entsprechenden Methodenaufruf bis zu der Stelle zurückzuverfolgen, an der dieses Interface durch eine Implementierung bedient wurde. Und das müsste bei jedem Aufruf der Methode `myMethod` erfolgen. Aber selbst auf diese Weise ist es nicht eindeutig vorherzusagen, welche Implementierung sich nun wirklich hinter ihm verbirgt:

```
int i = System.in.read();
MyInterface interf;
if(i > 0)
    interf = new Impl1();
else
    interf = new Impl2();
myMethod(interf);
```

Über `System.in.read()` wird eine Zahl eingelesen, die der Nutzer über die Kommandozeile eingibt. An Hand derer wird eine von zwei verschiedenen Implementierungen ausgewählt und der Methode `myMethod` übergeben. Um sicherzustellen, dass die Applikation lauffähig bleibt, müssen beide Varianten berücksichtigt werden. Es werden folglich auch beide Implementierungen benötigt.

Da das Zurückverfolgen des Kontrollflusses mit einem sehr hohen Aufwand verbunden ist, aber auch wegen solcher Fälle wie in obigem Beispiel, wird in dieser Arbeit ein approximierender Ansatz bei der Analyse auf Methodenebene gewählt.

Basis für diesen bildet folgende Beobachtung: Wird ein Interface benutzt, so muss zuvor eine Implementierung dieses Interfaces instanziiert worden sein, über die das Interface bedient wird.

Dies wird nun dahingehend ausgenutzt, dass in einer solchen Situation, nicht nur die Interface-Methode selbst als Abhängigkeit in die Resultatsmenge eingefügt wird, sondern gleichsam die entsprechenden Methoden von all denjenigen Implementierungen, die schon in der Abhängigkeitsmenge enthalten sind. Der letzte Punkt gewährleistet, dass auch wirklich nur die Implementierungen betrachtet werden, die in der Applikation benutzt werden.

Sicherlich ist es so möglich, dass Methoden aus diesen Klassen ins Resultat mitaufgenommen werden, die eigentlich gar nicht gebraucht werden. Dies betrifft dann insbesondere auch deren Abhängigkeiten. Dennoch ist garantiert, dass die so ermittelte Resultatsmenge eine Teilmenge von der ist, die bei einer Analyse auf Klassenebene bestimmt wurde. Dort werden aus diesen Interface-Implementierungen ohnehin die Abhängigkeiten aller Methoden übernommen.

Eine vergleichbare Problematik stellt hier das Vererbungskonzept von Java dar, genauer die Möglichkeit, Methoden der Superklasse zu überschreiben, wie folgendes Beispiel zeigt:

```java
class A {

    void methodA() {
        B.methodB();
    }
}


class A_EXT extends A {

    void methodA() {
        C.methodC();
    }
}


class D {

    void methodD(A a) {
        a.methodA();
    }
}
```

Die Klasse `A_EXT` erweitert `A` und überschreibt deren Methode `methodA`. `D`'s Methode erwartet als Argument ein Objekt des Typs `A`. Diese Bedingung erfüllt aber auch `A_EXT`, d.h. beim Ausführen von `a.methodA()` ist es sowohl möglich, dass die Methode von `A` aufgerufen wird, ebenso aber auch, dass es die von `A_EXT` ist. Im ersten Fall würde dann als weitere Abhängigkeit die Klasse `B`, im zweiten die Klasse `C` ermittelt werden.

Da jede Klasse eine Superklasse besitzt (außer `java.lang.Object`, von der alle Klassen erben), fällt die angesprochene Problematik hier noch viel stärker ins Gewicht als bei der Interfacebehandlung. Denn bei jedem Aufruf einer Klassenmethode ist zu überprüfen, ob hier auch eine Klasse verwendet worden sein kann, die diese Methode überschreibt.

Zur Lösung kann ganz genauso verfahren werden, wie im Falle der Interfaces. Es würde also in obigem Beispiel überprüft werden, ob `A_EXT` bereits in der Ergebnismenge enthalten ist. Ist dies der Fall, so wird auch dessen Methode `methodA` als Abhängigkeit übernommen.

4.2. Parsen von `class`-Dateien

Die Java-Klassen, die analysiert werden sollen, liegen als kompilierter Bytecode vor. Das Format dieser `class`-Dateien wird in [11] definiert. Darauf aufbauend wird nun die Implementierung eines Parsers vorgestellt.

Die Informationen, die dieser ermittelt, werden in einer speziell dafür entwickelten Datenstruktur gehalten, die folgende Eigenschaften der Klasse speichert:

- Name der Klasse
- Superklasse
- implementierte Interfaces
- ist Klasse selbst ein Interface?
- Felder der Klasse
- Methoden der Klasse
- in den Methoden dieser Klasse verwendete Klassen und Methoden

Eine `class`-Datei besteht generell aus einem Strom von 8-Bit Bytes. 16-, 32- oder 64-Bit-Quantitäten werden aus einer entsprechenden Anzahl von direkt aufeinanderfolgenden 8-Bit Bytes zusammengesetzt, basierend auf der Big-Endian-Codierung, wo hohe Bytes zuerst kommen.

Die Stream-Klasse `java.io.DataInputStream` unterstützt diese Codierung und ermöglicht durch Bereitstellung von Methoden wie `readUnsignedByte()`, `readUnsignedShort()` oder `readInt()` ein einfaches Lesen unterschiedlicher Bitgrößen. Daher wird sie auch in dieser Arbeit dazu verwendet. Der initiale Aufruf erfolgt folgendermaßen:

```
FileInputStream fin = new FileInputStream("NameOfClassFile");
DataInputStream din = new DataInputStream(fin);
```

Der generelle Aufbau einer `class`-Datei ist wie folgt:

```
ClassFile {
   u4 magic;
   u2 minor_version;
   u2 major_version;
   u2 constant_pool_count;
   cp_info constant_pool[constant_pool_count-1];
   u2 access_flags;
   u2 this_class;
   u2 super_class;
   u2 interfaces_count;
   u2 interfaces[interfaces_count];
   u2 fields_count;
   field_info fields[fields_count];
   u2 methods_count;
   method_info methods[methods_count];
   u2 attributes_count;
   attribute_info attributes[attributes_count];
 }
```

Die Typen `u1`, `u2` und `u4` geben dabei die Anzahl an Bytes an, durch die die nachfolgende Information gespeichert ist.

Für diese Arbeit ist es nicht notwendig alle Informationen, die in der `class`-Datei abgelegt sind, genau zu ermitteln, weshalb an dieser Stelle auch nur auf die hier wichtigen eingegangen wird.

4.2.1. Der Konstanten-Pool

Die erste wichtige Information, die extrahiert werden muss, ist der *Konstanten-Pool*. Dieser beinhaltet zahlreiche Klassennamen, `String`-Werte und andere Konstanten, auf die innerhalb der `class`-Datei verwiesen wird. Gespeichert ist dieser in Form einer Tabelle, wobei die Zahl der Einträge durch `constant_pool_count` angegeben wird.

Ein solcher Eintrag ist folgendermaßen aufgebaut:

```
cp_info {
   u1 tag;
   u1 info[];
}
```

Das `tag` gibt an, um welche Art es sich bei dem Eintrag handelt. Nachfolgend ist die Information über die Konstante gespeichert, deren Größe von der Art des Tags abhängt.

Die Konstantentypen, die für diese Arbeit von Bedeutung sind, sind:

- *CONSTANT_Class*:
  enthält einen Index in den Konstanten-Pool, wo der Name einer Klasse als UTF8-Wert gespeichert ist
- *CONSTANT_Fieldref, CONSTANT_Methodref, CONSTANT_InterfaceMethodref*:
  enthält einen Verweis auf eine Klasse, die im Konstanten-Pool gespeichert ist, sowie einen auf einen Eintrag des Typs `CONSTANT_NameAndType`, der das entsprechende Feld oder die Methode dieser Klasse näher beschreibt
- *CONSTANT_NameAndType*:
  hier ist ein Verweis auf den Namen eines Feldes oder einer Methode sowie auf deren Deskriptor gespeichert
- *CONSTANT_String*:
  enthält einen Index in den Konstanten-Pool, wo der Wert eines `String`s als UTF8-Wert gespeichert ist
- *CONSTANT_Utf8*:
  hier ist ein UTF8-codierter `String`-Wert gespeichert

Die gelesen Werte werden nun in einer Datenstruktur abgelegt, die gerade diesem Konstanten-Pool entspricht, jedoch eingeschränkt auf die hier benötigten Informationen.

Das nachfolgende Beispiel zeigt, wie im Pool die Referenz zu der Methode `addArchive` einer Klasse `datastructures.Archive` gespeichert wird:

```
index: 1 tag: CONSTANT_Methodref     name_and_typeIndex: 3  classIndex: 2
index: 2 tag: CONSTANT_Class         nameIndex: 4
index: 3 tag: CONSTANT_NameAndType   nameIndex: 5  descriptorIndex: 6
index: 4 tag: CONSTANT_Utf8          string_value: datastructures/Archive
index: 5 tag: CONSTANT_Utf8          string_value: addArchive
index: 6 tag: CONSTANT_Utf8          string_value: (Ljava/lang/String;)V
```

Die drei wichtigen Werte sind der Name der Klasse, der Name der Methode und der Deskriptor, der Argument- und Rückgabetypen der Methode angibt. Auf diesen wird später noch genauer eingegangen.

Zur Bestimmung einer im Pool gespeicherten Klasse oder Methode wurde eine Methode `getName` erstellt, die deren kompletten Namen aus den einzelnen Informationen zusammensetzt. Obiges Beispiel wird dann so ausgegeben:

```
datastructures/Archive.addArchive?(Ljava/lang/String;)V
```

Der Punkt separiert Klassen- und Methodenname, durch das '?' wird der Deskriptor abgetrennt. Dieser ist wichtig, da es unterschiedliche Methoden mit dem gleichen Namen geben kann. Somit kann die Methode jetzt eindeutig identifiziert werden.

4.2.2. Access-Flags

Nach dem Auslesen des Konstanten-Pools werden die *Access-Flags* der Klasse bestimmt. Diese definieren über eine Bit-Maske folgende Klasseneigenschaften:

Wert	Bedeutung
0x0001	Klasse ist als `public` deklariert
0x0010	Klasse ist als `final` deklariert
0x0020	Besondere Behandlung von Methoden der Superklasse
0x0200	Interface
0x0400	Klasse ist als `abstract` deklariert

Die hiervon benötigten Informationen können wie folgt gewonnen werden:

```
int accessFlags = din.readUnsignedShort();
boolean isInterface = ((accessFlags & 0x0200) != 0);
boolean isAbstract = ((accessFlags & 0x0400) != 0);
```

4.2.3. Superklasse und Interfaces

Im Anschluss werden durch Referenzen in den Konstanten-Pool der Name der zu Grunde liegenden Klasse und der Name ihrer Superklasse definiert, sowie die von dieser Klasse direkt implementierten Interfaces. Das sind genau diejenigen, die in der Quelldatei durch ein `implements`-Statement deklariert werden.

Hierbei ist noch eine kleine Besonderheit zu erwähnen: Repräsentiert die analysierte `class`-Datei ein Interface `I1` und erweitert dieses ein anderes Interface `I2`, so wird `I2` hier als ein implementiertes Interface und nicht als Superklasse gespeichert, denn diese ist dort stets `java.lang.Object`.

4.2.4. Felder

Die Felder der Klasse stellen die nächsten Einträge in der `class`-Datei dar und sind so aufgebaut:

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Die Access-Flags (`public`, `private`, `static`, ...) und auch der Name des Feldes sind für diese Arbeit uninteressant. Was benötigt wird, sind Informationen über den Typ des Feldes. Diesen liefert ein Deskriptor, der im Konstanten-Pool hinterlegt ist und über dessen Index referenziert wird. Die Grammatik für die Syntax eines *Feld-Deskriptors* ist folgendermaßen definiert:

FieldDescriptor := *Type*

Type := *BaseType* | *ObjectType* | *ArrayType*
BaseType := B | C | D | F | I | J | S | Z
ObjectType := L<classname>;
ArrayType := [*Type*

Es werden hierbei drei Arten von Typen unterschieden: Basisdatentypen (B $\approx$ `byte`, C $\approx$ `char`, D $\approx$ `double`, F $\approx$ `float`, I $\approx$ `int`, J $\approx$ `long`, S $\approx$ `short`, Z $\approx$ `boolean`), Objekttypen, die von einem 'L' und einem ';' eingeschlossen werden, und einem Typ für Arrays, der sich durch ein Präfix '[' kennzeichnet.

Ein zweidimensionales Feld vom Typ `java.lang.String` würde zum Beispiel so kodiert werden: `[[Ljava/lang/String;`. Man beachte auch, dass die Pakete durch '/' und nicht wie im Quellcode mit '.' abgetrennt werden.

Eine Besonderheit bilden hier noch parametrisierte Datentypen wie beispielsweise `java.util.List <java.lang.String>` (eine Liste, deren Elemente vom Typ `String` sind), die seit Version 1.5 in Java verfügbar sind ($\rightarrow$ Generizität). Diese werden im Deskriptor ohne Typparameter gespeichert, im Beispiel also durch `Ljava/util/List;`. Zu einem Feld-Eintrag können aber zusätzliche Attribute des Feldes definiert werden. Eines davon ist das Signatur-Attribut, welches zur Speicherung dieser Typparameter dient und auch das einzige ist, das für diese Arbeit eine Rolle spielt. Dieses Attribut verweist wieder auf einen Eintrag des Konstanten-Pools, in dem obiges Beispiel so gespeichert wäre: `Ljava/util/List<Ljava/lang/String;>;`.

4.2.5. Methoden

Nach der Deklaration der Felder werden die Methoden einer Klasse definiert. Zu jeder existiert ein `method_info`-Eintrag:

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Dieser ist analog zu den `field_info`-Elementen aufgebaut.

Es ist im Gegensatz zur Behandlung der Felder hier aber wichtig, zu bestimmen, ob die Methode als `private` deklariert wurde. Dies kann wie folgt ermittelt werden:

```
boolean is Private = ((accessFlags & 0x0002) != 0);
```

Der *Methoden-Deskriptor* ist jedoch etwas umfangreicher als der eines Feldes, da auch die verschiedene Argumenttypen sowie der Rückgabetyp der Methode definiert werden müssen. Die Grammatik für die Syntax eines Methoden-Deskriptors ist wie folgt:

MethodDescriptor := (*ParameterDescriptor**)*ReturnDescriptor*
ParameterDescriptor := *Type*
ReturnDescriptor := *Type* | V

Ein Methoden-Deskriptor besteht also aus einer beliebigen Anzahl von Parameter-Deskriptoren, gefolgt von einem Return-Deskriptor. Beide können aus denselben Typen bestehen, die schon beim Feld-

Deskriptor definiert wurden. Eine weitere Alternative für einen Rückgabewert ist `void`, was durch den Buchstaben 'V' gekennzeichnet wird.

Beispiel:

```
public int myMethod (String s, char c, List<String> l, Object[] o) {...}

(Ljava/lang/String;CLjava/util/List;[Ljava/lang/Object;)I
```

Auch hier wird die Liste im Deskriptor ohne den Typparameter `String` gespeichert. Wie oben auch erfolgt dies im Signatur-Attribut der Methode.

Generell spielen die Attribute einer Methode eine wichtige Rolle für die Analyse der Abhängigkeitsstruktur einer Klasse, weswegen hier auch genauer auf diese eingegangen wird.

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

Zunächst erfolgt ein Verweis auf die Art des Attributs, die wieder im Konstanten-Pool gespeichert ist. Dann folgt die Definition des eigentlichen Attributs, deren Byte-Größe durch `attribute_length` angegeben ist. Die für diese Arbeit wichtigen Attribute werden nun vorgestellt.

Im *Exceptions*-Attribut werden die Exceptions deklariert die einem `throws`-Statement im Methodenkopf folgen, wie zum Beispiel bei:

```
public void myMethod(int x) throws ArrayIndexOutOfBoundsException {...}
```

Am wichtigsten ist jedoch das *Code*-Attribut, weil in ihm der Rumpf der Methode gespeichert wird. Die Informationen, die hieraus später benötigt werden, umfassen die Typen der verwendeten Klassen, aber auch deren aufgerufene Methoden. Aufgebaut ist das Attribut wie folgt:

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {     u2 start_pc;
          u2 end_pc;
          u2 handler_pc;
          u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Der Methodenrumpf ist in der Tabelle `code` gespeichert. Seine Größe wird durch `code_length` bestimmt. Es folgt die Exceptionbehandlung, wo nur der Typ der gefangenen Ausnahmen `catch_type` interessiert, und die Definition weiterer Attribute.

Das Parsen dieses Codes ist etwas komplexer und wird daher im Folgenden auch ausführlicher dargestellt. Hierbei ist kein vollständiges Dekompilieren notwendig, sondern es wird sich darauf beschränkt, die verwendeten Klassen und Methoden zu ermitteln.

Es existieren 202 verschiedene Instruktionen der Java Virtual Machine [11], durch die der Code definiert wird. Die Instruktionen selbst werden durch ein Byte dargestellt, über das sie gemäß Anhang A identifiziert werden können.

Der Instruktion selbst können mehrere Argumente folgen. Die Anzahl dieser Operanden ist in den meisten Fällen fix. Es gibt jedoch drei Ausnahmen:

1. `WIDE`-Instruktion:
   Die Anzahl der Argumente bestimmter Instruktionen verdoppelt sich, wenn vor ihnen `WIDE` steht. Diese sind Anhang A zu entnehmen.
2. `LOOKUPSWITCH`-Instruktion:
   Diese Instruktion zur Darstellung eines `switch`-Statements hat eine variable Anzahl an Argumenten. Diese sind folgendermaßen aufgebaut:
   - 0-3 Null-Bytes als Padding
   - 4 Bytes für Default-Wert
   - 4 Bytes für die Anzahl `NUM` an Einträgen
   - `NUM` 4-Byte große Einträge der Switch-Tabelle

   Die 0-3 führenden Null-Bytes dienen dazu, dass das erste der folgenden 4 Bytes für den Default-Wert an einer Adresse liegt, die ein Vielfaches von 4 ist, gemessen von der Startadresse der ersten Instruktion der gerade geparsten Methode. Diese Anzahl lässt sich durch folgende Rechnung bestimmen, wenn die Zahl der bisher gelesenen Bytes in der Variablen `readBytes` gehalten wird:

   ```
   temp    = readBytes mod 4
   padding = (4 - temp) mod 4
   ```
3. `TABLESWITCH`-Instruktion:
   Auch hier gibt es eine variable Zahl an Operanden. Der Aufbau ist ähnlich wie oben:
   - 0-3 Null-Bytes als Padding
   - 4 Bytes für Default-Wert
   - 4 Bytes für untere Grenze `UG`
   - 4 Bytes für obere Grenze `OG`
   - `OG-UG+1` 4-Byte große Einträge der Switch-Tabelle

Die Argumente dieser Instruktionen sind nun gerade das, was für die Bestimmung der Abhängigkeiten einer Klasse, oder genauer einer Methode von ihr, von Bedeutung ist. Denn in einigen von ihnen stehen Verweise auf hier verwendete Klassen und Methoden. Welche dieser Instruktionen eine solche Referenz enthalten, findet sich in Anhang A. Sie haben alle mindestens 2 Operanden, von denen gerade die ersten beiden einen Index in den Konstantenpool darstellen, über den die jeweilige Klasse oder Methode referenziert wird. Alle anderen Operanden können überlesen werden.

Das folgende Beispiel zeigt einen Auszug der Methode des Parsers, die das Code-Attribut liest. Aus Gründen der Übersichtlichkeit werden hier nur einige Instruktionen betrachtet.

```
/* 'numOfBytes': Anzahl der zu lesenden Bytes
   'in': der zu parsende Stream */
void parseCode(int numOfBytes, DataInputStream in) {
  int readBytes = 0;      // Anzahl der gelesenen Bytes
  boolean wide = false;  // gibt an, ob zuvor "WIDE" gelesen wurde

  while(readBytes < numOfBytes){
    int instruction = in.readUnsignedByte(); // JVM-Instruktion
    readBytes++;

    switch (instruction){
      case WIDE:
        wide = true;
        break;
```

```
        /* überlesen */
        case ILOAD:
          /* wurde WIDE gesetzt, dann 2 statt 1 Argumente */
          if (wide){
            in.readUnsignedByte();
            readBytes++;
          }
          in.readUnsignedByte(); // 1 Byte überlesen
          readBytes++;
          wide = false;
          break;

        /* hier wird ein Methodenaufruf definiert */
        case INVOKEVIRTUAL:
          constantPoolIndex = in.readUnsignedShort();
          readBytes += 2;
          wide = false;
          break;

        [...]

      }

      String usedElement = getName(constantPoolIndex];
    }
  }
```

`INVOKEVIRTUAL` definiert beispielsweise den Aufruf einer Methode. Diese kann über die oben bereits erwähnte Funktion `getName` identifiziert werden.

Somit können nun der Rumpf einer Methode analysiert und die hier verwendeten Klassen und Methoden bestimmt werden.

Zwei Besonderheiten bei der Definition sind noch zu nennen. So werden Konstruktoren unter dem Namen `<init>` abgespeichert und es kann eine zusätzliche Methode `<clinit>` definiert sein. Diese dient im Gegensatz zu `<init>`, die ein Objekt der Klasse initialisiert, zur Initialisierung der Klasse selbst. Anwendung findet dies, wenn die Klasse statische Komponenten enthält oder ein Interface ist. Diese Methode wird dann implizit von der JVM aufgerufen, d.h. ein solcher Aufruf wird auch nie im oben beschriebenen Code-Attribut auftauchen.

4.2.6. Attribute

Am Ende der `class`-Datei werden noch einige Attribute der Klasse definiert. Dies erfolgt analog zu der von Feldern und Methoden. Hier können zum Beispiel Verweise auf innere Klassen deklariert werden. Dies ist aber, wie auch die anderen Attribute der Klasse, für diese Arbeit nicht von Bedeutung. Der Grund dafür ist, dass innere Klassen vom Compiler in separaten `class`-Dateien gespeichert werden, die von der JVM genauso wie ganz normale, äußere Klassen behandelt werden. Deshalb ist es für die Analyse der Abhängigkeitsstruktur auch unerheblich, wo diese deklariert wurden.

Es soll an dieser Stelle noch darauf hingewiesen werden, dass die Behandlung bestimmter Spezialfälle von TRAVIC Retail, wie die Verwendung von Reflection, schon hier beim Parsen beginnt. Dies wird jedoch erst in Abschnitt 4.4 vertieft, wo gerade diese Besonderheiten betrachtet werden.

Die Informationen, die für die Analyse der Abhängigkeiten zwischen Java-Klassen benötigt werden, sind nun durch Parsen der zu Grunde liegenden `class`-Dateien auf die hier beschriebene Weise identifizierbar.

4.3. Bestimmung der benötigten Klassen

In diesem Abschnitt wird die Implementierung eines Algorithmus vorgestellt, der zu einer gegebenen Menge von Klassen, all diejenigen bestimmt, die von diesen benötigt werden. Die zwei unterschiedlichen Alternativen bezüglich der Analysetiefe werden durch bestimmte Modi innerhalb des Algorithmus realisiert. D.h. im Wesentlichen ist dieser in beiden Fällen identisch und es wird nur an bestimmten Stellen eine Fallunterscheidung nach der Analysetiefe gemacht. Daher wird im Folgenden auch nur dort auf die verschiedenen Varianten eingegangen.

Vorweg soll allerdings noch auf eine Besonderheit bei der Analyse auf Methodenebene hingewiesen werden. So gilt es dort, zwei Arten von Abhängigkeiten zu unterscheiden:

- *Abhängigkeiten zu einer Klasse:*
  Gibt es eine Abhängigkeit zu einer Klasse, so werden neben der Klasse selbst benötigt:
  - Superklasse
  - implementierte Interfaces
  - verwendete Klassen und Methoden in der Methode `<clinit>`, die die Klasse initialisiert

  Dies Abhängigkeiten werden im Folgenden als *globale* Abhängigkeiten bezeichnet.
- *Abhängigkeiten zu einer Methode:*
  Gibt es eine Abhängigkeit zu einer Methode, so werden neben der zugehörigen Klasse benötigt:
  - globale Abhängigkeiten
  - in der Methode verwendete Klassen und Methoden

Wird also während der Analyse beispielsweise festgestellt, dass von Klasse `X` Methode `m` aufgerufen wird, so werden nicht nur die in `m` gefundenen Abhängigkeiten, sondern auch die Superklasse und die implementierten Interfaces von `X` benötigt. Von denen reichen dann wiederum die globalen Abhängigkeiten aus.

4.3.1. Datenstrukturen

Für die Abhängigkeitsanalyse wurden verschiedene Datenstrukturen entwickelt, die zur strukturierten Speicherung der ermittelten Informationen dienen. Diese sollen nun kurz vorgestellt werden, da sie im Folgenden häufiger verwendet werden.

Dabei wird sich auf die Felder der erstellten Klassen beschränkt, da die zugehörigen Methoden in der Regel nur dazu dienen, diese zu befüllen oder auszulesen. Die Klassen liegen in einem Paket `datastuctures`, dessen Präfix in den Klassennamen von nun an aus Übersichtlichkeitsgründen weggelassen wird.

Datenstrukturen zur Speicherung des Endresultats

Als Erstes werden nun die Datenstrukturen erläutert, in denen später das endgültige Analyseresultat enthalten sein wird. Diese geben Auskunft darüber, welche Klassen benötigt werden.

Die Klasse `Archive` repräsentiert ein Java-Archiv und dient zur Speicherung der in diesem enthaltenen Klassen. Ebenso umfasst sie eine Sammlung der Java-Archive, die im Klassenpfadeintrag des Manifests verzeichnet sind. Dieser muss am Ende der Analyse unter Umständen auch aktualisiert werden, falls

Referenzen zu einigen JARs nicht mehr gebraucht werden. Daher müssen in einer weiteren Menge die Archive vermerkt werden, aus denen gemäß der Analyse wirklich Klassen benötigt worden sind.

Die Klassen eines Archivs werden in `Archive` durch eine `HashMap` gespeichert, sodass ein schneller Zugriff auf die Informationen einer bestimmten Klasse möglich ist. Über einen Schlüssel (den Klassennamen) kann dann das entsprechende `JavaClass`-Objekt leicht abgerufen werden.

Die Datenstruktur `JavaClass` repräsentiert eine Klasse und umfasst diese Informationen:

- Name der Klasse
- Superklasse
- implementierte Interfaces
- ist Klasse ein Interface oder abstrakt?
- Speicherposition der Klasse
- das Archiv, in dem Klasse enthalten war
- ein Flag, das angibt, ob diese Klasse in die neu zu erstellende Anwendung zu übernehmen ist
- ein Flag, das angibt, ob alle Abhängigkeiten dieser Klasse bereits übernommen wurden
- ein Flag, das angibt, ob globale Abhängigkeiten dieser Klasse bereits übernommen wurden
- Auflistung der Methoden über `HashMap<String, Boolean>`,
  Schlüssel: Name der Methode
  Wert: `true`, falls Abhängigkeiten dieser Methode bereits übernommen wurden
- Auflistung aller privaten Methoden

Sie bildet den Kern der Analyse, da hier gespeichert ist, ob eine Klasse Teil der neuen Anwendung wird und welche ihrer Abhängigkeiten schon behandelt wurden.

Datenstrukturen zur Speicherung von Zwischenergebnissen

Zur Speicherung von temporären Zwischenergebnissen der Analyse werden drei Datenstrukturen verwendet.

`SimpleJavaClass` repräsentiert eine Sammlung von benötigten Komponenten einer Klasse, d.h. sie gibt an, ob von einer Klasse nur einzelne Methoden oder die komplette Klasse benötigt wird:

- Name der Klasse
- Flag, das angibt, ob die komplette Klasse benötigt wird
- Liste von Methoden aus dieser Klasse, die benötigt werden
- Flag, das angibt, ob globale Klassenabhängigkeiten benötigt werden

Mehrere solcher `SimpleJavaClasses` werden in einer `ClassCollection` zusammengefasst. Sie repräsentiert somit eine Menge von gefundenen Abhängigkeiten, die nach Klassen unterteilt sind. Ferner werden hier auch Informationen für die Behandlung von Spezialfällen gehalten, welche in Abschnitt 4.4 vorgestellt werden.

Anwendung findet die `ClassCollection` innerhalb der Datenstruktur `DependencyCollection`. Beim Parsen einer `class`-Datei werden die gefundenen Abhängigkeiten der Klasse hier gespeichert. Dabei wird für jede Methode eine `ClassCollection` erzeugt. Diese werden dann alle in der `DependencyCollection` gesammelt.

4.3.2. Der Algorithmus

Zunächst soll nun ein kurzer Überblick über die einzelnen Schritte des Algorithmus gegeben werden, ehe diese dann im Detail erläutert werden.

Eingabe:

- Menge M der Java-Klassen einer EAR-Datei
- Auswahl $I \subseteq M$
  (das Ergebnis aus Analyse der JavaEE-Komponenten, siehe 3)
- Klassenpfade der Java-Archive aus der EAR-Datei

Ziel:

- Menge $D \subseteq M$ von Java-Klassen, die von I benötigt werden

Algorithmus:

1. Vorbereitung der Analyse
   - Bestimmung des Suchraums A für abhängige Java-Klassen (Teilmenge der Java-Archive der EAR-Datei)
   - **for all** $a \in A$:
     - entpacke `class`-Dateien aus a
     - bestimme Basis-Informationen aller Java-Klassen aus a
2. Bestimmung der abhängigen Klassen
   - $D := getAllDependencies(I)$

   - **getAllDependencies(**C**)**
     $D_1 := \emptyset$
     for all $c \in C$
     $D_2 := getDirectDependencies(c)$
     $D_1 := D_1 \cup D_2$
     $D_1 := D_1 \cup getAllDependencies(D_2)$
     return D_1

   - **getDirectDependencies(**c**)**
     if $c.isAnalyzed()$ **then**
     return $\emptyset$
     else
     $D_c := Parser.getDependencies(c)$
     $c.isAnalyzed() :=$ **true**
     return D_c
3. Behandlung von Spezialfällen
   - Reflection
   - PPI-Spezialfälle

4.3.3. Eingabe

Als Eingabe für die Analyse dient neben der EAR-Datei selbst das Ergebnis aus Abschnitt 3, also das der Analyse der Abhängigkeiten zwischen JavaEE-Komponenten. Hierbei handelt sich um die Menge von Java-Klassen, die die Einstiegspunkte der benötigten JavaEE-Komponenten auf Klassenebene darstellen. Wichtig hierbei ist, dass diese Klassen nach den Archiven, in denen sie gespeichert sind, sortiert sind.

Ebenso sind die dort bestimmten Klassenpfade der einzelnen Java-Archive über die bereits erstellte Datenstruktur `datastuctures.Archive` abrufbar (vgl. 3.3).

4.3.4. Vorbereitung der Analyse

Bevor mit der Abhängigkeitsanalyse begonnen werden kann, müssen einige Vorbereitungen getroffen werden:

- Bestimmung des Suchraums A (Teilmenge der Java-Archive der EAR-Datei)
- **for all** $a \in A$:
  - entpacke `class`-Dateien aus a
  - bestimme Basis-Informationen aller Java-Klassen aus a

Bislang liegen alle Klassen innerhalb von Java-Archiven. Um diese aber parsen zu können, ist es erforderlich, sie daraus zu extrahieren. Nun ist es im Allgemeinen aber nicht erforderlich, alle Archive der EAR-Datei zu entpacken.

Die Klassen, die als Abhängigkeiten ermittelt wurden, müssen im aktuellen Klassenpfad liegen. Dieser wird durch das Manifest des Archivs definiert, in dem die analysierte Klasse enthalten ist. Demnach reicht es aus, nur die Archive zu betrachten, die von den Eingabe-Klassen gemäß der Klassenpfaddefinitionen erreichbar sind. Dies bietet sich vor allem deshalb an, weil das Entpacken sehr zeitintensiv ist.

Die Bestimmung dieser Archive lässt sich wie folgt durchführen. Die EAR-Datei wurde in einer Instanz der Datenstruktur `datastuctures.Archive` gespeichert (siehe Abschnitt 3.3). Sie umfasst bereits alle in ihr enthaltenen Archive, die über Instanzen der selben Datenstruktur gehalten werden, und stellt einen zentralen Punkt während der gesamten Analyse dar. Deswegen wird sie in einer globalen Variablen `mainArchive` gespeichert, auf die im weiteren Verlauf des Öfteren zurückgekommen wird. Über ein Flag `isEnabled` können nun die darin gehaltenen Archive für die Analyse zugelassen werden:

```
/* inputArchives: Archive, aus denen Eingabeklassen stammen */
for(String archiveName : inputArchives)
    enableArchive(archiveName);

void enableArchive(String archiveName){
    Archive archive = mainArchive.getArchive(archiveName);
    if(!archive.isEnabled()){
        archive.setEnabled(true);
        for(String referencedArchive : archive.getClasspathArchives())
            enableArchive(referencedArchive);
    }
  }
```

Ausgehend von den Archiven, aus denen die Eingabeklassen stammen, werden alle Archive aktiviert, die über den Klassenpfad direkt oder indirekt erreichbar sind.

Anschließend werden alle `class`-Dateien aus den eben bestimmten Archiven entpackt. Aber nicht nur diese, sondern alle enthaltenen Dateien, mit Ausnahme von Java-Archiven (diese wurden bereits extrahiert). Das ist notwendig, da später aus diesen ein neues Archiv zusammengesetzt werden soll.

Das Entpacken erfolgt im Gegensatz zu Abschnitt 3.3 nicht durch die Klasse `java.util.JarFile`, sondern mit Hilfe eines *Ant-Skripts* (vgl. 2.6). In 3.3 sollten mit den Deployment-Deskriptoren nur einige wenige, spezielle Dateien extrahiert werden. Nun sollen aber (fast) alle Dateien eines Archivs entpackt werden.

Dazu wird in der Datei `build.xml` ein ausführbares Target „unpack" definiert, das gerade alle Dateien (außer Java-Archive) in ein temporäres Verzeichnis entpackt:

```
<?xml version="1.0"?>
<project name="Archiv-Packer" basedir=".">
```

```
  <target name="unpack" description="Entpacken eines Archivs in ${dst.dir} (bis
      auf enthaltene Archive)">
    <mkdir dir="${dst.dir}"/>
    <unzip src="${originalArchive.file}" dest="${dst.dir}" >
      <patternset>
        <exclude name="*.jar"/>
        <exclude name="*.war"/>
        <exclude name="*.ear"/>
      </patternset>
    </unzip>
  </target>

</project>
```

Für jedes zu entpackende Archiv wird dieses Target einmal aufgerufen. Dort wird zunächst über den Befehl `<mkdir>` ein temporäres Verzeichnis erstellt, dessen Name dem des Archivs entspricht. Da dieser bei jedem Aufruf des Skripts unterschiedlich ist, wird er aus einer Property `${dst.dir}` gelesen. Diese entspricht einer Variablen und wird durch ein „$“ gekennzeichnet. Ihr Name wird innerhalb der anschließenden Klammern „{“ und „}“ definiert. Wie diese Property gesetzt werden kann, wird etwas später erläutert werden.

Die vordefinierte Task `<unzip>` entpackt nun das im Attribut `src` angegebene Archiv in das Verzeichnis `dest`. Über ein `patternset` können Muster für Dateien angegebenen werden, die nicht mit extrahiert werden sollen. Im Beispiel sind dies gerade die Java-Archive.

Die Ausführung des Ant-Skripts selbst kann direkt von Java aus erfolgen. Durch Verwendung der Bibliotheken *ant.jar* und *ant-launcher.jar* der Apache Ant API [12] können die oben beschriebenen Properties nun gesetzt, die Datei `build.xml` eingelesen und das entsprechende Projekt erzeugt und ausgeführt werden.

Dazu wird eine Klasse `AntRunner` erstellt, deren Methode `runAnt` gerade die hier beschriebenen Schritte durchführt:

```
static void runAnt(String buildFile, String target, HashMap<String, String>
    properties) throws BuildException {

    /* Erzeuge ein neues Projekt */
    Project project = new Project();
    project.init();

    /* Properties setzen */
    for(Entry<String, String> entry : properties.entrySet())
        project.setProperty(entry.getKey(), entry.getValue());

    /* Einlesen des Build-Files */
    ProjectHelper.getProjectHelper().parse(project, new File(buildFile));

    /* Ausführen des Skripts */
    project.executeTarget(target);
}
```

Ein Aufruf der Methode `runAnt` aus dem Programm heraus erfolgt dann so:

```
void unpack(Archive archive) {
    HashMap<String, String> properties = new HashMap<String, String>();
    properties.put("src.dir", tempDir + archive.getArchiveName());
    properties.put("originalArchive.file", archive.getArchiveSourceName());
    AntRunner.run(antDir + "/build.xml", "unpack", properties);
}
```

Damit wurden alle benötigten Dateien der Anwendung in ein temporäres Verzeichnis entpackt. Dieses enthält für jedes ihrer Archive ein eigenes Unterverzeichnis, in dem die Klassen gemäß der Verzeichnisstruktur innerhalb des zu Grunde liegenden Archivs gespeichert sind. Die genaue Speicherposition der einzelnen class-Dateien wird später noch wichtig sein, wenn bestimmte Klassen innerhalb dieses Verzeichnisses gesucht und wieder in ein neues Archiv gepackt werden müssen. Zum Beispiel wird die Klasse server.config.Routing aus dem Archiv Server.jar dann an der Position <TempDir>/Server.jar/server/config/Routing.class gespeichert.

Sind alle class-Dateien entpackt, so werden einige wichtige Basisinformationen zu jeder Klasse bestimmt, die vor allem bei der späteren Behandlung von Reflection benötigt werden (vgl. Abschnitt 4.4.1). Diese sind:

- Name der Klasse
- Superklasse
- implementierte Interfaces
- ist Klasse ein Interface oder abstrakt?
- Speicherposition der Klasse
- das Archiv, in dem die Klasse enthalten war

Sie werden weitestgehend über den bereits vorgestellten Parser ermittelt, wobei dieser nicht die komplette class-Datei, sondern nur ihren Anfang lesen muss, d.h. bis zu der Stelle, an der die implementierten Interfaces gespeichert sind. Vermerkt werden die Informationen in der Datenstruktur JavaClass.

Es stellt sich vielleicht die Frage, warum an dieser Stelle nicht gleich alle Informationen aus der class-Datei bestimmt werden, also insbesondere auch die Methoden und die darin festgestellten Klassenabhängigkeiten.

Der Grund dafür liegt im Umfang der analysierten JavaEE-Anwendungen, die aus bis zu 16.000 Klassen bestehen können.

4.3.5. Einschränkungen bei der Analyse

Aus der Größe der zu betrachtenden Software entstehen einige Einschränkungen für die Implementierung des Analysewerkzeugs.

Ideal wäre es, wenn zu jeder Methode einer Klasse alle in dieser verwendeten Klassen und Methoden dauerhaft gespeichert werden könnten.

Dies war auch der erste Ansatz in dieser Diplomarbeit, allerdings war das Resultat ein bereits frühzeitiger Speicherüberlauf. Prinzipiell ist es möglich, den Standardwert für den Speicher der JVM bei deren Start zu erhöhen (der Initialwert ist 64 MB). Doch ist das keine befriedigende Lösung, da dies auf dem System, auf dem das erstellte Analyseprogramm später eingesetzt wird, eventuell nicht möglich oder nicht gewünscht sein kann. Auch die wirklich benötigte Speichergröße kann von Fall zu Fall deutlich variieren, da dies vom Umfang der analysierten Applikation abhängt.

Die Alternative besteht nun darin, die benötigten Informationen über direkte Klassenabhängigkeiten nur temporär zu speichern. Diese werden ausgewertet und anschließend wieder verworfen. Letzteres hat einen entscheidenden Nachteil: Werden diese Informationen im weiteren Verlauf erneut benötigt, so müssen sie ein weiteres Mal bestimmt werden, d.h. die class-Datei muss mehrfach geparst werden. Dies tritt jedoch nur bei der Analyse auf Methodenebene auf, wie man später noch sehen wird.

Konsequenz dieses Ansatzes ist natürlich eine erhöhte Laufzeit bei der Analyse.

Wie zum Ende des vorherigen Abschnitts erläutert wurde, werden zu Beginn der Analyse Basisinformationen einer jeden Klasse bestimmt und dauerhaft gespeichert. Dies ist sicherlich weder in Hinsicht auf eine sparsame Speichernutzung noch in Bezug auf eine schnellere Laufzeit von Vorteil. Dennoch ist es notwendig.

Der Grund dafür ist die Verwendung von Reflection in TRAVIC Retail (vgl. Abschnitt 4.4.1). Bei deren Behandlung wird es erforderlich sein, zu jedem Interface bzw. zu jeder Klasse der Anwendung zu wissen, welche Klassen diese implementieren bzw. erweitern. Das heißt, es müssen wirklich alle Klassen einmal geparst werden.

Die Auswirkungen auf Speicherverbrauch und Laufzeit sind trotzdem überschaubar. Da nur einige Basisinformationen aus der `class`-Datei gelesen werden müssen, braucht auch nur ein relativ kleiner Anfangsteil davon geparst zu werden. Dies ist nicht sehr zeitintensiv und fällt in dieser Phase der Analyse auch nicht sonderlich ins Gewicht, da die dominierende Größe hier das Entpacken der Archive darstellt.

4.3.6. Bestimmung der abhängigen Klassen

Nun kann mit der eigentlichen Analyse der Abhängigkeiten begonnen werden. Für jede Klasse wurde ein Objekt des Typs `JavaClass` erzeugt und bereits mit den vorgestellten Basisinformationen befüllt. Sie sind wiederum in `Archive`-Objekten gespeichert.

Die Eingabe-Klassen liegen in Form von `ClassCollections` vor. Für jedes der Archive, aus denen diese Klassen stammen, wurde eine solche Collection erzeugt. Sie enthalten nun gerade die Klassen, die als Einstiegspunkte der zu Grunde liegenden JavaEE-Module ermittelt wurden. Von diesen Klassen werden alle Abhängigkeiten benötigt, d.h. in der jeweiligen Instanz von `SimpleJavaClass` ist das Flag `complete` auf `true` gesetzt.

Die Analyse wird nun durch Aufruf der initialen Methode `analyzeClasses` angestoßen:

```
void analyzeClasses(HashMap<String, ClassCollection> inputClasses) {
    /* hier werden Archive aus dem Class-Path gespeichert (Suchraum) */
    HashSet<String> archivesToSearch = new HashSet<String>();

    /* Eingabeklassen analysieren und rekursive Abhängigkeiten bestimmen */
    for(String archiveName : inputClasses.keySet()){
        archivesToSearch.add(archiveName);
        findDependencies(archiveName, inputClasses.get(archiveName),
            archivesToSearch);
        archivesToSearch.clear();
    }

    /* Interface-Implementierungen und Klassenerweiterungen behandeln */
    if(analyzeAtMethodLevel)
        handleInterfaces();
}
```

Wie man erkennt, wird für jede `ClassCollection` einmal die Methode `findDependencies` aufgerufen. Diese bestimmt alle Abhängigkeiten der Klassen aus der Collection, d.h. sowohl direkte als auch indirekte.

Anschließend erfolgt die Behandlung der Problematik von Interfaces und Klassenerweiterungen, die in Abschnitt 4.1 vorgestellt wurde. Hierauf wird aber erst später eingegangen.

Auffinden der benötigten Klassen

Wurde eine Menge von benötigten Klassen ermittelt, so müssen diese zunächst in der EAR-Datei gesucht werden. Es muss also bestimmt werden, in welchem Archiv sie enthalten sind. Anschließend können ihre Abhängigkeiten untersucht werden. Diese Suche erfolgt immer über die bereits erzeugten `Archive`-Instanzen, die bereits alle notwendigen Informationen umfassen.

Die Lösung dieser zwei Aufgaben startet bei der Methode `findDependencies`. Sie sucht nach den Klassen, die im Argument `classesToFind` gespeichert sind, und bestimmt deren Abhängigkeiten. `archivesToSearch` definiert den Suchraum, also diejenigen Archive, in denen diese Klassen liegen können. Der dritte Parameter `archiveName` liefert den Namen des Archivs, dessen benötigte Klassen (`classesToFind`) gesucht werden sollen.

```
void findDependencies(String archiveName, ClassCollection classesToFind,
    Collection<String> archivesToSearch) {

    /* hier werden neue Klassenabhängigkeiten gespeichert */
    ClassCollection dependendClasses = new ClassCollection();

    /* suche nach benötigten Klassen und bestimme neue Abhängigkeiten */
    boolean classFound;
    for(String archiveToSearch : archivesToSearch){

        /* Archiv durchsuchen und direkte Abhängigkeiten bestimmen */
        classFound = searchAndAnalyzeArchive(archiveToSearch, classesToFind,
            dependendClasses);

        /* wurde Klasse gefunden? */
        if(classFound){
            /* bestimme zunächst die Abhängigkeiten der neuen Abhängigkeiten */
            Archive archive = mainArchive.getArchive(archiveToSearch);
            findDependencies(archiveToSearch, dependendClasses,
                archive.getClasspathArchives());

            mainArchive.getArchive(archiveName).addNeededArchive(archiveToSearch);
        }
    }

    /* speichern, welche Klassen nicht gefunden werden konnten */
    notFoundClasses.addClassCollection(classesToFind);
    classesToFind.clear();
}
```

Die Bestimmung der Abhängigkeiten der Klassen erfolgt immer archivweise, d.h. es werden die Abhängigkeiten von allen benötigten Klassen eines Archivs gebündelt und erst dann erfolgt eine rekursive Analyse von deren Abhängigkeiten. Diese Vorgehensweise hat zwei Gründe: Zum Einen wird der Suchraum, in dem die benötigten Klassen liegen können, durch den Klassenpfad des Archivs definiert. Dieser ist natürlich für alle Klassen des Archivs gleich, aber auch nur für diese. Zum Anderen ist es deutlich effizienter, nicht für jede Klasse einzeln nach Abhängigkeiten suchen zu müssen. Denn so kann eine mehrfache Suche nach denselben Klassen verhindert oder zumindest reduziert werden.

In obiger Methode werden nun die Archive aus `archivesToSearch` nacheinander nach den benötigten Klassen abgesucht. Dies erfolgt über die Prozedur `searchAndAnalyzeArchive`, die `true` zurückgibt, falls sie eine Klasse in dem Archiv gefunden hat, das ihr als Argument übergeben wurde. Ist dies der Fall, so werden in der Variablen `dependendClasses` alle gefundenen, direkten Abhän-

gigkeiten der gesuchten Klassen (bzw. von deren benötigten Methoden) gespeichert. Konnte eine Klasse aus `classesToFind` gefunden werden, so wird sie aus dieser Collection herausgenommen.

Durch einen rekursiven Aufruf der Methode `findDependencies` werden jetzt auch alle indirekten Abhängigkeiten der in diesem Schleifendurchlauf gefundenen Klassen aus `classesToFind` bestimmt. Dazu wird als Argument einfach die Menge der direkten Abhängigkeiten übergeben.

Schließlich müssen noch Vorbereitungen für die Aktualisierung des Klassenpfads desjenigen Archivs getroffen werden, von dem hier benötigte Klassen gesucht und analysiert wurden. Dieses wird über den `String archiveName` referenziert. Die Elemente aus `archivesToSearch` repräsentieren gerade den alten Klassenpfad. Diejenigen dieser Archive, aus denen keine Klassen mehr benötigt werden, sollen aus dem Class-Path-Eintrag des Manifests entfernt werden. Daher werden an dieser Stelle die Archive vermerkt, in denen eine gesuchte Klasse gefunden wurde. Das realisiert `addNeededArchive`.

Ist die `for`-Schleife abgearbeitet und die Collection `classesToFind` nicht leer, so sind in ihr Klassen enthalten, die nicht gefunden werden konnten. Hierfür kann es zwei Gründe geben: Entweder ist der Class-Path-Eintrag der Manifestdatei unvollständig oder die gesuchte Klasse ist gar nicht in der EAR-Datei enthalten. Dies kann zum Beispiel daran liegen, dass die Klasse von der Laufzeitumgebung zur Verfügung gestellt wird. So würde `java.lang.Object` hier in der Regel nicht gefunden werden.

Bestimmung der direkten Abhängigkeiten

In der Methode `searchAndAnalyzeArchive` wird nun in einem Archiv nach bestimmten Klassen gesucht. Konnte eine gefunden werden, so wird das entsprechende `JavaClass`-Objekt zurückgegeben. Mit diesem als Argument wird eine weitere Methode `analyzeClass` aufgerufen, die die direkten Abhängigkeiten der gefundenen Klasse ermittelt. Das ist jedoch nur dann nötig, wenn nicht früher schon alle Abhängigkeiten der Klasse bestimmt worden sind. Dies kann leicht durch das entsprechende Flag der Datenstruktur `JavaClass` abgefragt werden.

Es folgt der Quellcode dieser Methode:

```
void analyzeClass(SimpleJavaClass classToFind, JavaClass javaClass,
    ClassCollection dependendClasses) {

    /* Analyse auf Methodenebene */
    if(analyzeAtMethodLevel){

        /* überprüfen, welche Teile der Klasse schon behandelt wurden */
        if(javaClass.isMethodsParsed())
            javaClass.checkForNeededClassParts(classToFind, dependendClasses);

        /* fehlt noch etwas? */
        if(!classToFind.isEmpty()){
            /* Parsen */
            DependencyCollection dependencyCollection
                = parser.parseClassFile(javaClass.getSourceFileName());

            /* Abhängigkeiten der benötigten Teile übernehmen */
            dependencyCollection.getDependencies(classToFind, javaClass,
                dependendClasses);
        }
    }

    /* Analyse auf Klassenebene */
    else{
        parser.parseClassFile(javaClass.getSourceFileName(), dependendClasses);
        javaClass.setComplete();
```

```
    }
}
```

Sie enthält drei Argumente:

- `classToFind`: enthält Informationen darüber, welche Teile der Klasse gebraucht werden
- `javaClass`: enthält Informationen darüber, inwieweit die Klasse schon analysiert wurde, d.h. von welchen Methoden die Abhängigkeiten schon bestimmt wurden
- `dependendClasses`: hier werden die gefundenen Abhängigkeiten gespeichert

An dieser Stelle ist jetzt eine Unterscheidung der Analyseebenen nötig.

Der einfachere Fall ist eine Analyse auf Klassenebene. Hier wird der `class`-Datei-Parser aufgerufen, die komplette Klasse geparst und *alle* gefundenen Abhängigkeiten direkt in `dependendClasses` eingefügt.

Bei einer Analyse auf Methodenebene ist es etwas komplizierter. Als Erstes wird überprüft, ob schon Abhängigkeiten bestimmter Teile (also vor allem Methoden) dieser Klasse analysiert wurden. Ist dies der Fall, so brauchen diese Abhängigkeiten nicht noch ein zweites Mal übernommen werden. Es werden also in `classToFind` die Referenzen zu eben diesen Methoden entfernt. Dies wird über die Prozedur `checkForNeededClassParts` der Datenstruktur `JavaClass` realisiert. Des Weiteren ist damit auch klar, dass die globalen Klassenabhängigkeiten bereits behandelt wurden und nicht noch einmal übernommen werden müssen.

Nachdem jetzt bekannt ist, von welchen Klassenbestandteilen Abhängigkeiten ermittelt werden müssen, wird die `class`-Datei geparst und das Ergebnis in einer `DependencyCollection` abgespeichert. Dabei werden die Abhängigkeiten gemäß der Methoden der Klasse unterteilt, indem für jede Methode eine Instanz einer `ClassCollection` erzeugt wird. Eine weitere wird zur Speicherung der globalen Abhängigkeiten verwendet.

`ClassCollections` bestehen aus `SimpleJavaClasses`. Wird beim Parsen festgestellt, dass eine Methode `m1` einer Klasse `C` gebraucht wird, so wird ein neues Objekt vom Typ `SimpleJavaClass` zur Repräsentation aller Abhängigkeiten zu `C` erstellt. In dieses wird dann auch die gefundene Methode `m1` eingefügt (bzw. deren Name). Wird herausgefunden, dass auch Methode `m2` von `C` benötigt wird, so wird auch sie in diesem `SimpleJavaClass`-Objekt gespeichert.

Es muss aber nicht immer so sein, dass ein solches Objekt Methodenreferenzen enthält. Dies ist genau dann nicht der Fall, wenn nur globale Abhängigkeiten benötigt werden. Ein Beispiel dafür wäre der direkte Zugriff auf ein Feld der entsprechenden Klasse.

Der Parser gibt also eine `DependencyCollection` zurück, die aus solchen `SimpleJavaClass`-Objekten besteht (strukturiert durch `ClassCollections`).

Nun gilt es, aus dieser Menge genau diejenigen Abhängigkeiten herauszufiltern, die zu den wirklich benötigten Methoden gehören. Das wird in der Klasse `DependencyCollection` durch die Methode `getDependencies` realisiert, die jetzt näher vorgestellt werden soll.

Die beim Parsen ermittelten Abhängigkeiten werden in den Variablen `globalDependencies` und `methodDependencies` gehalten, auf die von dieser Methode aus zugegriffen wird. Letztere ist eine `HashMap`, deren Werte die angesprochenen `ClassCollections` sind.

```
void getDependencies(SimpleJavaClass neededDependencies, ClassCollection
    newDependencies) {

    /* Werden globale Abhängigkeiten auch noch benötigt? */
    if(neededDependencies.needsGlobalDependencies()){
        /* alle globalen Abhängigkeiten übernehmen */
```

```
        for(SimpleJavaClass c : globalDependencies)
            newDependencies.addClass(c);
    }

    /* benötigte Methoden durchlaufen, um neue Abhängigkeiten zu finden */
    Collection<String> methodsOfSuperclass = new LinkedList<String>();
    for(String methodName : neededDependencies.getMethods()){

        /* Methode nicht hier? dann in Superklasse suchen */
        if(!methodDependencies.containsKey(methodName))
            methodsOfSuperclass.add(superClassName + "." + methodName);

        else{
            ClassCollection method = methodDependencies.get(methodName);
            newDependencies.addClassCollection(method);
        }
    }

    newDependencies.add(methodsOfSuperclass);
}
```

Die benötigten Abhängigkeiten werden in `newDependencies` gesammelt, indem der Inhalt der jeweiligen `ClassCollections` eingemischt wird. Welche davon nun benötigt werden, ist an Hand des Parameters `neededDependencies` feststellbar.

Nun kann aber auch der Fall eintreten, dass eine gesuchte Methode in der behandelten Klasse nicht enthalten ist. Ein möglicher Grund dafür ist, dass sie gar nicht existiert. Es kann sich dabei aber auch um eine Methode der Superklasse handeln. Denn diese können auf die selbe Art und Weise aufgerufen werden wie die Methoden der Klasse selbst, ohne dass hier ein Unterschied erkennbar ist. In so einem Fall wird als weitere Abhängigkeit die entsprechende Methode der Superklasse eingefügt.

Damit sind die Abhängigkeiten der benötigten Bestandteile dieser Klasse bestimmt und in der `ClassCollection newDependencies` abgespeichert. Zum Schluss müssen diese Methoden noch gekennzeichnet werden, damit sie später nicht noch einmal analysiert werden.

Die vom Parser erzeugte `DependencyCollection` wird nun verworfen. Werden zu einem späteren Zeitpunkt weitere Methoden der entsprechenden Klasse benötigt, so muss die `class`-Datei erneut geparst werden und eine neue `DependencyCollection` erzeugt werden. Dies ist sicherlich unschön, da hier dieselbe Information mehrfach bestimmt wird, doch ist es leider notwendig.

Der Versuch, diese Collection nach dem ersten Parsen dauerhaft zur Wiederverwendung zu speichern, scheiterte an der Größe der analysierten Anwendungen. Es kam zu einem Speicherüberlauf. Dasselbe Problem tritt auch dann auf, wenn man nur die Teile der Collection aufbewahrt, die zu Methoden gehören, deren Abhängigkeiten noch nicht übernommen wurden, und die anderen verwirft. Somit ist der einzig verbleibende Ausweg der, diese Abhängigkeiten mehrfach zu bestimmen.

Hierbei kommt es auch zu Gute, dass die Abhängigkeiten zunächst archivweise gebündelt und erst dann weiter analysiert werden. Dadurch kann die Häufigkeit des Mehrfachparsens deutlich reduziert werden.

Auf die hier vorgestellte Art und Weise wurden nun alle benötigten Klassen ermittelt, zumindest bei der Analyse auf Klassenebene. Auf Methodenebene fehlt noch die Behandlung von Interfaces und Klassenerweiterungen.

4.3.7. Behandlung von Interfaces und Klassenerweiterungen

In Abschnitt 4.1 wurde die Problematik vorgestellt, die bei der Abhängigkeitsanalyse auf Methodenebene in Bezug auf Interfaces und Klassenerweiterungen auftritt. Es gilt nun herauszufinden, welche Implementierungen hinter den verwendeten Interfaces stecken können und welche Klassen Methoden ihrer Superklasse überschreiben.

Die Behandlung dessen erfolgt am Schluss der Analyse, auch wenn die genauere Vorgehensweise eigentlich die folgende wäre:

- es wird beim Parsen von Methode `m1` aus Klasse `A` festgestellt, dass Methode `m2` von Interface `I` aufgerufen wird
- bestimme Abhängigkeiten von `A`
- übernehme ebenso die Abhängigkeiten der Methode `m2` einer jeden Implementierung von `I`, die im bisherigen Analyseresultat enthalten ist

Das wesentliche Kriterium für die Wahl des spätest möglichen Zeitpunkts ist die Tatsache, dass bei jedem Aufruf von `m1` eine andere Implementierung von `I` genommen worden sein kann. Als Folge davon müsste auch bei jedem dieser Aufrufe überprüft werden, welche Implementierung es in diesem Fall gewesen sein kann.

Dies gilt aber nicht nur für Interfacemethoden, sondern auch für Methoden, welche die ihrer Superklasse überschreiben. Im schlimmsten Fall muss so eine Überprüfung also bei jedem Methodenaufruf erfolgen, was sicherlich nicht sehr effizient wäre.

Daher entschloss man sich, die Behandlung gebündelt am Schluss durchzuführen, auch wenn die Konsequenz davon ist, dass die Menge der dort bestimmten Klassen größer sein kann als bei der alternativen Vorgehensweise.

Das Ziel ist es nun, von allen Methoden aus der ermittelten Abhängigkeitsmenge zu bestimmen, ob an Stelle derer auch eine andere verwendet worden sein kann. Dies gilt insbesondere auch für Methoden von Klassen, die in der EAR-Datei gar nicht gefunden werden konnten:

```
class MyListImpl implements java.util.List {...}

List list = new MyListImpl();
list.add(elem);
```

Hier wurde eine Klasse definiert, die das Interface `java.util.List` implementiert. Dieses wird in der Regel nicht in der EAR-Datei selbst enthalten sein, da es bereits von der Laufzeitumgebung zur Verfügung gestellt wird.

Wird bei der Abhängigkeitsanalyse nun die Methode `java.util.List.add` als benötigtes Element ermittelt, so wird die Klasse `java.util.List` nicht gefunden werden. Es werden aber die Abhängigkeiten von `MyListImpl.add` benötigt. Also müssen bei der Behandlung von Interfaces und Klassenerweiterungen auch nicht gefundene Klassen berücksichtigt werden.

Es werden jetzt, wieder archivweise, alle Klassen untersucht, die in der bisher ermittelten Abhängigkeitsmenge enthalten sind. Warum diese ausreichen, wurde bereits in Abschnitt 4.1 begründet.

Der Kern der Methode, die diese Analyse für alle Klassen eines Archivs durchführt, ist in diesem Code-Auszug dargestellt:

```
for(JavaClass javaClass : archive.getJavaClasses()){
    /* Gibt es Methoden, deren Abhängigkeiten noch nicht übernommen wurden? */
    if(javaClass.isTaken() && !javaClass.isComplete()){
```

```
            /* Bestimme noch nicht übernommene Methoden */
            for(String methodName : javaClass.getMethods()){
                if(!javaClass.isMethodTaken(methodName)
                      && !methodName.startsWith("<init>")
                      && !javaClass.isPrivate(methodName)){
                    methods.add(methodName);
                    methodsToTake.add(methodName);
                }
            }

            /* a) Überprüfe implementierte Interfaces */
            for(String implementedInterface : javaClass.getImplementedInterfaces())
                checkClass(implementedInterface, methods);

            /* b) Überprüfe, ob Methoden der Superklasse überschrieben werden */
            if(!methods.isEmpty())
                checkClass(javaClass.getSuperClassName(), methods);

            /* Bestimme noch benötigte Abhängigkeiten */
            if(!methodsToTake.isEmpty()){
                [Bestimme direkte Abhängigkeiten der Methoden]
            }
        }
    }
```

Es wird über alle Klassen des Archivs iteriert, behandelt werden aber nur diejenigen, die schon ins Endresultat der Analyse übernommen wurden. Wurden bereits die Abhängigkeiten aller Methoden übernommen, so kann die Klasse übersprungen werden.

Zunächst werden dann gerade die Methoden bestimmt, die bislang nicht benötigt wurden. Sie werden in der Variablen `methods` gespeichert. Ausgeschlossen davon sind lediglich Konstruktoren, da diese nur direkt über die Klasse aufgerufen werden können, und private Methoden, die weder eine Interfacemethode implementieren noch etwas überschreiben können. Sie spielen hier deshalb keine Rolle.

Anschließend wird für jedes von dieser Klasse implementierte Interface überprüft, ob eine der Methoden aus `methods` in diesem verwendet wurde. Ist dies der Fall, so wird die Methode aus `methods` entfernt, da sie nun nicht mehr gesucht werden muss. Diese Überprüfung wird von der Methode `checkClass` erledigt. Wichtig ist dabei, nicht nur die direkt implementierten Interfaces zu betrachten, sondern auch diejenigen, die von diesen erweitert werden.

Das gleiche gilt für die Superklassen. Hier müssen auch alle Superklassen der Superklasse betrachtet werden. Bei der Behandlung von Klassenerweiterungen ist die Analyse jedoch noch etwas komplexer, da dort nur Abhängigkeiten von Methoden übernommen werden müssen, die von der gerade behandelten Klasse überschrieben werden. Es muss also in einem ersten Schritt überprüft werden, ob gerade dies der Fall ist.

Generell wurden Methoden stets so abgespeichert, dass sich ihr Bezeichner aus dem Methodennamen und dem Methodendeskriptor zusammensetzt. An Hand dieser Informationen können nun zwei Methoden `a` und `b` dahingehend verglichen werden, ob `a` `b` überschreibt.

In Java überschreibt eine Methode `a` eine Methode `b` genau dann, wenn gilt:

- Methodennamen sind gleich
- Argumenttypen sind gleich
- der Rückgabetyp R_A der überschreibenden Methode `a` ist ein Untertyp des Rückgabetyps R_B der überschriebenen Methode `b`, d.h. $R_A = R_B$ oder R_A `extends` R_B

- die geworfenen Exceptions müssen folgendermaßen zueinander passen: wirft Methode `a` eine Exception E_A, so wirft `b` eine Exception E_B, die ein Untertyp von E_A ist. Ausnahme: E_A erweitert `java.lang.RuntimeException`
- Access-Flags der Methoden müssen konform sein (`public`, `private`, `static`,...)

Bei der Konformität der Access-Flags muss nur sichergestellt sein, dass `b` nicht `private` ist. Alle anderen Kombinationen, die vom Compiler erlaubt werden, ermöglichen eine Überschreibung.

Dies gilt auch für die Exceptions, d.h. alle Kombinationen, die nicht der oben genannten Charakterisierung genügen, werden vom Compiler nicht akzeptiert.

Es folgt ein Beispiel, wobei hier auf die Darstellung der Klassen, in denen diese Methoden definiert werden, verzichtet wurde:

```
(1)  public Object myMethod(List s) throws IOException {...}

(2)  public String myMethod(List s) throws ZipException, RuntimeException {...}

(3)  public String myMethod(Collection s) throws ZipException {...}
```

(2) überschreibt (1), da `String` die Klasse `Object` erweitert und `IOExcpetion` eine Superklasse von `ZipException` ist. Die `RuntimeException` ist hier auch erlaubt, obwohl sie keine Erweiterung von `IOExcpetion` ist. (3) hingegen überschreibt (1) nicht, da die Argumenttypen unterschiedlich sind: das Interface `List` erweitert `Collection`.

Nun bedarf es noch einer genaueren Betrachtung von parametrisierten Datentypen.

Bei Argumenttypen müssen die Typparameter entweder genau übereinstimmen oder die der überschreibenden Methoden besitzt gar keine. Andernfalls erfolgt schon beim Kompilieren der erweiternden Klasse eine Fehlermeldung.

Bei Rückgabetypen verhält es sich ähnlich. Dort müssen die Typparameter gleich sein oder eine der Methoden hat keine, wobei es hier egal ist, welche der beiden dies ist. Ansonsten schlägt auch hier das Kompilieren fehl.

Beispiel:

```
(1)  public Object myMethod(List<String> s) {...}

(2)  public Object myMethod(List s) {...}

(3)  public Object myMethod(List<Object> s) {...}
```

Hier überschreibt (2) wieder (1), da in (2) die Liste keine Typparameter hat. (1) hingegen könnte (2) nicht überschreiben. Auch kann (3) Methode (1) nicht überschreiben, da die Parametertypen unterschiedlich sind.

Während der Analyse ist es damit nicht erforderlich, an dieser Stelle parametrisierte Typen speziell zu behandeln, da alle kritischen Fälle bereits vom Compiler aussortiert worden sind. Gleiches gilt für Exceptions.

Die Kriterien, die für eine Überschreibung erfüllt sein müssen, können nun an Hand der Methodendeskriptoren überprüft werden, die alle restlichen Informationen bereitstellen. Private Methoden wurden in den entsprechenden Datenstrukturen speziell markiert, sodass auch diese erkannt werden können.

Damit kann nun festgestellt werden, ob eine Methode an Stelle einer Interface- oder einer überschriebenen Methode verwendet worden sein kann.

Dies bezieht sich jedoch nur auf Interfaces und erweiterte Klassen, die selbst in der EAR-Datei liegen. Solche, die Teil der Laufzeitumgebung sind, werden hier noch nicht abgedeckt.

Wurden solche Klassen verwendet, so werden diese bei der Analyse in einer Liste `notFoundClasses` gesammelt, da die betreffenden `class`-Dateien nicht aufgefunden werden können. Hier ist dann auch ein Verweis zu den jeweils benötigten Methoden gespeichert.

Wurde nun beispielsweise die Verwendung der Interface-Methode `java.util.Collection.add` innerhalb der analysierten Applikation festgestellt und implementiert eine Klasse `C` aus der EAR-Datei dieses Interface (und damit auch die Methode), so kann auch das festgestellt werden.

Das Problem ist nun, dass dies nur für direkte Implementierungen bzw. Erweiterungen möglich ist. Implementiert die oben geschilderte Klasse anstatt `java.util.Collection` das davon abgeleitete Interface `java.util.List`, so implementiert sie natürlich auch (indirekt) `Collection`. In der `class`-Datei werden aber nur direkte Implementierungen und Superklassen abgespeichert. Es muss also noch die fehlende Information bestimmt werden, wie die beiden Interfaces zusammenhängen.

Ist dies nicht möglich, so bleibt keine andere Wahl, als davon auszugehen, dass die betroffene Methode von `C` benötigt wird.

Bevor dies aber notwendig wird, kann man ausnutzen, dass das hier erstellte Analysewerkzeug selbst ein Java-Programm ist und somit auch in einer Java-Laufzeitumgebung ausgeführt werden muss, auch wenn dies in der Regel nicht die gleiche ist wie die der analysierten Anwendungen. Trotzdem ist es möglich, einen Großteil der abgehenden Informationen zu ermitteln.

Die Lösung heißt *Reflection* (vgl. Abschnitt 2.4). Mit Hilfe dieses Konzepts können implementierte Interfaces und Superklassen einer Klasse, deren Name in der Variablen `className` gehalten wird, so bestimmt werden:

```
Class c = Class.forName(className);

/* implementierte Interfaces */
for(Class interfaceClass : c.getInterfaces())
    checkClass(interfaceClass.getName(), methods);

/* Superklasse */
checkClass(c.getSuperclass().getName(), methods);
```

Durch den Aufruf der Methode `checkClass` wird wieder überprüft, ob aus der angegebenen Klasse einige der Methoden verwendet worden sind, die gerade untersucht werden (dies sind die aus der Liste `methods`).

Zum Schluss der Untersuchung werden die Methoden als neue Abhängigkeiten gespeichert, von denen festgestellt wurde, dass sie an Stelle einer Interfacemethode oder einer Methode der Superklasse aufgerufen worden sein könnten. Von ihnen ausgehend wird nun eine weitere Abhängigkeitsanalyse gestartet.

Damit terminiert der gesamte Algorithmus erst dann, wenn während der Behandlung von Interfaces und Klassenerweiterungen keine nicht verwendeten Methoden mehr gefunden werden können, die eine benutzte Interfacemethode implementieren oder eine benutzte Superklassenmethode überschreiben.

Die Terminierung selbst ist dadurch sichergestellt, dass im Algorithmus keine Kreise auftreten können, da Abhängigkeiten von bereits verwendeten Klassen und Methoden kein zweites Mal behandelt werden.

4.4. Behandlung von Spezialfällen

Im vorgestellten Algorithmus wurde die Behandlung bestimmter Spezialfälle der analysierten Anwendungen bisher ausgelassen. Wie mit diesen umgegangen wird, ist Gegenstand dieses Abschnitts.

4.4.1. Reflection

Die Verwendung von Reflection ist grundsätzlich ein Problem bei einer statischen Analyse, da hier Objektinstanzen von zur Compilezeit unbekannten Klassen erzeugt werden. Trotzdem muss dies natürlich berücksichtigt werden. Im Fall des PPI-Produkts TRAVIC Retail erfolgt die Nutzung von Reflection nach bestimmten Mustern. Diese erlauben auch die Bewältigung eines solchen dynamischen Konzepts.

Generell wird eine Klasse über Reflection geladen, in dem als Argument ihr Name als `String`-Wert verwendet wird:

```
Class c = Class.forName("myPackage.MyClass");
Object o = c.newInstance();
```

Es gibt zwei mögliche Arten, nach denen dieser `String`-Wert angegeben wird:

1. Konstante String-Werte

Wird wie in obigem Beispiel der Name der entsprechenden Klasse im Quellcode direkt angegeben, so wird beim Kompilieren der Wert des `Strings`, der dort bereits bekannt ist, innerhalb des Konstanten-Pools der `class`-Datei abgespeichert. Dies ist immer dann der Fall, wenn zuvor eine Konstante deklariert wurde, die als Argument der Methode `forName` dient oder wenn der Wert wie im Beispiel direkt dort eingesetzt wird.

Im Instruktionssatz der JVM existieren nun zwei besondere Befehle `LDC` und `LDC_W`, die als Argument einen Index in den Konstanten-Pool erwarten. An dieser Position ist eine Konstante vom Typ `int`, `float` oder `String` gespeichert. Die `LDC`-Instruktionen laden nun diese Konstante und legen sie oben auf dem Programmstack ab.

Beim Aufruf der Methode `forName` ist der Wert, der ihr als Argument übergeben wird, der oberste auf dem Stack. Ist dieses Argument ein konstanter String, so wurde direkt vor dem Methodenaufruf dessen Wert mit einer der `LDC`-Anweisungen auf den Stack geladen.

Beim Parsen des `Code`-Attributs in einer `class`-Datei muss nun überwacht werden, wann eine `LDC`-Instruktion verwendet wurde. Der geladene `String`-Wert kann über den Konstanten-Pool bestimmt werden. Ist die folgende Anweisung ein Aufruf von `Class.forName`, so ist der Name der geladenen Klasse derjenige, der im ermittelten `String`-Wert gespeichert ist.

Ein Instanzobjekt der geladenen Klasse kann dann mit `Class.newInstance` erzeugt werden. Für die Analyse ist es nun notwendig, dass diese Instanziierung direkt hinter `Class.forName` erfolgt, da sonst eine Zuordnung zu dem `Class`-Objekt, von dem der Aufruf gestartet wurde, nicht ohne Weiteres möglich ist. Ein denkbares Szenario, das dies verdeutlicht, ist folgendes:

```
void myMethod1 () {
    Class c = Class.forName("myPackage.MyClass");
    myMethod2(c);
}

void myMethod2 (Class c) {
    Object o = c.newInstance();
}
```

Im Allgemeinen müsste hier der komplette Kontrollfluss bzw. alle mögliche Kontrollflüsse zwischen den beiden Aufrufen identifiziert werden, um die Zuordnung zu bestimmen. Andernfalls ist es nicht möglich, herauszufinden, welche Klasse sich hinter dem Argument von `myMethod2` verbirgt. Im Fall von TRAVIC Retail wird die geforderte Richtlinie jedoch eingehalten.

Anzumerken ist noch, dass die Methode `newInstance` den Konstruktor der entsprechenden Klasse aufruft, der keine Argumente erwartet. Gibt es auch Konstruktoren mit Argumenten, so können diese über

java.lang.reflect.Constructor.newInstance(Object[]) aufgerufen werden, wobei das Object-Array gerade die Argumente enthält. Da die Typen dieser Argumente genauso schwer zu ermitteln sind wie obige Zuordnung, werden in so einem Fall alle Konstruktoren der entsprechenden Klasse als Abhängigkeiten ins Ergebnis übernommen.

2. Variable String-Werte

Die zweite, weitaus problematischere Variante bei der Verwendung von Reflection ist die Übergabe von variablen String-Werten als Argument von Class.forName. Hier ist es unmöglich im Rahmen einer statischen Analyse festzustellen, welche Klasse dort geladen wurde.

An dieser Stelle muss nun der Nutzer des Analyseprogramms angeben, welche Klassen hier verwendet worden sein können. Prinzipiell sind erst einmal alle Klassen aus der EAR-Datei oder der Laufzeitumgebung mögliche Kandidaten.

Bei TRAVIC Retail lässt sich diese Menge aber deutlich reduzieren, da der Aufruf auch hier nach einem bestimmten Muster abläuft. Es wird für die analysierten Anwendungen vorausgesetzt, dass die Reflection-Benutzung nach genau diesem erfolgt. Ein Beispiel dafür ist folgender Code-Auszug:

```
Class providerClass = Class.forName(configEntry);
ReportProvider provider = (ReportProvider) providerClass.newInstance();
```

Aus einer externen Konfigurationsdatei wird ein Klassenname geladen und in der Variablen configEntry gespeichert. Zu diesem wird das entsprechende Class-Objekt erzeugt und die zu Grunde liegende Klasse (hier: ReportProvider) instanziert. Diese hat zunächst den Typ java.lang.Object. Nun wird ein Type-Casting gegen eine Klasse ReportProvider durchgeführt.

Genau hier ist der Ansatzpunkt, die Auswahlmenge einzugrenzen. An diesem Punkt weiß man, dass die geladene Klasse gegen ReportProvider castbar sein muss. Ist dies ein Interface, so muss die geladene Klasse dieses implementieren. Ansonsten muss sie ReportProvider erweitern oder natürlich diese Klasse selbst sein. Ebenfalls muss die Klasse über einen Konstruktor verfügen (wegen des Aufrufs von newInstance). Sie kann also weder ein Interface noch abstrakt sein.

Die Auswahlmenge kann nun bestimmt werden, indem alle Klassen aus dem Klassenpfad des Archivs, in dem der Reflection-Aufruf erfolgte, darauf überprüft werden, ob sie die genannten Eigenschaften erfüllen. Die notwendigen Informationen wurden bereits vor der eigentlichen Analyse bestimmt (vgl. Abschnitt 4.3.4). Dies ist auch der Grund, warum dort bereits alle Klassen entpackt und geparst werden mussten.

Der Nutzer des Analysewerkzeugs kann nun aus der Menge der möglichen Klassen beliebig viele auswählen, die an der entsprechenden Stelle geladen werden können. Diese werden dann als neue Abhängigkeiten übernommen.

Nun bleibt noch zu klären, wann diese Interaktion mit dem Nutzer erfolgen soll. Es ist sicherlich keine gute Lösung, direkt nach jedem Reflection-Aufruf die Analyse zu unterbrechen, um vom Nutzer die möglichen Klassen zu erfragen. Stattdessen wird die Interaktion gebündelt am Ende der Analyse durchgeführt.

Dazu werden alle per Reflection bedienten Klassen (wie im Beispiel ReportProvider) zunächst in einer Liste gesammelt. Zum Schluss wird dem Nutzer für jede dieser Klassen eine Auswahl an möglichen Implementierungen angeboten. Von den selektierten Klassen müssen dann natürlich wiederum die Abhängigkeiten ermittelt werden.

Bei der Speicherung der Klassen, die per Reflection bedient wurden, ist es wichtig, auch das Archiv zu vermerken, in dem der entsprechende Aufruf erfolgt ist. Dies hat zwei Gründe. Zum Einen müssen die Klassen, die dem Nutzer als mögliche Implementierungen angeboten werden, im Klassenpfad dieses Ar-

chivs liegen. Dieser ist dann bei der Bestimmung der Auswahlmenge zu berücksichtigen. Zum Anderen soll dieser Klassenpfad später auch aktualisiert werden, d.h. im entsprechenden Archiv muss vermerkt werden, zu welchen anderen JARs noch direkte Abhängigkeiten bestehen. Hier wären das diejenigen, aus denen der Nutzer Klassen ausgewählt hat.

4.4.2. „Version"-Klassen

In TRAVIC Retail gibt es eine spezielle Menge von Klassen, die Versionsinformationen über die einzelnen Archive bereithalten. Sie kennzeichnet das Suffix „Version" im Klassennamen. Fast jedes Archiv enthält eine solche Klasse. Diese beinhaltet auch Referenzen zu den „Version"-Klassen von den von ihm referenzierten Archiven.

Bei unterschiedlichen Auslieferungsumfängen kann es durchaus vorkommen, dass sich diese Archivreferenzen voneinander unterscheiden. Um die betreffenden „Version"-Klassen nicht jedes Mal ändern zu müssen, bedient sich man des Reflection-Konzepts, wie folgendes Beispiel verdeutlicht:

```
try {
    referenced.add(Class.forName("server.ServerVersion").newInstance());
} catch (ClassNotFoundException e) { }
```

Der Vorteil hierbei ist der folgende: Wurde eine Referenz aufgebrochen, d.h. existiert das Archiv, auf das verwiesen wird, in der Anwendung nicht, so wird bei obigem Aufruf von `Class.forName` eine `ClassNotFoundException` geworfen, die dann gefangen werden kann. Auch wird die Klasse `ServerVersion` erst gar nicht zum Kompilieren obigen Codefragments benötigt.

Bei der bisher vorgestellten Abhängigkeitsanalyse würden solche Klassen immer in die neu zu erstellende Anwendung übernommen werden, auch wenn ansonsten nichts anderes aus dem betreffenden Archiv gebraucht wird. Dieses würde in der neuen Applikation dann nur aus der „Version"-Klasse bestehen. Das widerspricht nun gerade dem Ziel, dass durch die Reflection-Verwendung angestrebt wurde.

Die „Version"-Klasse soll in diesem Fall nicht in die neue Anwendung, d.h. auch nicht die Abhängigkeitsmenge übernommen werden, da sie eigentlich nicht gebraucht wird. Für diese Klassen ist somit eine Sonderbehandlung nötig.

Durch die charakteristische Namensendung „Version" und die Verwendung des Klassennamens als konstanten Argumentwert für `Class.forName` lassen sich nun leicht die betreffenden Klassen ermitteln. Sie werden erst dann in die Abhängigkeitsmenge eingefügt, sobald diese auch andere Klassen aus dem gleichen Java-Archiv enthält.

4.4.3. Automatisch generierte Klassen

Am Ende von Abschnitt 2.2 wurde darauf eingegangen, dass eine EJB-Klasse Implementierungen der Methoden für die zwei zugehörigen Home- und Remote-Interfaces bereitstellt, diese aber nicht direkt implementiert. Stattdessen wird vom Laufzeitsystem automatisch ein Implementierung aus der EJB-Klasse heraus erzeugt. Mit dieser zusammen werden noch einige weitere Klassen generiert, die zur Verwendung des betreffenden EJBs innerhalb des Application-Servers gebraucht werden.

Sie sind für die korrekte Ausführung einer JavaEE-Anwendung notwendig, jedoch existieren keine direkten Abhängigkeiten der bisher analysierten Klassen zu ihnen. Somit ist hier eine Spezialbehandlung erforderlich.

Erkennbar sind diese Klassen daran, dass ihr Name mit einem "_" beginnt. Zu jeder EJB werden einige solcher Klassen erzeugt. Aber natürlich dürfen die nur dann in die Abhängigkeitsmenge übernommen werden, wenn die zu Grunde liegende Enterprise JavaBean auch dort enthalten ist.

Also müssen die richtigen generierten Klassen erst noch herausgefiltert werden. Dies kann an Hand der Paket- bzw. Verzeichnisstruktur innerhalb eines Archivs erfolgen. Denn diese Klassen werden im selben Paket (und damit auch im selben Verzeichnis) gespeichert wie die betreffenden Home- und Remote-Interfaces.

Im Anschluss an die bisher vorgestellten Analyseschritte wird also nach Klassen gesucht, deren Name mit einem "_" beginnt. Existiert im gleichen Verzeichnis eine Klasse, die bereits in der Resultatsmenge enthalten ist, so wird die "_"-Klasse als Abhängigkeit übernommen. Klassen, die von ihr benötigt werden, müssen dann natürlich ebenfalls noch ermittelt werden.

Es können aber noch weitere Klassen existieren, die vom System automatisch erzeugt wurden und zur Ausführung der Anwendung benötigt werden, ohne dass direkte Abhängigkeiten zu ihnen existieren.

Im Fall von TRAVIC Retail sind diese Klassen an Hand des Paketnamens erkennbar. Alle von PPI entwickelten Klassen liegen in einem Paket `de.ppi.fis` (in den bisher vorgestellten Beispielen wurde dieses Namenspräfix aus Übersichtlichkeitsgründen weggelassen). Dies betrifft auch die "_"-Klassen. Die Übrigen jedoch sind in Paketen mit anderen Präfixen gespeichert, wodurch sie identifizierbar sind.

Sie werden nun genau dann in die Abhängigkeitsmenge eingefügt, wenn diese bereits Klassen enthält, die aus demselben Archiv stammen. Dies betrifft nur Archive, die ein JavaEE-Modul enthalten, also keine Klassenbibliotheken.

Durch die Berücksichtigung der hier dargestellten Spezialfälle ist die Einsatzmöglichkeit des Analysewerkzeugs natürlich zunächst einmal sehr auf das PPI-Produkt TRAVIC Retail eingeschränkt. Um diesen Nachteil zu reduzieren werden dem Nutzer einige Konfigurationsmöglichkeiten zur Verfügung gestellt, die in Abschnitt 4.6 vorgestellt werden.

Damit ist die Analyse der Abhängigkeiten zwischen Java-Klassen abgeschlossen. Das Ergebnis wurde in der Variablen `mainArchive` vom Typ `Archive` abgespeichert, die `JavaClass`-Objekte für jede Klasse der Anwendung strukturiert nach Archiven enthält. In diesen wurde vermerkt, ob die entsprechende Klasse in die zu erstellende JavaEE-Applikation zu übernehmen ist, d.h. ob sie in der Abhängigkeitsmenge enthalten ist

4.5. Erstellung der neuen Applikation

Es wurden nun alle Informationen bestimmt, die nötig sind, um eine neue JavaEE-Anwendung zu erstellen. Es kann also begonnen werden, eine neue Applikation aus der alten zusammenzusetzen.

Der erste Schritt ist es, die einzelnen JARs und WARs zu erzeugen, ehe diese dann zu einer neuen EAR-Datei vereinigt werden. Das Schema ist für beide Fälle das gleiche.

Der Inhalt eines jeden Archivs wurde zu Beginn der Analyse in ein temporäres Verzeichnis entpackt. Dieses dient nun als Basis für die Erzeugung der neuen Archive.

Zunächst wird die alte Manifestdatei des Archivs durch eine neue mit einem aktualisierten Class-Path-Eintrag überschrieben. Im Falle von JavaEE-Modulen werden aktualisierte Deployment-Deskriptoren erstellt, wie in Abschnitt 3.6 bereits beschrieben. Auch diese werden an derselben Stelle gespeichert wie die Originale, d.h. sie überschreiben diese.

Soll ein zu erstellendes Archiv auch innere Archive enthaltenen (dies trifft insbesondere auf die EAR-Datei zu), so werden diese nach ihrer Erzeugung ebenfalls in dieses temporäre Verzeichnis verschoben.

Es enthält damit für jedes Archiv, in korrekter Verzeichnisstruktur, eine Obermenge der Dateien, die das neue bilden sollen.

Zum Packen der Archive wird ein Ant-Skript verwendet. Die Informationen darüber, welche `class`-Dateien nicht mit in das neue Archiv übernommen werden sollen, werden über eine individuell definierte Properties-Datei mitgegeben. Bei dieser handelt es sich um eine Textdatei, wo in jeder Zeile eine Datei deklariert wird, die vom Packen ausgeschlossen werden soll.

Das auszuführende Ant-Target ist dann wie folgt definiert:

```
<target name="packArchive" description="Packen eines Archivs">
    <!-- Erstelle das Archiv, lasse dabei Klassen aus ${temp.dir}/exclude.txt weg -->
    <zip destfile="${destination.file}" filesonly="true">
        <fileset dir="${src.dir}" excludesFile="${temp.dir}/exclude.txt"/>
    </zip>
</target>
```

Die Properties `destination.file` und `src.dir` geben die zu erstellende Archivdatei bzw. das oben beschriebene temporäre Quellverzeichnis an. Sie können beim Ant-Aufruf direkt übergeben werden. Die Task `zip` erzeugt Archive im ZIP-Format, also auch JARs, WARs und EARs.

Für jedes neu zu erstellende Archiv wird dieses Target einmal ausgeführt. Der letzte Aufruf erzeugt die gewünschte EAR-Datei.

Zum Abschluss werden die erzeugten temporären Dateien wieder gelöscht sowie eine Übersicht über das Ergebnis der Analyse in Form von XML-Dateien erzeugt. Diese enthalten Informationen darüber, aus welchen Klassen und Archiven sich die neue Anwendung zusammensetzt und welche Probleme bei der Analyse auftraten. Dabei kann es sich zum Beispiel um benötigte Klassen handeln, die in der EAR-Datei nicht gefunden werden konnten. Ferner werden auch die Klassen aus der Original-EAR-Datei aufgelistet, die nicht mit in die neue Applikation übernommen wurden.

Gerade der letzte Punkt liefert ein gutes Hilfsmittel, um zu ermitteln, wo in der analysierten Anwendung nicht mehr verwendeter Code vorhanden ist. So etwas ist in großen Softwareprojekten häufiger der Fall, da sich die Wartung auf Grund mangelnder Transparenz oft als schwierig herausstellt. Das in dieser Arbeit vorgestellte Werkzeug kann hier Abhilfe leisten.

4.6. Konfigurationsmöglichkeiten

Zur Berücksichtigung gewisser Spezialfälle der betrachteten Applikationen, aber auch zur Steigerung der Effizienz der Analyse wurden verschiedene Möglichkeiten zur Konfiguration des Werkzeugs geschaffen, die hier kurz präsentiert werden sollen.

Ignorierbare Pakete

Wie bereits erläutert wurde, sind in der analysierten EAR-Datei in der Regel nicht alle Java-Pakete enthalten, die von der Anwendung benötigt werden. Sie werden von der Laufzeitumgebung zur Verfügung gestellt. Klassische Beispiele dafür sind `java.lang` oder `java.util`.

Wurden Klassen dieser Pakete während der Analyse als Abhängigkeiten ermittelt, so wird der anschließende Versuch, sie in der EAR-Datei aufzuspüren, erfolglos verlaufen. Um diese überflüssige Suche zu vermeiden, kann der Nutzer vor der Analyse angeben, ob Abhängigkeiten zu bestimmten Paketen ignoriert werden sollen. Wird also festgestellt, dass zum Beispiel die Klasse `java.util.List` gebraucht wird, so wird die gefundene Abhängigkeit einfach fallen gelassen.

Bei der Analyse auf Methodenebene ist hier jedoch Vorsicht geboten. Wird beispielsweise die Interfacemethode `java.util.List.add` aufgerufen, so ist es natürlich möglich, dass die Implementierung, die sich hinter dem Interface verbirgt, eine Klasse aus der EAR-Datei ist. Insofern ist es notwendig, bei

der Behandlung von Interfaces und Klassenerweiterungen (vgl. Abschnitt 4.3.7) auch die „ignorierten“ Klassen zu berücksichtigen.

Klassenbibliotheken von Dritten

Innerhalb einer EAR-Datei finden häufig auch Klassenbibliotheken Dritter Anwendung. Zum Beispiel dienen die Archive `xml-apis.jar` und `xercesImpl.jar` im Rahmen von TRAVIC Retail zur XML-Behandlung.

Bei der Generierung einer neuen Anwendung kann es nun gewünscht sein, dass solche Bibliotheken grundsätzlich vollständig in diese eingefügt werden. Daher wird dem Nutzer die Möglichkeit gegeben, einzustellen, welche Archive einer solchen Spezialbehandlung unterzogen werden sollen.

Wird bei der Abhängigkeitsanalyse nun eine Abhängigkeit zu einer Klasse aus so einem Archiv gefunden, so wird das komplette JAR samt aller von ihm abhängigen übernommen.

Spezielle Reflection-Klassen

In Abschnitt 4.4.2 wurde die besondere Behandlung der „Version“-Klassen im Rahmen von Reflection vorgestellt. Hier wurde eine spezielle Art von Klassen an Hand des Suffixes „Version“ definiert. Um diese Identifizierung möglichst variabel zu gestalten, ist das entsprechende Muster für den Klassennamen vom Nutzer konfigurierbar.

Wiederverwendbarkeit alter Konfigurationen

Um eine durchgeführte Analyse später leicht wiederholen zu können, besteht die Möglichkeit, die vom Nutzer angegebenen Parameter in einer XML-Datei abspeichern zu können. Diese kann dann bei einer neuen Analyse geladen werden, sodass die hier getroffenen Einstellungen wiederverwendet werden können. Diese Parameter umfassen die Auswahl der JavaEE-Komponenten, die die neue Anwendung bilden sollen, die bei der Reflection-Behandlung getätigte Klassenauswahl, die Wahl der Analysetiefe sowie alle Konfigurationsmöglichkeiten, die in diesem Abschnitt vorgestellt worden sind.

4.7. Vergleich Klassen- und Methodenebene

Am Ende dieses Kapitels sollen die zwei vorgestellten Alternativen bezüglich der Analysetiefe in Hinsicht auf Laufzeit, Größe der generierten Applikationen und Verwendung in der Praxis miteinander verglichen werden.

Abbildung 4.1 zeigt vier typische Beispiele für die Generierung einer neuen Applikation, an Hand derer Vor- und Nachteile der beiden Analyseebenen (abgekürzt durch K für Klassenebene und M für Methodenebene) verdeutlicht werden sollen.

Durchgeführt wurde der Test auf einem Rechner mit Intel Pentium 4 CPU mit 1,6 GHZ und 1 GB RAM. Als Betriebssystem wurde Windows XP verwendet.

Grundlage für die Analyse sind zwei verschiedene EAR-Dateien, die jeweils einer Auslieferung des PPI Produkts TRAVIC Retail entsprechen. Die Beispiele 1 und 2 basieren auf der größten betrachteten JavaEE-Applikation mit über 16.000 Java-Klassen, die für den IBM Websphere Application Server konfiguriert wurde. Die beiden anderen Beispiele basieren auf einer Applikation für den JBoss Application Server.

	Beispiel 1		Beispiel 2		Beispiel 3		Beispiel 4	
Analyseebene	K	M	K	M	K	M	K	M
Gesamtanzahl Komponenten	83	83	83	83	32	32	32	32
Ausgewählte Komponenten	83	83	6	6	32	32	1	1
Benötigte Komponenten	83	83	65	65	32	32	20	20
Gesamtanzahl Klassen	16466	16466	16466	16466	7481	7481	7481	7481
Eingabe-Klassen	260	260	208	208	93	93	67	67
Benötigte Klassen	13653	12563	10129	9016	6931	6810	4174	4166
Vergleich Klassenanzahl	$M/K = 92\%$		$M/K = 89\%$		$M/K = 98\%$		$M/K = 99\%$	
Laufzeit Gesamt	362 s	509 s	279 s	429 s	140 s	201 s	122 s	152 s
Laufzeit Entpacken	177 s	177 s	150 s	150 s	78 s	78 s	69 s	69 s
Laufzeit Analyse	126 s	279 s	68 s	223 s	33 s	95 s	24 s	56 s
Laufzeit Packen	59 s	53 s	61 s	56 s	29 s	28 s	29 s	27 s
Vergleich Laufzeit	$K/M = 71\%$		$K/M = 65\%$		$K/M = 70\%$		$K/M = 80\%$	

Abbildung 4.1.: Praktischer Vergleich der Analyseebenen

Um die beiden Analyseebenen miteinander vergleichen zu können, müssen bei den jeweiligen Testläufen natürlich die gleichen Konfigurationseinstellungen vorgenommen worden sein. Ebenso ist die Auswahl der Klassen im Rahmen der Reflection-Behandlung identisch.

Es ist jeweils angegeben, aus wie vielen JavaEE-Komponenten die Anwendung besteht, wie viele davon der Nutzer für die neue Applikation ausgewählt hat und wie groß die Anzahl an Komponenten ist, die von diesen benötigt werden. Deren Einstiegspunkte in Form von Java-Klassen werden hier als Eingabe-Klassen bezeichnet.

Der erste entscheidende Messwert ist nun die Anzahl der für die neue Anwendung benötigten Klassen, gefolgt vom prozentualen Anteil der Klassen auf Methodenebene von denen der Analyse auf Klassenebene.

Anschließend werden die Laufzeiten für die einzelnen Phasen des Analysewerkzeugs einander gegenübergestellt. Die hier angegebenen absoluten Zahlen hängen natürlich stark von der Leistung des Systems ab, auf dem die Analyse durchgeführt wurde. Es ist jedoch weniger die reine, absolute Zeit von Interesse, sondern das Verhältnis der Laufzeiten auf den verschiedenen Analyseebenen.

Die erste Phase ist das Entpacken der Archive inklusive des Parsens zur Bestimmung von Basisinformationen der Klassen. Es folgt die Laufzeit der eigentlichen Analyse, ehe die fürs Packen der neuen Anwendung angegeben wird. Den Abschluss bildet die Angabe des prozentualen Verhältnisses der Gesamtlaufzeiten.

In den Beispielen 1 und 3 wurden nun alle Komponenten der Applikation ausgewählt. Eine erste Beobachtung dabei ist, dass nicht alle Klassen der EAR-Datei wirklich gebraucht werden. In den zwei anderen Fällen wurde lediglich ein kleiner Teil der Komponenten ausgewählt, der jedoch Abhängigkeiten zu vielen weiteren aufweist.

Wie erkennbar ist, werden bei einer Analyse auf Methodenebene in den Beispielen 1 und 2 nur rund 90% der Klassen gebraucht, die bei einer Analyse auf Klassenebene ermittelt wurden. Bei den Beispielen 3 und 4 liegt der Unterschied hingegen nur bei 1-2%. Ob eine Analyse auf Methodenebene nun wirklich effektive Vorteile bezüglich der Größe der neuen Applikation bietet, hängt somit stark von der Struktur der Originalanwendung ab.

Die Ursachenforschung im Fall von Beispiel 1 hat ergeben, dass mehrere vom Aufbau her verwandte Klassen jeweils eine Methode besitzen, die innerhalb der Anwendung nie aufgerufen wird. In diesen wird exklusiv je eine Klasse verwendet, die daher ebenfalls nie benötigt wird.

Im Bereich der Laufzeiten wird deutlich, dass der Analyseanteil auf Methodenebene zwei- bis dreimal so groß ist wie der auf Klassenebene. Dies ist sicherlich nicht verwunderlich, da hier zum Einen `class`-Dateien mehrfach geparst werden müssen und zum Anderen auch noch die spezielle Behandlung von Interfaces und Klassenerweiterungen hinzukommt.

Es fällt weiterhin auf, dass die Zeit für das Entpacken der Archive einen hohen Anteil an der Gesamtlaufzeit besitzt, teilweise sogar den dominierenden. Die Ursache davon liegt in der immensen Datenmenge, die dort extrahiert werden muss, im Fall von Beispiel 1 betrifft dies fast 80 MB. Dies führt auch dazu, dass das Löschen dieser temporär erstellten Dateien, was hier nicht betrachtet wurde, einen ebenfalls sehr hohen Zeitbedarf besitzt.

Die Auswirkungen der letzten zwei Punkte können aber durch die Verwendung der im folgenden Kapitel 5 vorgestellten GUI abgeschwächt werden. Sollen dort mehrere Analysen auf derselben EAR-Datei durchgeführt werden, so erfolgt das Entpacken nur vor der ersten, das Löschen temporärer Dateien nur nach der letzten Analyse.

Insgesamt betrachtet ist eine Analyse auf Klassenebene im Schnitt um 30% schneller, liefert aber genauere Ergebnisse.

Abschließend soll noch darauf eingegangen werden, wie in der Praxis die zwei unterschiedlichen Analyseebenen Verwendung finden. Die verschiedenen Laufzeiten spielen hierbei keine große Rolle.

Die Analyse auf Methodenebene ist zunächst ein gutes Mittel, um Hinweise darüber zu bekommen, ob es bestimmte Methoden gibt, die innerhalb der Applikation nicht mehr verwendet werden. Die Basis dafür liefert ein Vergleich mit der Menge an Klassen, die nur auf Klassenebene als Abhängigkeiten ermittelt wurden.

Ebenso liefert diese Variante bei der Generierung einer neuen JavaEE-Applikation einen geringeren Umfang, was zunächst einmal natürlich wünschenswert ist. Dennoch kann dieser in bestimmten Fällen auch zu klein sein.

Im Rahmen von TRAVIC Retail ist es möglich, dass der Kunde die ausgelieferte Anwendung um selbst erstellte Programmkomponenten erweitert bzw. Teile von dieser durch solche ersetzt. Dabei kann er dann Klassen und Methoden aus der Originalanwendung benutzen. Ist unter diesen nun eine Methode, die in der Applikation bislang nicht verwendet worden ist, so fehlen unter Umständen von dieser abhängige Klassen in der Auslieferung. In einem solchen Fall, in dem der Kunde plant die Anwendung selbst noch zu verändern, ist es daher notwendig, dass die Generierung der EAR-Datei durch eine Analyse auf Klassenebene erfolgt, wo die beschriebene Problematik nicht auftreten kann.

5. Visualisierung im Rahmen einer graphischen Benutzeroberfläche

Zur komfortablen Ansteuerung des Analysewerkzeugs wurde mit Hilfe von *Java Swing* [13] eine graphische Benutzeroberfläche entwickelt. In diese wurde eine Visualisierung der Abhängigkeitsstruktur der JavaEE-Komponenten integriert, über die direkt die Auswahl der Komponenten erfolgen kann, die die neue Anwendung bilden sollen. Wie dies umgesetzt worden ist, wird in diesem Kapitel erläutert.

5.1. Visualisierung der Abhängigkeiten zwischen JavaEE-Komponenten

In Abschnitt 3.5 wurde bereits eine Datenstruktur zur Darstellung der JavaEE-Komponenten in Form eines Graphen beschrieben. Auf dieser baut nun die Visualisierung auf.

Nachdem der Nutzer die zu analysierende EAR-Datei ausgewählt hat, wird als Erstes die oben erwähnte Datenstruktur erzeugt. Über die GUI können dann die gewünschten Komponenten selektiert werden.

5.1.1. Beschreibung der Visualisierung

Abb. 5.1 zeigt die fertige Visualisierung. Wie diese realisiert wurde, wird im Folgenden beschrieben.

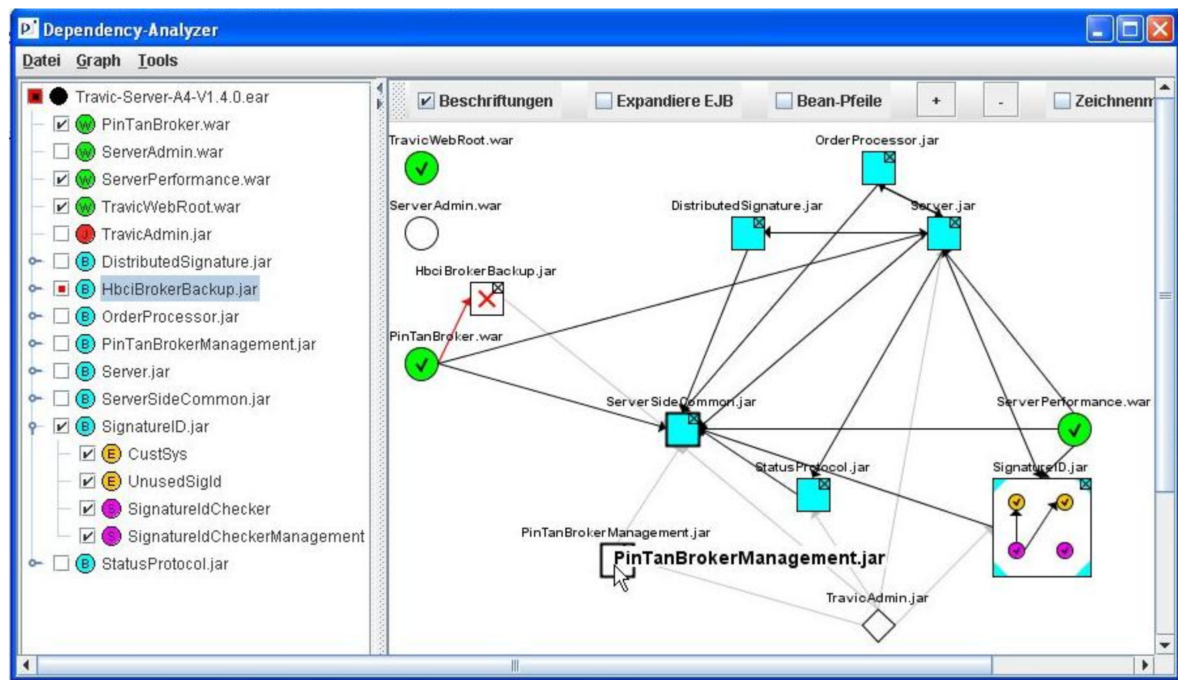


Abbildung 5.1.: Visualisierung der Abhängigkeitsstruktur im Rahmen der GUI

Es ist zunächst eine Teilung der GUI in zwei Hälften erkennbar. Auf der linken Seite ist eine Auflistung der Komponenten der Applikation in Form einer Baumstruktur zu sehen. Die unterschiedlichen Arten von JavaEE-Komponenten werden hierbei durch verschiedenfarbige Symbole gekennzeichnet. Zu beachten ist, dass sich ein EJB-Modul (hellblau) aus mehreren einzelnen Komponenten, den Enterprise JavaBeans, zusammensetzt, die durch Auf- und Zuklappen des übergeordneten Modulelements angezeigt bzw. verborgen werden können.

Um schnell bestimmte Komponenten finden zu können, sind diese nach ihrer Art und ihrem Namen sortiert. Jedes Element ist mit einer `JCheckBox` versehen, über die dieses mit einem Linksklick der Maus selektiert werden kann. Ein Rechtsklick bewirkt, dass es nicht mit in die neue Anwendung übernommen wird (gekennzeichnet durch roten Punkt in der CheckBox). Bestehende Abhängigkeiten zu ihm werden dann ignoriert.

In der rechten Hälfte des Fensters ist die Abhängigkeitsstruktur der JavaEE-Applikation dargestellt. Die Unterscheidung der Komponentenarten wird hier zunächst durch verschiedene Knotenformen realisiert. Dabei werden EJB-Module als Superknoten dargestellt, die einzelne EJB-Knoten enthalten. Sie können ähnlich wie die Checkboxen durch einen Klick in die rechte obere Ecke auf- und zugeklappt werden, um die EJBs anzuzeigen oder auszublenden. In Abbildung 5.1 wurde beispielsweise der Knoten „SignatureID.jar“ auf diese Weise expandiert.

Die Auswahl der Komponenten kann analog zu den Checkboxen erfolgen, d.h. durch einen Linksklick auf einen Knoten wird dieser in die neue Anwendung übernommen (zu sehen an einem Häkchen in der Knotenmitte), durch einen Rechtsklick aus dieser entfernt (erkennbar an einem roten Kreuz).

Zusätzlich werden die direkt oder indirekt von ihm abhängigen Knoten eingefärbt. Die entsprechenden Komponenten werden die neue Anwendung bilden. Der Nutzer kann also direkt verfolgen, welche Auswirkungen seine Auswahl auf die neue Applikation haben wird.

Die direkten Abhängigkeiten selbst werden durch Pfeile repräsentiert. Sie werden im Normalfall grau gezeichnet. Ist die betreffende Abhängigkeit aktiv, d.h. Start- und Zielknoten sind Bestandteil der neuen Anwendung, so werden sie schwarz gefärbt und rot, wenn eine Abhängigkeit aufgebrochen wurde.

Um von einem bestimmten Knoten dessen direkt von ihm benötigten Komponenten besser hervorzuheben, gibt es noch eine weitere Möglichkeit. Wird der Mauszeiger über den betreffenden Knoten gehalten, so wird sein Name größer dargestellt (in Abb. 5.1 ist dies der Knoten „PinTanBrokerManagement.jar“) und es werden bei seinen direkten Abhängigkeiten die Knotenränder dicker gezeichnet.

Ferner gibt es zahlreiche Möglichkeiten, die Darstellung des Graphen zu konfigurieren. Einige davon sollen hier kurz aufgelistet werden:

- Auf-/ Zuklappen von EJB-Modul-Knoten
- Hinein- und Hinauszoomen
- Knotenbeschriftungen ein- und ausblenden
- Darstellung einschränken auf bestimmte Knoten: aktive, inaktive oder gelöschte
- Darstellung einschränken auf bestimmte Pfeile: aktive, inaktive oder gelöschte
- Manuelles Verschieben der Knoten

5.1.2. Graph-Layout

Eine der wichtigste Fragen beim Zeichnen ist: Wie sollen die Knoten angeordnet werden? Es ist natürlich wünschenswert, dass die Darstellung so übersichtlich wie möglich ist, d.h. dass voneinander abhängige

Knoten auch möglichst nah zusammen gezeichnet werden und Knoten, die in keiner Beziehung stehen, weiter weg voneinander.

LinLog-Layout

Zur Lösung des Problems in dieser Arbeit wurde das Programm LinLog-Layout verwendet. Andreas Noack entwickelte dieses aufbauend auf seinen Arbeiten [14], [15] und [16].

Hierbei werden die *Cluster* eines ungerichteten Graphen identifiziert. Unter einem Cluster versteht man eine Menge von Knoten, die viele Kanten untereinander, aber nur wenige zu Knoten außerhalb dieser Menge besitzen. Die Cluster (und deren Knoten) werden dann so angeordnet, dass die Distanz zwischen ihnen ihre Beziehung untereinander widerspiegelt.

Um solche Cluster bestimmen zu können, wird das in [14] und [15] vorgestellte LinLog-Energie-Model benutzt. Ein Energie-Model misst die Qualität eines Graph-Layouts. Je geringer die Energie ist, desto besser ist das Layout.

Der in Noacks Programm umgesetzte Algorithmus basiert nun gerade auf der Minimierung der Energie des Graph-Layouts. Die hierbei ermittelte Knotenanordnung ist jedoch nicht deterministisch. Zwar werden dieselben Cluster erkannt, die genaue Anordnung der Knoten in der Ebene kann sich aber durchaus unterscheiden.

Als Eingabe dient dem Programm eine Textdatei, in der zeilenweise die Kanten des Graphen definiert werden. Die Ausgabe ist wiederum eine Textdatei, in der ebenfalls zeilenweise zu jedem Knoten die X- und die Y-Koordinate der ermittelten Position angegeben werden.

Anordnen der Knoten

Es sei noch erwähnt, dass beim Layouten vorerst nur Knoten betrachtet werden, die ein ganzes JavaEE-Modul repräsentieren, d.h. die EJB-Knoten, die in einem EJB-Modul-Knoten enthalten sind, werden erst einmal weggelassen.

Die über das LinLog-Layout ermittelten Koordinaten dienen nun als Basis für das endgültige Anordnen der Knoten innerhalb der vorgestellten GUI. Sie liegen in Form von Gleitkommazahlen vor, können aber so nicht direkt verwendet werden.

Das kleinere Problem ist, dass beim Zeichnen die Werte für die Koordinaten natürliche Zahlen sein müssen. Entscheidender sind hier die teilweise sehr kleinen Abstände zwischen den Knoten. Diese würden beim Zeichnen zu Knotenüberschneidungen führen, da vor allem die EJB-Modul-Knoten im expandierten Zustand einen recht hohen Platzbedarf haben, wie in Abbildung 5.1 schon zu erkennen ist.

Als Zeichenfläche wird ein `javax.swing.JPanel` verwendet, dessen Größe im Wesentlichen von den Extremwerten der im LinLog-Layout ermittelten Koordinaten abhängt. Um die Knoten nun überschneidungsfrei anordnen zu können, wird diese Fläche durch ein Raster unterteilt. Jeder Knoten wird einem Feld aus diesem zugeordnet. Für einen EJB-Modul-Knoten reicht ein Feld in der Regel jedoch nicht aus, da dieser mehrere (kleinere) Subknoten für die einzelnen EJBs des Moduls enthalten kann. Also werden für ihn mehrere Felder reserviert, deren Zahl von der Anzahl seiner EJBs abhängt.

Die Knoten müssen nun gemäß ihrer Koordinaten in das Rasterfeld eingefügt werden, auf das die Koordinaten verweisen. Tritt der Fall auf, dass ein Feld bereits durch einen anderen Knoten belegt ist, so muss stattdessen ein benachbartes Feld verwendet werden. Die Frage ist nun, in welche Richtung diese Verschiebung erfolgen soll, sodass die vom LinLog-Layout berechnete Anordnung weitestgehend erhalten bleibt.

Beim LinLog-Layout-Algorithmus werden die Knoten um den Ursprung herum angeordnet. Die Richtung, in die ein Knoten verschoben wird, wird daher in Abhängigkeit von der relativen Position des Knotens zum Ursprung definiert. Teilt man die Zeichenfläche in vier Bereiche, so sind die Verschiebungsrichtungen für diese wie in Abbildung 5.2 definiert.

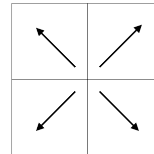


Abbildung 5.2.: Verschiebungsrichtungen

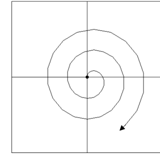


Abbildung 5.3.: Reihenfolge der Anordnung

Da die Verschiebung stets vom Ursprung weg erfolgt, wird mit den Knoten begonnen, die diesem am nächsten sind. Sie werden daher bezüglich ihrer Entfernung zum Nullpunkt sortiert und gemäß dieser ins Raster eingefügt. Die Reihenfolge der Knotenanordnung veranschaulicht Abbildung 5.3.

Tritt der Fall ein, dass ein Knoten nicht weiter verschoben werden kann, da die Grenze der Zeichenfläche erreicht worden ist, so wird für ihn eine beliebige freie Position im Raster gewählt.

Die Speicherung der Koordinaten findet nicht in absoluter Form statt, sondern es werden lediglich die Positionen der Knoten im Raster vermerkt. Der Vorteil davon wird später noch deutlich werden.

Auf diese Weise konnten nun alle Knoten angeordnet werden, mit Ausnahme derer, die die EJBs repräsentieren. Sie sind in dem Knoten für das jeweils übergeordnete EJB-Modul enthalten.

Die Anzahl von Beans pro Modul ist relativ gering (in den betrachteten Applikation maximal 9), sodass hier kein besonderes Layout nötig ist. Sie können beliebig auf die Rasterfelder, die ihr Superknoten umschließt, verteilt werden.

Manuelles Verschieben von Knoten

Nach der algorithmischen Bestimmung eines Graphlayouts kann der Nutzer dieses noch manuell verändern. Hierfür ist ein spezieller Zeichenmodus zu aktivieren.

Wie so ein manuelles Knotenverschieben durchgeführt werden kann, soll an Hand von Abbildung 5.4 beschrieben werden, die das Beispiel aus Abbildung 5.1 wieder aufgreift.

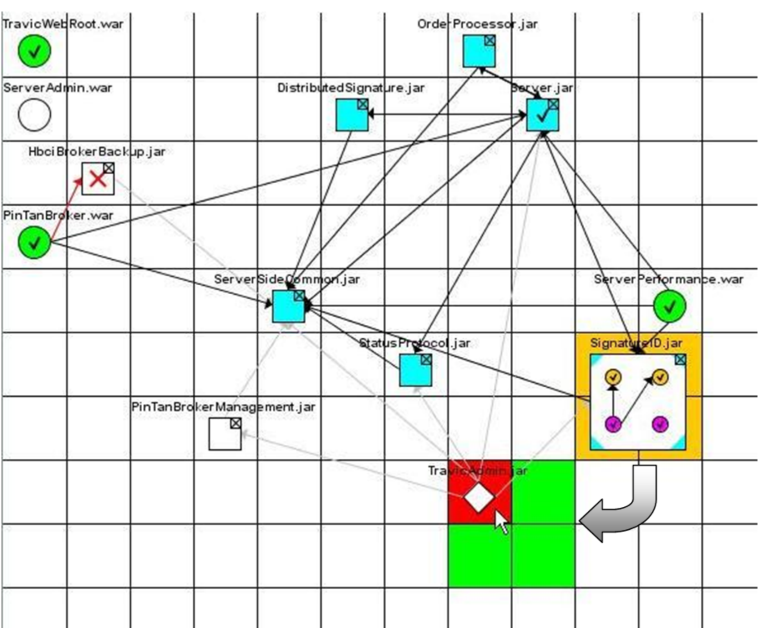


Abbildung 5.4.: Manuelles Knotenverschieben

Es ist zu erkennen, dass hier ein Gitternetz angezeigt wird. Dieses entspricht dem angesprochenen Raster. Durch einen Mausklick wird der zu verschiebende Knoten selektiert, die Felder auf denen er momentan liegt werden orange untermalt. Im Beispiel ist dies der Knoten „SignatureID.jar“, der vier Rasterfelder umfasst.

Bewegt man nun den Mauszeiger über die Zeichenfläche, so wird direkt angezeigt, ob der Knoten an die Position, über der der Cursor aktuell steht, verschoben werden kann. Ist ein benötigtes Feld von einem anderen Knoten besetzt, so wird dieses rot hinterlegt, andernfalls grün. Diese farbliche Markierung ist insofern nützlich, als dass bei einem EJB-Modul-Knoten im zugeklappten Zustand nicht erkennbar ist, wie viel Platz er eigentlich benötigt. Sind alle gewünschten Rasterpositionen frei, so kann der Knoten mit einem erneuten Mausklick hierhin verschoben werden.

Wiederverwenden eines Layouts

Hat der Nutzer ein ansprechendes Layout für die analysierte JavaEE-Applikation gefunden, so kann er dies in Form einer XML-Datei abspeichern, um es später bei einer erneuten Analyse derselben Anwendung wiederverwenden zu können. Dies ist hilfreich, da bei wiederholtem Aufruf des LinLog-Layout-Algorithmus mit der gleichen Eingabe in der Regel verschiedene Knotenanordnungen zurückgeliefert werden.

5.1.3. Zeichnen der Knoten

Es ist jetzt bekannt, wo die Knoten auf der Zeichenfläche positioniert werden sollen. Das Zeichnen selbst erfolgt mit Hilfe des *Abstract Window Toolkit (AWT)* [17], einer Java-API zur Erstellung von graphischen Benutzeroberflächen.

Dieses bietet unter anderem über die Klasse `java.awt.geom.Area` eine einfache Möglichkeit zur Darstellung verschiedener geometrischer Figuren, wie sie bei der Visualisierung der Graphknoten verwendet werden.

In der in Abschnitt 3.5 eingeführten Datenstruktur `Node` wird für den zu Grunde liegenden Knoten neben dessen Position im Raster auch eine `Area` gespeichert, die seine zeichnerische Darstellung repräsentiert. Diese liegt grundsätzlich in absoluter Form vor, d.h. es werden die direkten Koordinaten auf der Zeichenfläche verwendet.

Folgender Auszug aus dem Quellcode zeigt die Erzeugung einer `Area` für einen Application-Client-Knoten. Diese hat die Form eines auf der Spitze stehenden Quadrates. Die Positionen im Raster werden in den Variablen `xGridPos` und `yGridPos` gehalten, `gridSize` gibt die Größe eines Rasterfeldes an.

```
/* absolute Koordinaten bestimmen */
int xcoord = xGridPos * gridSize;
int ycoord = yGridPos * gridSize;
int size = gridsize/2;

/* Polygon über seine Ecken definieren */
Polygon polygon = new Polygon();
polygon.addPoint(xcoord + size/2, ycoord);
polygon.addPoint(xcoord + size, ycoord + size/2);
polygon.addPoint(xcoord + size/2, ycoord + size);
polygon.addPoint(xcoord, ycoord + size/2);
Area area = new Area(polygon);
```

Die Größe der Knoten wird in Abhängigkeit zu der eines Rasterfeldes definiert. Dadurch, dass auch ihre Positionen auf der Zeichenfläche relativ zum Raster vermerkt wurden, ist nun ein leichtes Hinein- und Hinauszoomen auf dem Graphen möglich. Es braucht dazu lediglich die Größe eines Rasterfeldes

(`gridSize`) verändert werden und die des Graphen wird automatisch angepasst. Dazu muss obige Erzeugung der `Area` dann natürlich einmal wiederholt werden.

5.2. Beispiel einer kompletten Analyse-Session

Nachdem im vorherigen Abschnitt die Visualisierung des Abhängigkeitsgraphen erläutert wurde, inklusive der dort möglichen Nutzerinteraktion, soll nun aus Sicht des Anwenders geschildert werden, wie die Erzeugung einer neuen JavaEE-Applikation über die graphische Oberfläche erfolgen kann.

1. Auswahl einer EAR-Datei
Als Erstes wird die EAR-Datei ausgewählt, die die zu analysierende Anwendung enthält. Direkt im Anschluss werden die Komponenten der Applikation sowie deren Abhängigkeiten bestimmt und in Form eines Graphen dargestellt.

2. Auswahl der JavaEE-Komponenten
Der Nutzer wählt nun aus, welche der angezeigten Komponenten in die neue Anwendung übernommen werden sollen und welche in dieser auf keinen Fall enthalten sein dürfen. Im Abhängigkeitsgraphen werden während dieser Selektion sofort alle weiteren benötigten Elemente hervorgehoben. Abbildung 5.1 zeigt dies an Hand der EAR-Datei, die in den Beispielen 3 und 4 aus 4.1 verwendet wurde. Sie ist noch relativ klein. In den Beispielen 1 und 2 wurde eine JavaEE-Applikation untersucht, die deutlich umfangreicher ist. Die Darstellung ihrer Abhängigkeitsstruktur ist in Abbildung 5.5 illustriert.

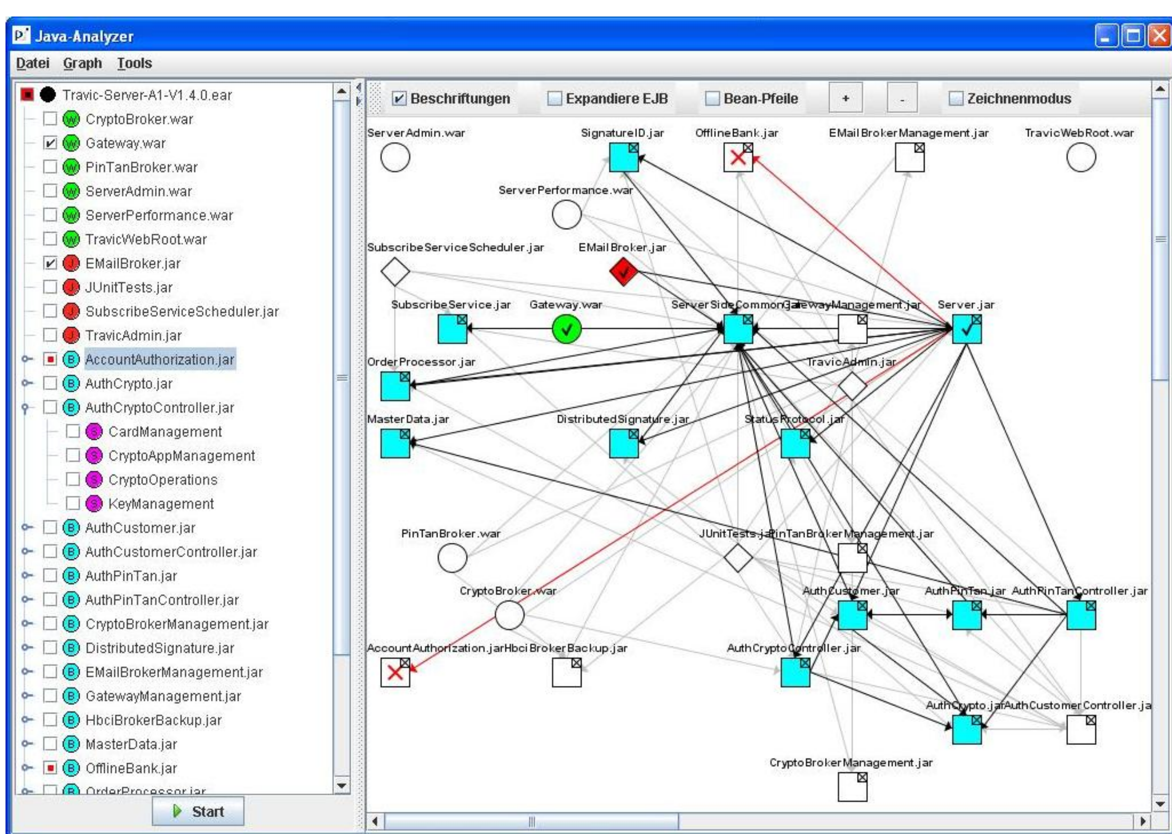


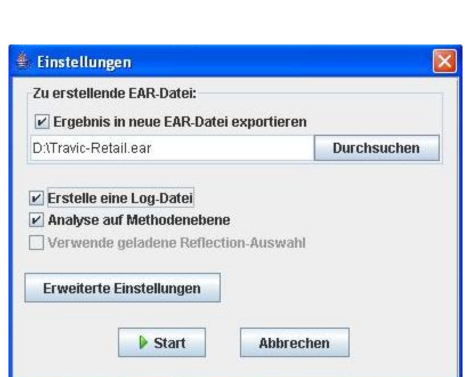
Abbildung 5.5.: Visualisierung der Abhängigkeitsstruktur einer großen Applikation

3. Konfiguration der Analyseparameter
Nach Drücken des Start-Knopfes öffnet sich ein Fenster, in dem die unterschiedlichen Parameter für die Analyse gesetzt werden können (vgl. auch Abschnitt 4.6). Es ist dabei möglich, eine schon früher gespeicherte Konfiguration über das Menü der GUI zu laden. Die dort getätigten Einstellungen werden dann in diesem Fenster angezeigt und können verändert werden.

Diese Parameter umfassen:

- Soll eine neue EAR-Datei generiert werden?
- Name der zu erstellenden EAR-Datei
- Soll Analyseergebnis in XML-Datei ausgegeben werden?
- Analyse auf Klassen- oder Methodenebene?
- Verwendung einer zuvor geladenen Auswahl von Klassen für Reflection-Behandlung
- ignorierbare Pakete
- besondere Behandlung von Klassenbibliotheken Dritter
- speziell zu behandelnde Klassen bei Reflection (→ „Version"-Klassen)

Die zwei Dialoge, über die diese angegeben werden können, sind in Abbildung 5.6 zu sehen.



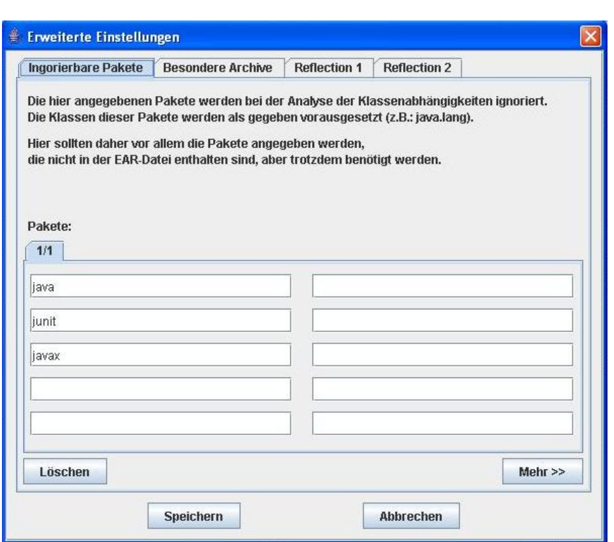


Abbildung 5.6.: Einstellungs-Dialoge

4. Start der Analyse
Wurden die gewünschten Parameter gesetzt, so beginnt die Bestimmung der benötigten Klassen:

- Entpacken der Archive
- Bestimmung von Basisinformationen aller Klassen
- Bestimmung direkter und indirekter Abhängigkeiten
- spezielle Behandlung von Interfaces und Klassenerweiterungen

5. Reflection-Behandlung
Sind die oben angegebenen Schritte der Abhängigkeitsanalyse durchgeführt worden, so erfolgt die Behandlung von Reflection, d.h. der Nutzer muss angeben, welche Klassen über Reflection geladen worden sein könnten. Diese Interaktion entfällt, wenn vor der Analyse eine bereits früher getätigte Auswahl von Klassen geladen worden ist. Es werden dann diese genommen.

Andernfalls erfolgt über ein Dialogfenster die entsprechende Selektion (Abb. 5.7). Zu jeder Klasse, die per Reflection bedient wurde, werden all diejenigen Klassen aufgelistet, die diese implementieren bzw. erweitern. Sie sind dabei nach den Archiven sortiert, in denen sie enthalten sind. Im Beispiel hat das per Reflection bediente Interface `DOMParserFactory` zwei Implementierungen, die beide im Archiv `TransactionDataObjects.jar` liegen.

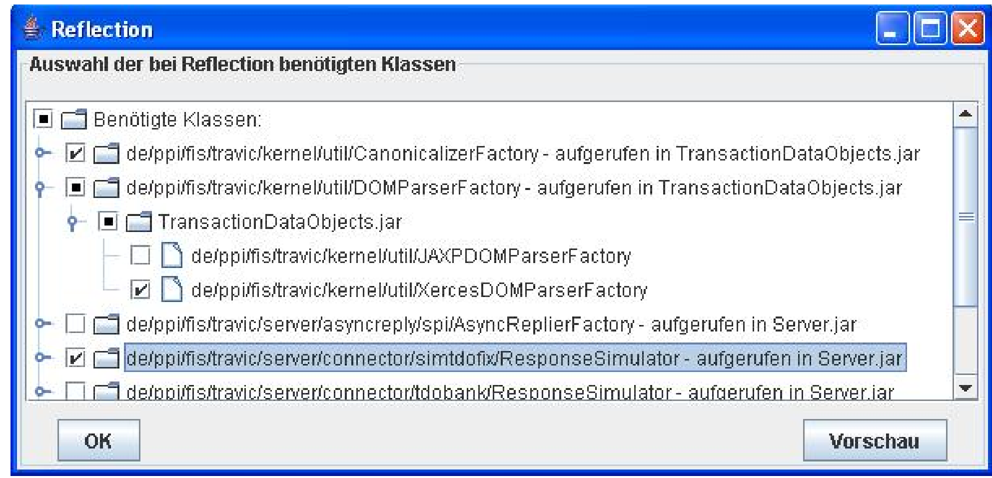


Abbildung 5.7.: Auswahl von Klassen bei Reflection-Behandlung

An dieser Stelle gibt es nun eine besondere Hilfestellung für den Anwender. Es ist möglich, sich eine Vorschau über die Entwicklung der Abhängigkeitsstruktur bei Auswahl bestimmter Klassen erstellen zu lassen (s. Abb. 5.8).

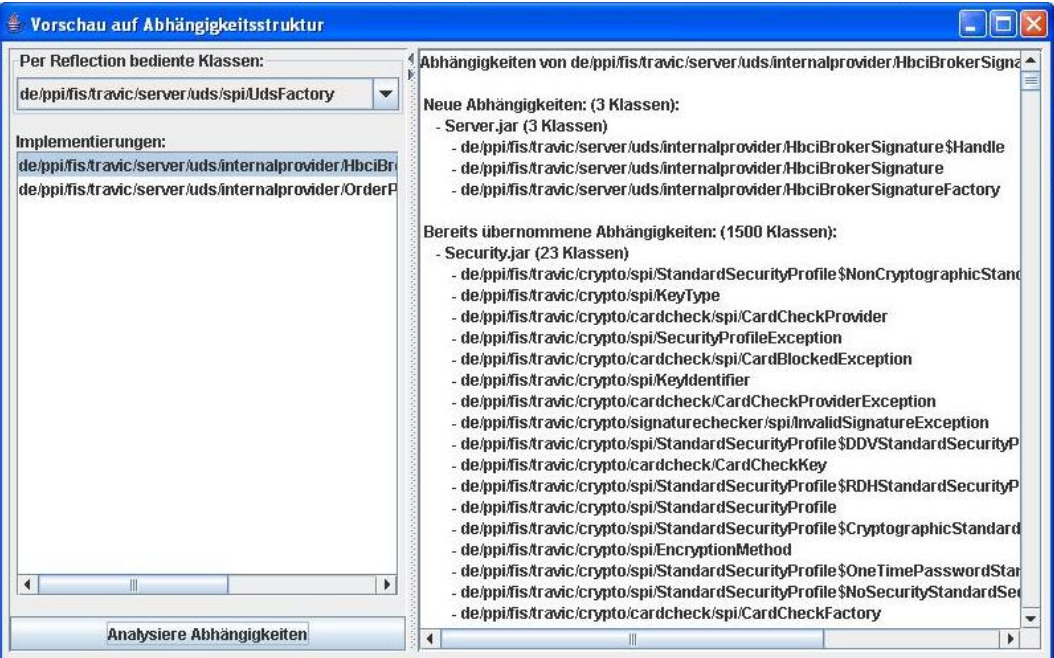


Abbildung 5.8.: Vorschau über Entwicklung der Abhängigkeiten bei Reflection-Behandlung

Angenommen, ein Interface `I` wird per Reflection bedient. Es gibt zwei Implementierungen `A` und `B` für `I`, die dem Nutzer zur Auswahl gestellt werden. Er möchte nun eine davon auswählen (nicht beide, was aber auch möglich wäre), ist aber unschlüssig welche davon.

Um seine Entscheidungsfindung zu erleichtern, kann er sich anzeigen lassen, welche Klassen bei einer Wahl von `A`, welche bei einer Wahl von `B` benötigt werden. Bei einer Analyse auf Methodenebene muss hier natürlich beachtet werden, dass von `A` und `B` genau die Methoden gebraucht werden, die auch von `I` benötigt wurden. Diese Klassen werden des Weiteren danach unterteilt, ob sie ohnehin schon in die neue Applikation übernommen wurden oder ob sie neu hinzukämen. So kann dann zum Beispiel die Alternative gewählt werden, bei der die Größe der Abhängigkeitsmenge am wenigsten wächst.

Es sei jedoch erwähnt, dass in der Regel diese Selektion an Hand technischer Kriterien erfolgt und nicht auf Basis der resultierenden Vergrößerung der neuen Anwendung. Trotzdem liefert diese Vorschau interessante Informationen über die Entwicklung der Abhängigkeitsstruktur in Bezug auf die einzelnen Alternativen für die Auswahl.

Wurden die gewünschten Klassen selektiert, so wird die Abhängigkeitsanalyse fortgesetzt. Unter Umständen ist dort eine erneute Reflection-Behandlung nötig.

6. Abschluss der Analyse

Nach Bestimmung aller benötigten Klassen wird die neue JavaEE-Applikation generiert, d.h. das entsprechende EAR gepackt. Ferner wird das Ergebnis der Analyse in eine XML-Datei ausgegeben, sofern dies vom Nutzer gewünscht wurde.

Die Parameter dieser Analyse-Session können jetzt zur späteren Verwendung gespeichert werden. Dies betrifft nicht nur die aus 3., sondern auch die getroffene Selektion der Komponenten der Applikation. So wird die Durchführung einer Analyse auch ohne Benutzung der GUI ermöglicht, da alle notwendigen Informationen aus der hier erzeugten XML-Datei gewonnen werden können.

6. Service Provider Interfaces

Dieses Kapitel beschreibt, wie das in dieser Arbeit erstellte Analyse-Werkzeug durch geringfügige Modifikationen auch im Rahmen der Behandlung von Service Provider Interfaces eingesetzt werden kann.

6.1. Zielsetzung

6.1.1. Definition und Ziel

Ein *Service Provider Interface (SPI)* ist eine Schnittstelle, über die austauschbare Komponenten in eine Anwendung integriert werden können.

Im Umfeld des PPI Produkts TRAVIC Retail werden solche Interfaces unter anderem dazu verwendet, es dem Kunden zu ermöglichen, eigene Implementierungen in die bestehende Applikation einbinden zu können. Dies erfolgt über das schon ausführlich dargestellte Reflection-Konzept. Dazu wird in einer externen Konfigurationsdatei hinterlegt, welche Klasse das Interface bedienen soll.

Das Ziel ist es nun, die Java-Klassen zu bestimmen, die gebraucht werden, um das SPI implementieren zu können. Sie können dem Kunden dann schon vor der Auslieferung der kompletten Anwendung zur Verfügung gestellt werden, sodass er frühzeitig mit der Implementierung beginnen kann.

Diese Menge von Klassen soll so klein wie möglich sein. Was darunter genau verstanden wird, soll nun erläutert werden.

6.1.2. Anforderungen

Die zu bestimmende Menge soll genau die Klassen enthalten, die zur *Implementierung* des SPIs gebraucht werden. Das heißt, es muss damit nicht zwingend möglich sein, die erstellte Implementierung auch zu testen.

Dies soll an folgendem Beispiel verdeutlicht werden, welches die Deklaration eines Service Provider Interfaces zeigt.

```
public interface MyInterface extends MyOtherInterface{

    public static final MyField f = new MyField();

    public R myMethod (A1 arg1, A2 arg2, A3 arg3) throws E1, E2;

}
```

Zunächst einmal ist zu sagen, dass die Bestimmung der benötigten Klassen im Folgenden stets auf Methodenebene erfolgt. Auf Klassenebene wäre bei der Analyse kein Unterschied zu der des in den vorherigen Kapiteln vorgestellten Tools erkennbar.

Die grundlegende Fragestellung ist nun, welche Methoden aus den Argumenten, der Rückgabe und den geworfenen Exceptions der Interfacemethoden benötigt werden, um das SPI zu implementieren. Sie werden den Ausgangspunkt für die Abhängigkeitsanalyse bilden.

Grundsätzlich können in einer Interface-Implementierung von allen Parametern keine privaten Methoden aufgerufen werden. Dies ist nur in der jeweiligen Klasse selbst möglich. Sie werden also nicht gebraucht, zumindest nicht an dieser Stelle.

Ein Argument muss nicht erzeugt werden können, da bereits eine Instanz beim Aufruf der Methode übergeben wird. Es werden von `A1`, `A2` und `A3` also keine Konstruktoren benötigt. Dies führt zwar dazu, dass die Implementierung an Hand der ermittelten Klassenmenge nicht getestet werden kann, aber dies war auch explizit nicht gefordert. Alle anderen, nicht privaten Methoden könnten während der Implementierung verwendet werden, d.h. die von ihnen benötigten Klassen müssen in der Resultatsmenge enthalten sein.

Ein Rückgabewert muss hingegen instanziiert werden können, genauso wie eine zu werfende Exception. Von `R`, `E1` und `E2` sind also alle nicht privaten Methoden zu berücksichtigen.

Ist nun beispielsweise `R` ein Interface, kann also nicht instanziiert werden, so ist es dem Kunden überlassen, auch eine Implementierung für dieses zu entwickeln. Es bedarf bei der Analyse also keiner besonderen Unterscheidung, ob es sich bei den Parametern um Interfaces oder instanziierbare Klassen handelt.

Weiterhin kann auch auf alle nicht privaten Felder von `R`, `A1`, `A2`, `A3`, `E1` und `E2` zugegriffen werden.

`MyInterface` ist eine Erweiterung von `MyOtherInterface`. Eine Implementierung muss also nicht nur `myMethod` bedienen, sondern auch alle Methoden aus `MyOtherInterface`. Demnach müssen die Abhängigkeiten des erweiterten Interfaces auf die gleiche Art und Weise bestimmt werden.

Zusammenfassend werden also gebraucht:

- keine privaten Methoden
- alle anderen Methoden von `R`
- alle anderen Methoden von `A1`, `A2`, `A3`, außer Konstruktoren
- alle anderen Methoden von `E1`, `E2`
- alle nicht privaten Felder von `R`, `A1`, `A2`, `A3`, `E1` und `E2`
- das erweiterte Interface `MyOtherInterface`

Es fehlen jetzt noch die Klassenfelder von `MyInterface`.

Nach [18] sind Felder in einem Interface grundsätzlich öffentliche, statische Konstanten, die hier auch direkt instanziiert werden. Damit eine Implementierung dieses Interfaces kompiliert werden kann, wird die entsprechende Klasse, im Beispiel `MyField`, ebenso benötigt, wie die im aufgerufenen Konstruktor verwendeten Klassen. Sie müssen also ebenfalls in das zu ermittelnde Resultat übernommen werden.

Wie schon in Abschnitt 4.2.5 erläutert worden ist, wird beim Kompilieren des Quellcodes eine spezielle Methode `<clinit>` erzeugt, in der die Initialisierung von Klassenfeldern durchgeführt wird. Hier finden sich also genau die von den Feldern benötigten Klassen und Methoden. Somit genügt es, die Abhängigkeiten von `<clinit>` auf die gleiche Weise zu ermitteln, wie im Algorithmus aus Kapitel 4.

Natürlich reicht es nicht aus, nur direkte Abhängigkeiten zu bestimmen. Auch hier müssen rekursiv die indirekten ermittelt werden. Dabei bedarf es erneut einer speziellen Behandlung.

Sei die Klasse `A1` aus obigem Beispiel wie folgt definiert:

```
public class A1 extends A {

    public Z zField;

    public Y method (X arg) throws MyException {
        MyClass.method1();
        arg.method2();
        return new Y();
```

```
    }
}
```

A1 erweitert eine Klasse A. Die Methoden von A können bei der Implementierung von MyInterface genauso aufgerufen werden wie die von A1. Sie gehören dann natürlich auch in die Abhängigkeitsmenge. Ausnahmen bilden hier wieder Konstruktoren und private Methoden.

Wie nun A1 in einer Implementierung von MyInterface verwendet werden kann, soll an diesem Beispiel gezeigt werden:

```
public class MyImpl implements MyInterface {

    public R myMethod (A1 arg1, A2 arg2, A3 arg3) throws E1, E2 {
        try {

            X x = new X();
            x.callAnyMethod();

            Y y = arg1.method(x); // Methodenaufruf von A

            [...]

        } catch (MyException myEx) {
            myEx.callAnyMethod();
        }

        return new R();
    }

    [...]
}
```

Soll jetzt A1.method in der SPI-Implementierung aufgerufen werden, so muss dafür zunächst ein Argument vom Typ X erzeugt worden sein. Es werden also X's Konstruktoren benötigt. Ebenso können alle anderen, nicht privaten Methoden von X aufgerufen werden.

Vom Rückgabewert Y und auch von MyException werden hingegen nicht alle Konstruktoren gebraucht. Ihre Erzeugung erfolgt innerhalb des Methodenrumpfs von method und muss daher nicht vom Kunden vorgenommen werden. Die übrigen Methoden jedoch können durchaus gebraucht werden, beispielsweise bei der Behandlung einer gefangenen MyException.

Dasselbe gilt wiederum für Superklassen von Y, X und MyException.

Die Bestimmung der im Rumpf von method auftretenden Abhängigkeiten kann auf dieselbe Weise durchgeführt werden, wie bei der in Kapitel 4 vorgestellten Analyse.

Noch einmal zusammengefasst: Von A's Methode method werden gebraucht:

- alle nicht privaten Methoden von X
- alle nicht privaten Methoden von Y, außer Konstruktoren
- alle nicht privaten Methoden von MyException, außer Konstruktoren
- im Methodenrumpf verwendete Klassen und Methoden

Da auf die nicht privaten Felder von A1, in diesem Fall also Z, auch von außen zugegriffen werden kann, werden von diesen auch alle nicht privaten Methoden benötigt, mit Ausnahme der Konstruktoren.

Die Abhängigkeitsanalyse der Klassen X, Y, Z und MyException erfolgt nun genauso wie die von A1.

Der entscheidende Unterschied zur Analyse aus Kapitel 4 besteht nun gerade in der Behandlung von Klassenfeldern und Methodenköpfen. Die Analyse des Rumpfes erfolgt auf die gleiche Weise. Damit ist auch klar, wo der bisherige Algorithmus für die SPI-Behandlung angepasst werden muss.

6.2. Implementierung

In diesem Abschnitt sollen nun die Unterschiede der Implementierung dieser Analyse zu der aus Kapitel 4 näher erläutert werden.

6.2.1. Eingabe und Vorbereitungen

Die Eingabe für den Analysealgorithmus umfasst zum Einen eine Menge von Service Provider Interfaces. Zum Anderen muss auch ein Suchraum für benötigte Klassen angegeben werden, der aber nicht mehr nur aus einer EAR-Datei bestehen muss, wie es vorher der Fall war. Er kann sich aus beliebigen Archiven, aber auch aus Verzeichnissen zusammensetzen.

Für die Analyse ist es weder von Belangen, ob es sich bei der analysierten Anwendung um eine JavaEE-Applikation handelt, noch aus welchen Archiven die jeweiligen Klassen stammen. D.h., im Gegensatz zum Algorithmus aus den vorherigen Kapiteln kann sich hier auf die Bestimmung der Abhängigkeiten zwischen Java-Klassen beschränkt werden.

Sind im angegebenen Suchraum Archive enthalten, so werden diese wieder in ein temporäres Verzeichnis entpackt.

6.2.2. Bestimmung der direkten Abhängigkeiten

Direkte Abhängigkeiten einer Klasse werden wie bisher durch das Parsen der entsprechenden `class`-Datei bestimmt. Es ist jetzt aber zusätzlich eine Sonderbehandlung bei den Methodenköpfen einiger Klassen erforderlich. Von den dort verwendeten Klassen werden alle Methoden gebraucht, außer private und eventuell Konstruktoren. Dies muss beim Parsen speziell vermerkt werden.

Wie in Abschnitt 4.3.1 vorgestellt wurde, werden die gefundenen Abhängigkeiten in einer Datenstruktur `SimpleJavaClass` gespeichert, die Referenzen über die benötigten Methoden enthält. Über ein Flag lässt sich nun angeben, dass alle nicht-privaten Methoden der Klasse gebraucht werden. Ein weiteres Flag dient dazu, Konstruktoren von dieser Menge auszuschließen.

Werden beim Parsen einer Methode eines Service Provider Interfaces abhängige Klassen ermittelt, so werden deren benötigte Bestandteile durch Setzen dieser Flags gekennzeichnet. Diese Abhängigkeiten stammen im Wesentlichen aus dem Methodendeskriptor. Die geworfenen Exceptions finden sich im Exceptions-Attribut (vgl. Abschnitt 4.2.5) der Methode .

Sind parametrisierte Typen im Methodenkopf enthalten, so muss zusätzlich das Signatur-Attribut hinzugezogen werden, da auch von den Typparametern alle erwähnten Methoden gebraucht werden. Soll zum Beispiel eine Liste vom Typ `R`, also `List<R>`, zurückgegeben werden, so müssen auch Objekte vom Typ `R` erzeugt werden können.

Die einzige Methode eines Interfaces, die auch einen Rumpf haben kann, ist `<clinit>`. Sie wird aber wie bisher als globale Klassenabhängigkeit betrachtet und bedarf somit keiner gesonderten Behandlung.

Bei den Feldern einer Klasse wird analog zu den Methoden verfahren.

Auf dieselbe Art und Weise werden Abhängigkeiten von Interfaces bestimmt, die dieses SPI erweitert.

Beim Parsen selbst müssen nun drei Fälle unterschieden werden, wie die gefundenen Abhängigkeiten gespeichert werden sollen.

1. Bei der Behandlung eines SPIs brauchen von den Argumentparametern seiner Methoden keine Konstruktoren übernommen werden.
2. Bei der rekursiven Bestimmung der Abhängigkeiten der Parameter der SPI-Methoden hingegen kann bei Rückgabewerten und Exceptions auf Konstruktoren verzichtet werden.
3. Der dritte Fall ist gerade der, dass die Analyse wie bisher, also ohne jegliche Sonderbehandlung von Methodenköpfen, erfolgen soll.

Die Unterscheidung danach, in welchem Modus das Parsen durchzuführen ist, erfolgt wieder durch Setzen gewisser Flags.

6.2.3. Rekursive Bestimmung indirekter Abhängigkeiten

Es wurde nun beschrieben, wie direkte Abhängigkeiten in den verschiedenen Modi bestimmt werden. Nun ist noch zu klären, wie bei der rekursiven Analyse festgestellt werden kann, in welchem Modus der Parser aufgerufen werden soll.

Der einfache Fall ist der zur Bestimmung direkter Abhängigkeiten eines SPIs, da dies zu Beginn des Algorithmus für alle Service Provider Interfaces und deren Superklassen erfolgt, noch ehe die rekursive Analyse für eines von ihnen begonnen wird.

Die Unterscheidung der beiden anderen Fälle ist etwas schwieriger. Wurde festgestellt, dass von einer Klasse `C` gemäß Fall 2 alle Methoden gebraucht werden, so ist im betreffenden `SimpleJavaClass`-Objekt das Flag `takeAllMethods` gesetzt. Ob Konstruktoren davon ausgeschlossen werden sollen, ist für die Fallunterscheidung unerheblich.

Bei der Bestimmung der direkten Abhängigkeiten von `C` ist nun bekannt, dass `takeAllMethods` den Wert `true` hat. An Hand dessen ist klar, dass von den Parametern aller nicht-öffentlichen Methoden von `C` ebenfalls alle Methoden benötigt werden (außer den privaten und evtl. Konstruktoren). Gleiches gilt für Klassenfelder. In den für diese Klassen erzeugten `SimpleJavaClass`-Objekten wird das Flag `takeAllMethods` daher ebenfalls gesetzt.

Ist das angesprochene Flag nicht gesetzt, so wird die Analyse ohne jegliche Sonderbehandlung durchgeführt.

6.2.4. Abschluss der Analyse

Eine Spezialbehandlung, wie sie in Kapitel 4 zum Ende des Algorithmus beispielsweise für Reflection vollzogen wurde, ist hier nicht nötig, da hier keine lauffähige JavaEE-Anwendung erzeugt werden soll, sondern lediglich eine Sammlung von Klassen, mit denen ein SPI implementiert werden kann.

Was aber noch durchzuführen ist, ist eine Behandlung von Interfaces und Klassenerweiterungen analog zu Abschnitt 4.3.7.

Sind alle benötigten Java-Klassen ermittelt, so werden diese zusammen mit den Service Provider Interfaces aus der Eingabe in eine JAR-Datei gepackt, die nun dem Kunden zur Verfügung gestellt werden kann.

6.3. Eine graphische Oberfläche

Abbildung 6.1 zeigt eine graphische Oberfläche für das hier vorgestellte Werkzeug.

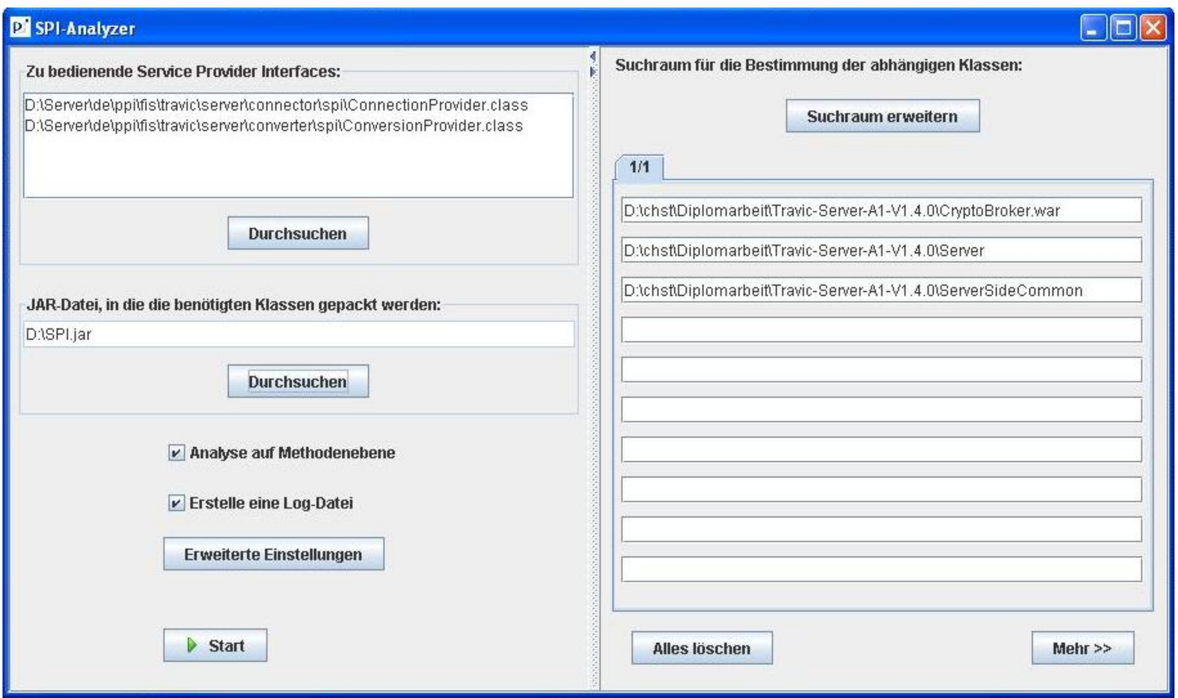


Abbildung 6.1.: GUI für SPI-Behandlung

Der Nutzer kann über diese folgende Parameter für die Analyse angeben:

- zu analysierende Service Provider Interfaces
- JAR-Datei, in die die Ergebnismenge gepackt werden soll
- Suchraum der Analyse (Archive und / oder Verzeichnisse)
- Analyse auf Methoden- oder Klassenebene?
- Speichern des Analyseergebnisses in einer XML-Datei
- Angabe von Paketen, die bei der Analyse ignoriert werden können

Nach Start der Analyse ist im Gegensatz zur Erstellung einer JavaEE-Applikation hier keine weitere Interaktion mit dem Anwender erforderlich. Die benötigten Klassen werden bestimmt, das gewünschte Java Archiv erzeugt. Zusätzlich lässt sich das Ergebnis auch in Form einer XML-Datei ausgeben, die Informationen darüber liefert, welche Klassen in der JAR-Datei enthalten sind und welche nicht im Suchraum gefunden werden konnten.

7. Signatur-Matching

In diesem Kapitel wird die Implementierung eines Signatur-Matchings für Java vorgestellt. Sie basiert auf der Arbeit *Signature Matching, a Tool for Using Software Libraries* von Amy Moormann Zaremski und Jeannette M. Wing [1], in der die theoretischen Grundlagen hierzu definiert werden.

7.1. Ziel

Die Wiederverwendbarkeit von bestehenden Programmkomponenten ist ein erstrebenswertes Ziel in der Softwareentwicklung. Bei steigendem Volumen wird es aber oft schwierig, einen genauen Überblick zu behalten. Hier soll diese Arbeit in Bezug auf Java-Programme Abhilfe leisten.

In diesem Kapitel wird daher untersucht, ob in einer Menge von Java-Klassen solche enthalten sind, die ein gegebenes Interface implementieren *könnten*. Die Bestimmung erfolgt dabei durch einen Vergleich der Signaturen der betreffenden Klassen.

Definition: Signatur
Die Signatur einer Methode definiert ihre formale Schnittstelle, d.h. sie besteht im Wesentlichen aus den Argument- und Rückgabewerten.

In Java setzt sie sich zusammen aus:

- Name der Methode
- Anzahl der Argumente
- Reihenfolge der Argumente
- Typen der Argumente
- Typ des Rückgabewertes
- Anzahl der geworfenen Exceptions
- Typen der geworfenen Exceptions

Die Namen der Argumente sind hier unerheblich. Ebenso wenig werden die Access-Flags der Methoden wie `public`, `private` oder `static` an dieser Stelle berücksichtigt. Später wird auf sie aber noch einmal zurückgekommen werden.

Der Vergleich der Signaturen zweier Methoden erfolgt an Hand der oben aufgelisteten Kriterien. Dies wird nach [1] als *Function Matching* bezeichnet. In Abschnitt 7.2 werden unterschiedliche Klassifizierungen bezüglich der Genauigkeit dieses Matchings vorgestellt, sodass zwei Methodensignaturen nicht nur dann erfolgreich matchen, wenn sie absolut identisch sind.

Es soll nun eine Java-Klasse `C` dahingehend untersucht werden, ob sie ein bestimmtes Interface implementieren könnte. Dazu muss für jede Methode `m` des Interfaces überprüft werden, ob `C` über eine Methode verfügt, deren Signatur gegen die von `m` matcht. Es werden also die Signaturen der Klassen verglichen. Diese umfassen grundsätzlich auch die der jeweiligen Superklasse.

Nach [1] handelt es sich dabei um ein *Module Matching*. Auch hier können prinzipiell mehrere Matching-Arten unterschieden werden. In Hinsicht auf die Zielsetzung, mögliche Interface-Implementierungen zu

finden, wird hier nur eine der in [1] vorgestellten Varianten umgesetzt: das *Specialized Match*. Nach diesem matcht ein Modul M_2 ein Modul M_1, wenn M_2 mindestens die Methoden umfasst (in Bezug auf deren Signaturen), die auch M_1 enthält. Dies entspricht gerade der Voraussetzung dafür, dass M_2 M_1 implementieren könnte.

Die erfolgreich gematchten Klassen sind nun potentielle Kandidaten dafür, das gegebene Interface implementieren zu können. Es werden jedoch keinerlei Hinweise darüber geliefert, ob dies in allen Fällen aus technischer Sicht auch Sinn macht, d.h. ob die Klasse die vom Interface erwünschte Funktionalität erfüllt.
Das Signatur-Matching soll dem Entwickler vielmehr einen Überblick darüber verschaffen, welche Klassen bereits existieren, die, gegebenenfalls durch einige Modifikationen, als Implementierung für das Interface in Frage kommen.

7.2. Matching-Arten

In [1] werden unterschiedliche Arten von Signatur-Matchings für Funktionen eingeführt, welche in der in dieser Arbeit erstellten Implementierung umgesetzt wurden. Sie unterscheiden sich in der Genauigkeit des Vergleichs und sollen nun in Bezug auf Java-Programme vorgestellt werden. Hierbei werden die originalen, englischen Bezeichnungen aus [1] verwendet.
Zuvor wird jedoch definiert, wann in Java eine Klasse ein Ober- bzw. ein Untertyp einer anderen ist.

7.2.1. Definition: Subtyp und Supertyp

In der objektorientierten Programmierung ist ein Typ σ ein *Unter-* oder *Subtyp* eines Typs τ ($\sigma < \tau$), wenn in jedem Kontext, der einen Wert des Typs τ erwartet, ein Wert des Typs σ verwendet werden kann.
Ein Typ σ ist ein *Ober-* oder *Supertyp* eines Typs τ ($\sigma > \tau$), wenn τ ein Subtyp von σ ist.

Angewendet auf Java heißt dies nach [18]:
Eine Klasse U ist ein direkter Subtyp einer Klasse A, wenn einer der folgenden vier Fälle gilt:

- U erweitert A
  Bsp: `java.lang.String` ist ein Subtyp von `java.lang.Object`
- U implementiert das Interface A
  Bsp: `java.util.LinkedList` ist ein Untertyp des Interfaces `java.util.List`
- U ist ein Array vom Typ U_1, A ist ein Array vom Typ A_1 und $U_1 < A_1$
  Bsp. `String[]` ist ein Untertyp von `Object[]`
- U ist ein parametrisierter Typ und es gilt:
  - $U := U_1 < T_1, ..., T_n >$
  - $A := A_1 < S_1, ..., S_m >$
  - $U1 \leq A1$
  - A hat entweder keine Typparameter ($m = 0$) oder dieselben wie U ($T_i = S_i$, $i = 1, ..., n$, $n = m$)

  Bsp.: `java.util.List<String>` ist Untertyp von `java.util.List`
  Bsp.: `java.util.LinkedList<String>` ist Untertyp von `java.util.List<String>`
  Bsp.: `java.util.List<String>` ist **kein** Untertyp von `java.util.List<Object>`

Für primitive Datentypen gilt ferner:

- `float < double`
- `long < float`
- `int < long`
- `char < int`
- `short < int`
- `byte < short`

Außerdem gilt:

- `Object[] < java.lang.Object`
- `Object[] < java.lang.Cloneable`
- `Object[] < java.io.Serializable`
- Ist `p` ein primitiver Datentyp, so gilt:
  - `p[] < java.lang.Object`
  - `p[] < java.lang.Cloneable`
  - `p[] < java.io.Serializable`

7.2.2. Exact Match

Es seien nun zwei Methoden M_1 und M_2 wie folgt definiert:

M_1:

$R_1\ methodName1\ (A_1\ arg_1,\ ...,\ A_n\ arg_n)\ throws\ E_1,\ ...\ E_r$

M_2:

$R_2\ methodName2\ (B_1\ arg_1,\ ...,\ B_m\ arg_m)\ throws\ F_1,\ ...\ F_s$

Die verschiedenen Matching-Arten sollen an Hand dieser beiden Methodendeklarationen in Bezug auf Java vorgestellt werden. M_1 sei dabei die Methode, zu der ein passendes Matching gefunden werden soll. Es wird nun definiert, wann M_2 dies in den einzelnen Varianten erfüllt.

M_2 matcht M_1 *exact*, wenn gilt:

- $methodName1 = methodName2$
- $R_1 = R_2$
- $n = m,\ A_i = B_i,\ i = 1, ..., n$
- $r = s$, es existiert Permutation P von $(F_1, ..., F_s)$ so, dass $P_j = E_j,\ j = 1, ..., r$

Die Signaturen der Methoden müssen also identisch sein. Lediglich die Reihenfolge, in der die Exceptions deklariert wurden, ist beliebig.

Beispiel:

M_1

```
int myMethod (int i, List l) throws IOException, ClassNotFoundException
```

M_2

```
int myMethod (int i, List l) throws ClassNotFoundException, IOException
```

7.2.3. Reorder Match

M_2 matcht M_1 *reordered*, wenn gilt:

- $methodName1 = methodName2$
- $R_1 = R_2$
- $n = m$, es existiert Permutation P von $(B_1, ..., B_m)$ so, dass $P_i = A_i,\ i = 1, ..., n$
- $r = s$, es existiert Permutation P' von $(F_1, ..., F_s)$ so, dass $P'_j = E_j,\ j = 1, ..., r$

Beim Matching ist es also unerheblich, in welcher Reihenfolge die Argumente auftreten. Es wird lediglich gefordert, dass diese so umgeordnet werden können, dass beide Methoden exakt matchen.

Beispiel:
M_1

```
int myMethod (int i, List l) throws IOException, ClassNotFoundException
```

M_2

```
int myMethod (List l, int i) throws IOException, ClassNotFoundException
```

7.2.4. Generalized Match

M_2 matcht M_1 *generalized*, wenn gilt:

- $methodName1 = methodName2$
- $R_1 \leq R_2$
- $n = m,\ A_i \leq B_i,\ i = 1, ..., n$
- $\forall$ Exceptions e_1 von M_1: $\exists$ Exception e_2 von M_2: $e_1 \leq e_2$
- $\forall$ Exceptions e_2 von M_2: $\exists$ Exception e_1 von M_1: $e_1 \leq e_2$

In diesem Fall matchen die beiden Methoden auch dann, wenn die in M_2 verwendeten Typen Supertypen der von M_1 sind. Die Signatur der Methode M_2 ist also allgemeiner als die von M_1.

Bei den Exceptions wird gefordert, dass zu jeder in M_1 geworfenen eine in M_2 existiert, die ein Supertyp von ihr ist. Es dürfen in M_2 aber keine weiteren Exceptions geworfen werden, d.h. es darf keine geben, zu der kein Untertyp in M_1 existiert. Ein paarweises Matching ist hierbei jedoch nicht erforderlich. Die Anzahl an Exceptions in M_2 kann somit ungleich der von M_1 sein, wie auch das folgende Beispiel zeigt.

M_1

```
int myMethod (int i, List l) throws IOException, ClassNotFoundException
```

M_2

```
long myMethod (int i, Collection c) throws Exception
```

Im Beispiel ist erkennbar, dass der Rückgabewert `long` von M_2 ein Supertyp von dem von M_1 ist. Auch bei den Argumenten enthält M_2 an Stelle des Interfaces `java.util.List` dessen direkte Superklasse `java.util.Collection`. M_2 umfasst nur eine Exception: `java.lang.Exception`, die aber eine Superklasse von beiden in M_1 verwendeten ist.

7.2.5. Specialized Match

M_2 matcht M_1 *specialized*, wenn gilt:

- $methodName1 = methodName2$
- $R_1 \geq R_2$
- $n = m,\ A_i \geq B_i,\ i = 1, ..., n$
- $\forall$ Exceptions e_1 von M_1: $\exists$ Exception e_2 von M_2: $e_1 \geq e_2$
- $\forall$ Exceptions e_2 von M_2: $\exists$ Exception e_1 von M_1: $e_1 \geq e_2$

Dies ist der entgegengesetzte Fall zum Generalized Match, d.h. in M_2 werden Subtypen von M_1 verwendet. M_2 ist spezieller als M_1.

Beispiel:
M_1

```
int myMethod (int i, List l) throws IOException, ClassNotFoundException
```

M_2

```
short myMethod (int i, LinkedList l) throws FileNotFoundException, ZipException,
    ClassNotFoundException
```

Bei der Rückgabe wurde statt `int` in M_2 dessen Subtyp `short` verwendet, das Interface `java.util.List` wurde durch eine seiner Implementierungen, `java.util.LinkedList`, ersetzt.
Die geworfene `java.io.IOException` ist in M_2 durch zwei ihrer Unterklassen ausgetauscht worden: die `java.io.FileNotFoundException` und die `java.util.zip.ZipException`.

7.2.6. Rename Match

Eine letzte Variante, die nicht in [1] vorgestellt wurde, hier aber trotzdem behandelt werden soll, betrifft den Fall, dass sich zwei Methoden nur durch den Namen unterscheiden. Diese Art des Matchings wird als *Rename Match* bezeichnet.

Die hier vorgestellten Matching-Arten können natürlich untereinander beliebig kombiniert werden.

7.3. Implementierung

In diesem Abschnitt wird nun vorgestellt, wie durch eine Implementierung des erläuterten Signatur-Matchings herausgefunden werden kann, welche möglichen Implementierungen für ein Interface vorhanden sind.
Abstrakte Klassen werden von der Suche ausgeschlossen, da sie bis auf geringfügige Einschränkungen nahezu jedes Interface implementieren können.

7.3.1. Eingabe

Die Eingabe für die Analyse bildet zum Einen eine Menge von Interfaces, gegen die gematcht werden soll. Zum Anderen wird hier aber auch der Suchraum spezifiziert, aus dem die zu untersuchenden Klassen bezogen werden. Diese liegen wiederum in Form von `class`-Dateien vor.

Abbildung 7.1 zeigt die graphische Oberfläche zur Ansteuerung des Signatur-Matchings.
Hier ist auch eine Auflistung von vier der fünf beschriebenen Matching-Arten zu finden. Ihre Reihenfolge gibt Aufschluss darüber, welche Matching-Art beim Signaturenvergleich zuerst betrachtet werden soll,

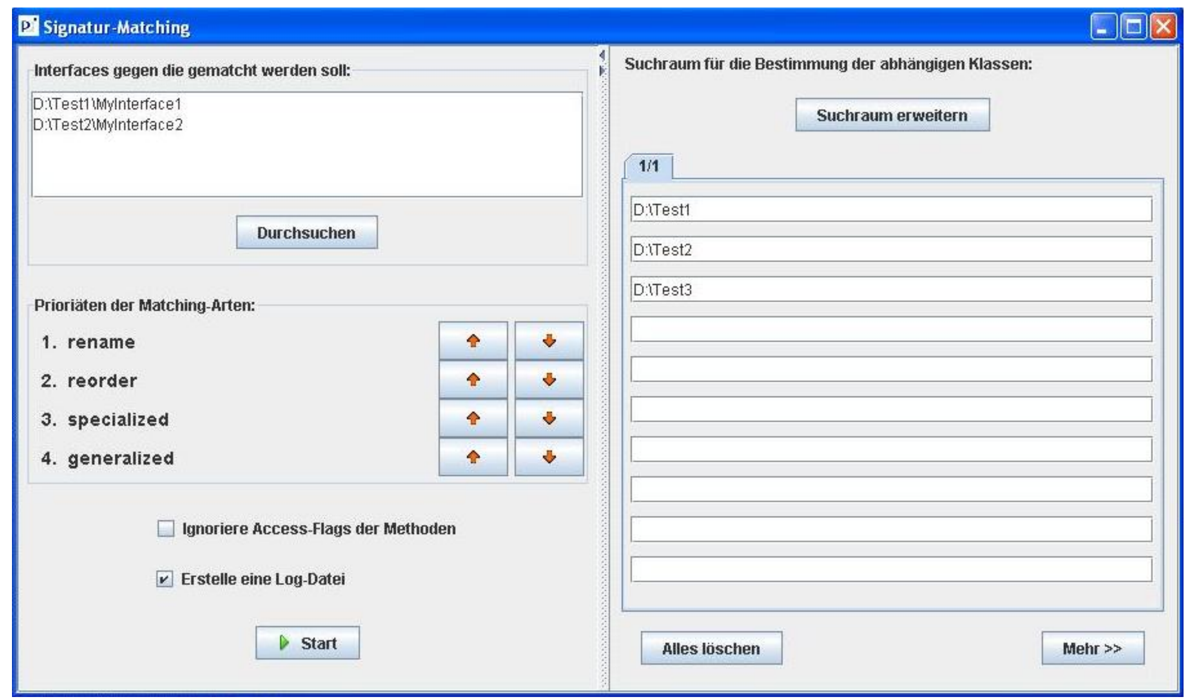


Abbildung 7.1.: Eingabe fürs Signatur-Matching

wenn ein exaktes Matching nicht möglich war. Da Letzteres grundsätzlich zuerst überprüft wird, fehlt es in der Auflistung, die der Nutzer des Werkzeugs über die Pfeil-Buttons umordnen kann. Es ist also möglich, Präferenzen bezüglich der verschiedenen Matching-Varianten anzugeben.

Die Reihenfolge, in der gemäß Abbildung 7.1 versucht werden würde, ein passendes Matching zu finden, wäre also:

- Exact Match
- Rename Match
- Reorder Match
- Rename + Reorder Match
- Specialized Match
- Rename + Specialized Match
- ...

Es werden demnach zuerst Kombinationen der Matching-Arten hoher Priorität untersucht, bevor die von niedereren einzeln betrachtet werden.

Ferner ist über eine Check-Box einstellbar, ob die Access-Flags der Methoden ignoriert werden sollen. Ist dies nicht der Fall, so werden beim Matching nur Methoden berücksichtigt, die als `public` und nicht als `static` deklariert wurden.

Dies begründet sich darin, dass Interfacemethoden stets implizit `public` sind und nicht `static` sein dürfen (vgl. [18]). Da nur gegen Interfaces gematcht werden soll, reicht es aus, auch nur solche Methoden in den untersuchten Klassen zu betrachten. Wird hingegen vom Nutzer in Betracht gezogen, die Access-Flags gegebenenfalls zu ändern, so können diese bei der Analyse ignoriert werden.

7.3.2. Bestimmung von Supertypen

Die Basis für Generalized und Specialized Match stellen die Supertyp-Beziehungen zwischen den betrachteten Java-Klassen dar. Der erste Schritt ist es also, diese zu bestimmen und global abzuspeichern, wobei es ausreicht sich auf direkte Beziehungen zu beschränken. Indirekte können durch Iteration über diese transitive Relation erhalten werden.

Die dafür notwendigen Informationen können durch Parsen der zu Grunde liegenden class-Dateien ermittelt werden. Entscheidend dafür sind die Deklarationen der implementierten Interfaces und natürlich der Superklasse. Lediglich die Supertypen von primitiven Datentypen und Arrays sind nicht so bestimmbar und müssen daher gemäß Abschnitt 7.2.1 direkt angegeben werden.

Da diese Beziehungen für alle Typen ermittelt werden müssen, die in den Klassen aus dem Suchraum verwendet werden, werden auch schon zu Beginn alle Klassen geparst. Neben den Supertypen werden hierbei ebenfalls die Signaturen der Methoden bestimmt und gespeichert. Hierauf wird im folgenden Abschnitt 7.3.3 genauer eingegangen.

Die beim Parsen ermittelten Supertypbeziehungen decken jedoch nicht alle Klassen ab. So fehlen hier diejenigen, die von der Java-Laufzeitumgebung bereitgestellt werden. Es ist so zum Beispiel noch nicht bekannt, dass java.util.List eine Erweiterung von java.util.Collection ist.

Diese Informationen werden jetzt analog zu Abschnitt 4.3.7 über Reflection bestimmt. Kann zu einer Klasse dennoch die direkte Superklasse nicht identifiziert werden, so wird hier java.lang.Object angenommen, also die Klasse, von der jede erbt.

7.3.3. Bestimmung der Signaturen

Zu jeder Klasse aus dem Suchraum werden nun die Signaturen ihrer Methoden bestimmt. Nicht benötigt werden dabei Konstruktoren und je nach Nutzereingabe auch Methoden, die static oder nicht public sind.

Beim Parsen der class-Datei werden die Methodendeskriptoren ausgelesen, die den wesentlichen Teil der Signatur definieren. Treten hier parametrisierte Typen auf, so muss zusätzlich das Attribut *Signature* der Methodendeklaration betrachtet werden. Die geworfenen Exceptions finden sich im *Exceptions*-Attribut wieder. Dies wurde bereits in Abschnitt 4.2.5 erläutert.

Die dort gewonnen Informationen werden in folgenden Feldern der Datenstruktur MethodSignature gespeichert:

- String methodName: Name der Methode
- Type returnType: Typ der Rückgabe
- List<Type> argumentTypes: geordnete Liste von Argumenttypen
- Set<Type> exceptionTypes: ungeordnete Menge von Exception-Typen

Zur Repräsentation eines verwendeten Typs wurde eine Datenstruktur Type erstellt, die vor allem zur Speicherung parametrisierter Typen wichtig ist.

Sie enthält eventuelle Typparameter als geordnete Liste des Typs Type.

Auf die gleiche Weise werden Parameter eines Arrays gespeichert, welches hier als eigenständiger Typ angesehen wird.

Folgendes Beispiel zeigt die Repräsentation eines Arrays von HashMaps:

```
HashMap<Integer, List<String>>[]

- Typ: "["
```

```
Parameter:
  - Typ: "HashMap"
    Parameter:
      - Typ: "Integer"
      - Typ: "List"
        Parameter:
          - Typ: "String"
```

Handelt es sich bei der Klasse, deren Signatur gerade bestimmt wird, um ein Interface, zu dem mögliche Implementierungen gesucht werden sollen, so werden die einzelnen `Type`-Objekte aus den Methoden-Signaturen um weitere Informationen angereichert. Diese sollen ein schnelleres Matching ermöglichen. Es werden dazu sämtliche Sub- und Supertypen des jeweiligen Typs vermerkt, sodass bei einem Specialized oder Generalized Match sofort die notwendigen Informationen verfügbar sind.

Die erzeugten `MethodSignature`-Objekte werden dann in einer Datenstruktur `ClassSignature` zusammengefasst, die weiterhin Informationen über die zugehörige Klasse wie deren direkte Superklasse oder implementierte Interfaces bereitstellt.
Die Signatur der Klasse ist damit jedoch noch nicht komplett. Sie müssen noch um die ihrer Superklassen erweitert werden. Dazu werden die jeweiligen `MethodSignature`-Objekte aus den Superklassen übernommen. Dies gilt aber nur für Methoden, die nicht überschrieben werden. Wie dies überprüft werden kann, wurde bereits in Abschnitt 4.3.7 erläutert.

Sind die Signaturen für alle Klassen und Methoden wie hier beschrieben bestimmt worden, so kann das Matching gestartet werden.

7.3.4. Durchführen des Matchings

Sei im Folgenden `I` das Interface, gegen das gematcht werden soll. Es wird nun über alle Klassen `C` aus dem Suchraum iteriert, die nicht abstrakt sind.

Der folgende, vereinfachte Auszug aus dem Quellcode zeigt den Algorithmus, der das Matching von zwei Klassen-Signaturen berechnet:

```
boolean matchClassSignature (ClassSignature I, ClassSignature C) {
    for (MethodSignature interfaceMethod : I.getMethods()) {
        boolean matchingFound = false;

        /* unterschiedliche Matching-Arten in bestimmter Reihenfolge überprüfen */
        for(MatchType matchType : matchTypes) {

            for (MethodSignature classMethod : C.getMethods()) {
                /* Wurde diese Methode schon verarbeitet? */
                if (classMethod.isMatched())
                    continue;
                /* matchen beide Signaturen? */
                if (matchMethodSignature(interfaceMethod,
                                         classMethod, matchType)) {
                    classMethod.setIsMatched;
                    matchingFound = true;
                    break;
                }
            }

        }
```

```
            /* Wurde kein Matching gefunden? */
            if(matchingFound == false)
                return false;
        }

        return true;
    }
```

Wichtig hierbei ist, dass eine Methode von `C` nicht gegen mehr als eine Methode von `I` gematcht wird. Denn es muss sichergestellt werden, dass zu jeder Interfacemethode eine exklusive Implementierung vorhanden ist.

Das Methoden-Matching erfolgt nun durch die Funktion `matchMethodSignature`. Ihr werden als zusätzliche Argumente die zugelassen Matching-Arten übergeben. Dies wurde in obigem Code aus Übersichtlichkeitsgründen etwas vereinfacht dargestellt.

Das folgende Beispiel zeigt nun einen Ausschnitt dieser Methode, der sich auf den Vergleich der Argumenttypen ohne Berücksichtigung ihrer Reihenfolge beschränkt. Die Matching-Arten sind dabei durch Wahrheitswerte codiert.

```
boolean matchMethodSignature (MethodSignature i, MethodSignature c, boolean
    differentName, boolean reordered, boolean subTypes, boolean superTypes) {

    [...]

    /* unterschiedliche Anzahl an Argumenten? */
    if(i.getArgumentTypes().size() != c.getArgumentTypes().size())
        return false;

    /* andere Reihenfolge erlaubt? */
    if(reordered){
        /* suche zu jedem Argument von 'i' eine passendes in 'c' */
        for(Type argType1 : i.getArgumentTypes()){
            boolean found = false;
            for(Type argType2 : c.getArgumentTypes()){

                /* vorher schon gematcht? */
                if(argType2.isMatched())
                    continue;

                /* passt Typ genau? */
                if(argType1.equals(argType2)){
                    found = true;
                    argType2.setMatched(true);
                    break;
                }

                /* passt es mit Subtypen */
                if(subTypes && argType1.isSubType(argType2)){
                    found = true;
                    argType2.setMatched(true);
                    break;
                }

                /* passt es mit Supertypen */
                if(superTypes && argType1.isSuperType(argType2)){
                    found = true;
                    argType2.setMatched(true);
```

```
                    break;
                }
            }
            /* kein passender Typ gefunden? */
            if(!found)
                return false;
        }
    }
    else{
        [...]
    }

    [...]

    return true;
}
```

Nachdem sicher gestellt wurde, dass beide Methoden die gleiche Argumentanzahl besitzen, wird danach unterschieden, ob deren Reihenfolge von Bedeutung ist. Ist sie das nicht, so wird jedem Argument der Interfacemethode `i` ein passendes in der Methode `c` gesucht. Dies ist dann der Fall, wenn sie identisch sind oder wenn der eine Sub- oder Supertyp des anderen ist und das entsprechende Flag `subTypes` bzw. `superTypes` gesetzt ist, die jeweiligen Matching-Arten also zugelassen sind.

Bei erfolgreichem Matching wird das Argument aus `c` als „gematcht" markiert, sodass es beim weiteren Abgleich nicht noch einmal einem anderen Parameter von `i` zugeordnet wird.

Konnte zu einem Argumenttyp von `i` kein passender in `c` gefunden werden, so schlägt das Matching dieser Methoden fehl.

Wurde das Flag `reordered` nicht gesetzt, so erfolgt obiger Vergleich der Argumente paarweise.

Auf analoge Weise können die zwei Rückgabetypen miteinander verglichen werden.

Beim Vergleich der Exception-Mengen von `i` und `c` muss zu jeder Exception von `i` (mindestens) eine bezüglich der Typanforderungen passende in `c` gefunden werden und umgekehrt. Exceptions können dabei auch mehrfach gematcht werden.

Eine solche Suche lässt sich so realisieren:

```
/* suche für jede Exception von 'i' */
for(Type iException : i.getExceptionTypes()){
    boolean found = false;
    /* suche passende Exception in 'c' */
    for(Type cException : c.getExceptionTypes()){
        if(iException.equals(cException)){
            found = true;
            break;
        }
        /* Untertypen erlaubt? */
        if(subTypes && iException.isSubType(cException)){
            found = true;
            break;
        }
        /* Obertypen erlaubt? */
        if(superTypes && iException.isSuperType(cException)){
            found = true;
            break;
        }
    }
```

```
        /* kein passender Typ gefunden? */
        if(!found)
            return false;
    }
```

Es wird hier zu jeder Exception von `i` eine in `c` gesucht. Für den umgekehrten Fall brauchen lediglich die beiden `for`-Schleifen vertauscht zu werden. Wann auch Sub- und Supertypen berücksichtigt werden dürfen, wird erneut über die zwei Flags `subTypes` und `superTypes` geregelt.

Es wurde nun vorgestellt, wie die Signaturen zweier Methoden und damit dann auch die Signaturen zweier Klassen miteinander verglichen werden können.

7.3.5. Ergebnis der Analyse

Wurde eine Klasse erfolgreich gegen ein Interface gematcht, so wird vermerkt, welche Matching-Varianten dafür verwendet wurden. Nach Abschluss des Vergleichs aller Klassen wird dieses Ergebnis dann in Form einer XML-Datei gespeichert und graphisch in Form einer Tabelle dargestellt (Abbildung 7.2).

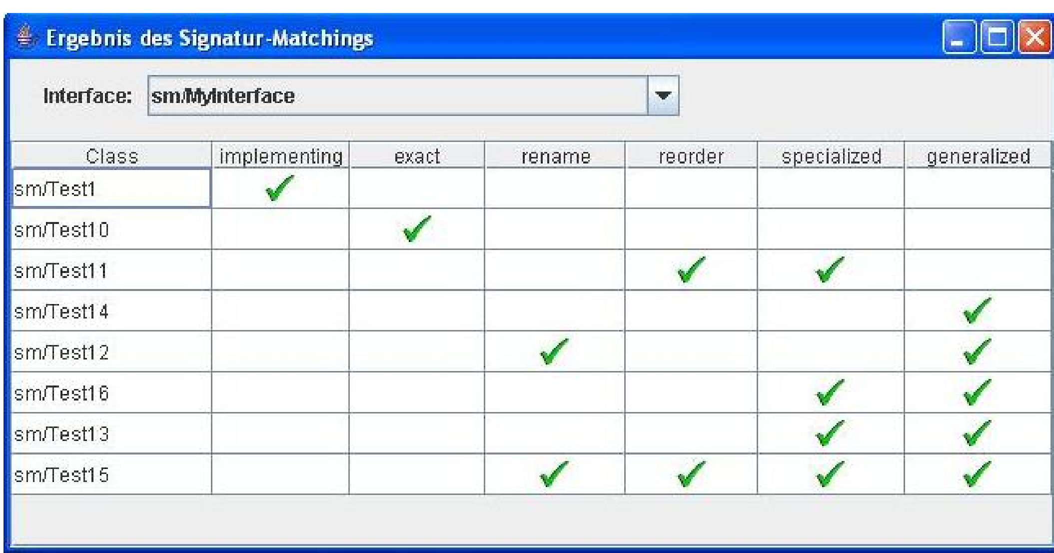


Abbildung 7.2.: Ergebnis des Signatur-Matchings

In dieser ist eine Auflistung der möglichen Implementierungen erkennbar, die sich nach der Güte des Matchings richtet. Zuerst werden Klassen angezeigt, die das betreffende Interface bereits implementieren, gefolgt von Klassen, die dieses exakt matchen. Die weitere Reihenfolge hängt von der bei der Eingabe getätigten Präferenzen des Nutzers bezüglich der restlichen Matching-Arten ab.

8. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Werkzeug zur Analyse und Generierung von Java-Applikationen vorgestellt. Es liefert wertvolle Informationen über die Struktur großer Software. Diese ermöglichen eine größere Transparenz und eine leichtere Wartung der untersuchten Anwendungen. Ferner können so Hinweise bezüglich der Wiederverwendbarkeit bestehender Komponenten gewonnen werden.

Durch Bestimmung der Abhängigkeitsstruktur bietet es Informationen über den Zusammenhang zwischen den einzelnen Programmkomponenten. Diese Analyse wurde auf zwei Hierarchieebenen durchgeführt. Zunächst wurden Beziehungen zwischen den technischen Komponenten einer JavaEE-Anwendung (wie Enterprise JavaBeans oder Application Clients) ermittelt. Für die anschließende Abhängigkeitsanalyse in der untergelagerten Schicht der Java-Klassen wurden zwei alternative Vorgehensweisen vorgestellt: eine Analyse auf Klassen- und eine auf Methodenebene. Letztere liefert genauere Ergebnisse, ist aber umso komplexer, sodass hier ein approximierender Ansatz verfolgt wurde.

Aufbauend auf dieser Untersuchung wurde ein Werkzeug erstellt, über welches aus einer bestehenden JavaEE-Anwendung durch Selektion einzelner Komponenten eine neue erzeugt werden kann. Es wird über eine graphische Benutzeroberfläche angesteuert, in die auch eine Visualisierung der Abhängigkeitsstruktur der JavaEE-Komponenten integriert wurde. An Hand dieser kann der Nutzer direkt die gewünschte Auswahl vornehmen.

Durch geringfügige Anpassungen konnte dieses Programm dahingehend erweitert werden, zu einem Service Provider Interface eine minimale Menge von Java-Klassen zu ermitteln, die zu dessen Implementierung benötigt wird.

Im letzten Teil der Arbeit wurde die Umsetzung eines Signatur-Matchings präsentiert. Sie liefert Erkenntnisse darüber, ob bestehende Softwarekomponenten in der weiteren Entwicklung wiederverwendet werden können. Dies wurde realisiert, indem potentielle Implementierungen zu einem Interface an Hand der Methodensignaturen identifiziert worden sind. Hierbei wurden verschiedene Matching-Arten berücksichtigt.

Zum Abschluss dieser Arbeit soll noch ein Ausblick auf Erweiterungsmöglichkeiten des entwickelten Programms gegeben werden.

Einen ersten Ansatz bietet die Analyse von Abhängigkeiten zwischen Java-Klassen auf Methodenebene. In dieser Arbeit wurde dort ein approximierender Ansatz verwendet. Es ist aber durchaus möglich, an dieser Stelle noch genauer vorzugehen. Die Problematik bestand in der Behandlung von Interfaces und Klassenerweiterungen. So war es erforderlich, zu jeder verwendeten Interfacemethode herauszufinden, welche Implementierung sich dahinter verbergen könnte. Um dies genau feststellen zu können, ist es jedoch erforderlich, den Kontrollfluss, oder genauer alle möglichen Kontrollflüsse, vom Aufruf der Methode bis zur Bedienung des Interfaces durch ihre Implementierung zurückzuverfolgen.

Ist dies realisiert worden, so wäre auch die Behandlung von Reflection flexibler möglich. Eine der Voraussetzungen dieser Arbeit war es (vgl. Abschnitt 3.1.3), dass die Verwendung von Reflection stets nach bestimmten Mustern erfolgt. Dies war nötig, um feststellen zu können, welches Interface von einer per Reflection geladenen Klasse bedient wird. Es wurde gefordert, dass ein entsprechendes Type-Casting direkt nach dem Reflectionaufruf durchgeführt wird. Würde diese Anforderung fallen gelassen werden, so wäre auch hier eine Verfolgung des Kontrollflusses notwendig.

In dieser Arbeit wurde ebenfalls vorausgesetzt, dass als Laufzeitumgebung der untersuchten JavaEE-Anwendungen nur die Application Server IBM Websphere und JBoss verwendet werden. Hier ist eine Erweiterung auf andere Server aber relativ leicht möglich. Es bedarf dafür lediglich einer zusätzlichen Behandlung der speziellen Deployment Deskriptoren des hinzuzufügenden Application Servers. Während der Implementierung wurden bereits Vorkehrungen getroffen, dies leicht in das entwickelte Werkzeug integrieren zu können.

Bei der Visualisierung der Abhängigkeitsstruktur der untersuchten Anwendungen wurde sich auf die JavaEE-Komponenten beschränkt. Hier wäre es natürlich vorstellbar, auch die Beziehungen zwischen Java-Klassen oder gar die ihrer Methoden darzustellen. Programme, die dieses auf unterschiedlichste Art und Weise realisieren können, wurden in Abschnitt 1.3 vorgestellt. Das Hauptproblem aber stellt die Größe der untersuchten Anwendungen dar, die eine aussagekräftige Visualisierung auf dieser Ebene kaum zulässt.

A. Instruktionssatz der Java Virtual Machine

Codierung	Bezeichnung	Args	Klassenref.
0x00	NOP	0	-
0x01	ACONST_NULL	0	-
0x02	ICONST_M1	0	-
0x03	ICONST_0	0	-
0x04	ICONST_1	0	-
0x05	ICONST_2	0	-
0x06	ICONST_3	0	-
0x07	ICONST_4	0	-
0x08	ICONST_5	0	-
0x09	LCONST_0	0	-
0x0a	LCONST_1	0	-
0x0b	FCONST_0	0	-
0x0c	FCONST_1	0	-
0x0d	FCONST_2	0	-
0x0e	DCONST_0	0	-
0x0f	DCONST_1	0	-
0x10	BIPUSH	1	-
0x11	SIPUSH	2	-
0x12	LDC	1	-
0x13	LDC_W	2	-
0x14	LDC2_W	2	-
0x15	ILOAD	1	-
0x16	LLOAD	1	-
0x17	FLOAD	1	-
0x18	DLOAD	1	-
0x19	ALOAD	1	-
0x1a	ILOAD_0	0	-
0x1b	ILOAD_1	0	-
0x1c	ILOAD_2	0	-
0x1d	ILOAD_3	0	-
0x1e	LLOAD_0	0	-
0x1f	LLOAD_1	0	-
0x20	LLOAD_2	0	-
0x21	LLOAD_3	0	-
0x22	FLOAD_0	0	-
0x23	FLOAD_1	0	-
0x24	FLOAD_2	0	-
0x25	FLOAD_3	0	-
0x26	DLOAD_0	0	-
0x27	DLOAD_1	0	-
0x28	DLOAD_2	0	-
0x29	DLOAD_3	0	-
0x2a	ALOAD_0	0	-
0x2b	ALOAD_1	0	-
0x2c	ALOAD_2	0	-
0x2d	ALOAD_3	0	-
0x2e	IALOAD	0	-
0x2f	LALOAD	0	-
0x30	FALOAD	0	-
0x31	DALOAD	0	-

Codierung	Bezeichnung	Args	Klassenref.
0x32	AALOAD	0	-
0x33	BALOAD	0	-
0x34	CALOAD	0	-
0x35	SALOAD	0	-
0x36	ISTORE	1	-
0x37	LSTORE	1	-
0x38	FSTORE	1	-
0x39	DSTORE	1	-
0x3a	ASTORE	1	-
0x3b	ISTORE_0	0	-
0x3c	ISTORE_1	0	-
0x3d	ISTORE_2	0	-
0x3e	ISTORE_3	0	-
0x3f	LSTORE_0	0	-
0x40	LSTORE_1	0	-
0x41	LSTORE_2	0	-
0x42	LSTORE_3	0	-
0x43	FSTORE_0	0	-
0x44	FSTORE_1	0	-
0x45	FSTORE_2	0	-
0x46	FSTORE_3	0	-
0x47	DSTORE_0	0	-
0x48	DSTORE_1	0	-
0x49	DSTORE_2	0	-
0x4a	DSTORE_3	0	-
0x4b	ASTORE_0	0	-
0x4c	ASTORE_1	0	-
0x4d	ASTORE_2	0	-
0x4e	ASTORE_3	0	-
0x4f	IASTORE	0	-
0x50	LASTORE	0	-
0x51	FASTORE	0	-
0x52	DASTORE	0	-
0x53	AASTORE	0	-
0x54	BASTORE	0	-
0x55	CASTORE	0	-
0x56	SASTORE	0	-
0x57	POP	0	-
0x58	POP2	0	-
0x59	DUP	0	-
0x5a	DUP_X1	0	-
0x5b	DUP_X2	0	-
0x5c	DUP2	0	-
0x5d	DUP2_X1	0	-
0x5e	DUP2_X2	0	-
0x5f	SWAP	0	-
0x60	IADD	0	-
0x61	LADD	0	-
0x62	FADD	0	-
0x63	DADD	0	-

Codierung	Bezeichnung	Args	Klassenref.
0x64	ISUB	0	-
0x65	LSUB	0	-
0x66	FSUB	0	-
0x67	DSUB	0	-
0x68	IMUL	0	-
0x69	LMUL	0	-
0x6a	FMUL	0	-
0x6b	DMUL	0	-
0x6c	IDIV	0	-
0x6d	LDIV	0	-
0x6e	FDIV	0	-
0x6f	DDIV	0	-
0x70	IREM	0	-
0x71	LREM	0	-
0x72	FREM	0	-
0x73	DREM	0	-
0x74	INEG	0	-
0x75	LNEG	0	-
0x76	FNEG	0	-
0x77	DNEG	0	-
0x78	ISHL	0	-
0x79	LSHL	0	-
0x7a	ISHR	0	-
0x7b	LSHR	0	-
0x7c	IUSHR	0	-
0x7d	LUSHR	0	-
0x7e	IAND	0	-
0x7f	LAND	0	-
0x80	IOR	0	-
0x81	LOR	0	-
0x82	IXOR	0	-
0x83	LXOR	0	-
0x84	IINC	2	-
0x85	I2L	0	-
0x86	I2F	0	-
0x87	I2D	0	-
0x88	L2I	0	-
0x89	L2F	0	-
0x8a	L2D	0	-
0x8b	F2I	0	-
0x8c	F2L	0	-
0x8d	F2D	0	-
0x8e	D2I	0	-
0x8f	D2L	0	-
0x90	D2F	0	-
0x91	I2B	0	-
0x92	I2C	0	-
0x93	I2S	0	-
0x94	LCMP	0	-
0x95	FCMPL	0	-
0x96	FCMPG	0	-
0x97	DCMPL	0	-
0x98	DCMPG	0	-
0x99	IFEQ	2	-
0x9a	IFNE	2	-
0x9b	IFLT	2	-
0x9c	IFGE	2	-
0x9d	IFGT	2	-
0x9e	IFLE	2	-
0x9f	IF_ICMPEQ	2	-

Codierung	Bezeichnung	Args	Klassenref.
0xa0	IF_ICMPNE	2	-
0xa1	IF_ICMPLT	2	-
0xa2	IF_ICMPGE	2	-
0xa3	IF_ICMPGT	2	-
0xa4	IF_ICMPLE	2	-
0xa5	IF_ACMPEQ	2	-
0xa6	IF_ACMPNE	2	-
0xa7	GOTO	2	-
0xa8	JSR	2	-
0xa9	RET	1	-
0xaa	TABLESWITCH	(*)	-
0xab	LOOKUPSWITCH	(**)	-
0xac	IRETURN	0	-
0xad	LRETURN	0	-
0xae	FRETURN	0	-
0xaf	DRETURN	0	-
0xb0	ARETURN	0	-
0xb1	RETURN	0	-
0xb2	GETSTATIC	2	ja
0xb3	PUTSTATIC	2	ja
0xb4	GETFIELD	2	ja
0xb5	PUTFIELD	2	ja
0xb6	INVOKEVIRTUAL	2	ja
0xb7	INVOKESPECIAL	2	ja
0xb8	INVOKESTATIC	2	ja
0xb9	INVOKEINTERFACE	4	ja
0xba	XXXUNUSEDXXX	0	-
0xbb	NEW	2	ja
0xbc	NEWARRAY	1	-
0xbd	ANEWARRAY	2	ja
0xbe	ARRAYLENGTH	0	-
0xbf	ATHROW	0	-
0xc0	CHECKCAST	2	ja
0xc1	INSTANCEOF	2	ja
0xc2	MONITORENTER	0	-
0xc3	MONITOREXIT	0	-
0xc4	WIDE	(***)	-
0xc5	MULTIANEWARRAY	3	ja
0xc6	IFNULL	2	-
0xc7	IFNONNULL	2	-
0xc8	GOTOW	4	-
0xc9	JSRW	4	-

(*): TABLESWITCH hat variable Anzahl an Operanden:

- 0-3 Null-Bytes als Padding
- 4 Bytes für Default-Wert
- 4 Bytes für untere Grenze UG
- 4 Bytes für obere Grenze OG
- OG-UG+1 4-Byte große Einträge der Switch-Tabelle

(**): LOOKUPSWITCH hat variable Anzahl an Operanden:

- 0-3 Null-Bytes als Padding
- 4 Bytes für Default-Wert
- 4 Bytes für die Anzahl an Einträgen NUM
- NUM 4-Byte große Einträge der Switch-Tabelle

(***): WIDE beeinflusst Zahl der Operanden anderer Instruktionen:

- steht WIDE vor ILOAD, FLOAD, ALOAD, LLOAS, DLOAD, ISTORE, FSTORE, ASTORE, LSTORE, DSTORE, oder RET, so haben diese Instruktionen 2 anstatt 1 Operanden
- steht WIDE vor IINC, so hat diese Instruktion 4 anstatt 2 Operanden

Literaturverzeichnis

[1] ZAREMSKI, AMY MOORMANN und JEANNETTE M. WING: *Signature Matching, a Tool for Using Software Libraries*. ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 4, No. 2, April 1995. Online: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/venari/www/tosem95.html.

[2] SUN MICROSYSTEMS, INC.: *JAR File Specification*. Online: http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html, 2003.

[3] HOSANEE, MATTHEW: *Manually Creating a Simple Web ARchive (WAR) File*. Online: http://access1.sun.com/techarticles/simple.WAR.html, Juni 2002.

[4] JENDROCK, ERIC, JENNIFER BALL ANDD EBBIE CARSON, IAN EVANS, SCOTT FORDIN und KIM HAASE: *The Java EE 5 Tutorial, Third Edition*, Kapitel 1 Overview. Addison-Wesley Longman, Februar 2007. Online: http://java.sun.com/javaee/5/docs/tutorial/doc/.

[5] LEE, ROSANNA und SCOTT SELIGMAN: *JNDI API Tutorial and Reference*. Prentice Hall, Juni 2000.

[6] BEECH, DAVID, NOAH MENDELSOHN, MURRAY MALONEY und HENRY S. THOMPSON: *XML Schema Part 1: Structures Second Edition*. W3C Recommendation, W3C, Oktober 2004. Online: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[7] SUN MICROSYSTEMS, INC.: *JAXB*. Online: https://jaxb.dev.java.net/, 2007.

[8] SUN MICROSYSTEMS, INC.: *J2EE DTDs*. Online: http://java.sun.com/dtd/, 2007.

[9] HUNTER, JASON: *JDOM*. Online: http://www.jdom.org/, 2000.

[10] JBOSS: *JBoss J2EE DTDs*. Online: http://www.jboss.org/j2ee/dtd/, 2007.

[11] LINDHOLM, TIM und FRANK YELLIN: *The Java Virtual Machine Specification, Second Edition*, Kapitel 4 The class File Format, 6 The Java Virtual Machine Instruction Set. Prentice Hall, April 1999. Online: http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html.

[12] THE APACHE SOFTWARE FOUNDATION: *The Apache Ant Project*. Online: http://ant.apache.org/, 2004.

[13] LOY, MARC, ROBERT ECKSTEIN, DAVE WOOD, JAMES ELLIOTT und BRIAN COLE: *Java Swing, Second Edition*. O'Reilly, 2002.

[14] NOACK, ANDREAS: *Energy Models for Drawing Clustered Small-World Graphs*. Computer Science Report 07/03, Brandenburg University of Technology at Cottbus, Mai 2003. Online: `http://www-sst.informatik.tu-cottbus.de/~an/GD/`.

[15] NOACK, ANDREAS: *An Energy Model for Visual Graph Clustering*. Proceedings of the 11th International Symposium on Graph Drawing (GD 2003, Perugia, Italy, Sep. 21-24), LNCS 2912, Seiten 425–436, 2004. Online: `http://www-sst.informatik.tu-cottbus.de/~an/GD/`.

[16] NOACK, ANDREAS: *Energy-Based Clustering of Graphs with Nonuniform Degrees*. Proceedings of the 13th International Symposium on Graph Drawing (GD 2005, Limerick, Ireland, Sep. 12-14), 2005. Online: `http://www-sst.informatik.tu-cottbus.de/~an/GD/`.

[17] JOHN ZUKOWSKI: *Java AWT Reference*. O'Reilly, 1997. Online: `http://www.oreilly.com/catalog/javawt/book/index.html`.

[18] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java Language Specification, Third Edition*, Kapitel 4.10 Subtyping, 8.4 Method Declarations, 9 Interfaces. Prentice Hall, Juni 2005. Online: `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`.

Abbildungsverzeichnis

1.1. Ebenen der Abhängigkeitsanalyse 2

2.1. Mehrschichtige Anwendungen 6
2.2. Aufbau einer EAR-Datei 7

3.1. Ebenen der Abhängigkeiten 12
3.2. Diagramm der von JAXB aus dem Schema zu `application.xml` erzeugten Klassen . 15
3.3. Beispiel: `ejb-jar.xml` 17

4.1. Praktischer Vergleich der Analyseebenen 54

5.1. Visualisierung der Abhängigkeitsstruktur im Rahmen der GUI 56
5.2. Verschiebungsrichtungen 59
5.3. Reihenfolge der Anordnung 59
5.4. Manuelles Knotenverschieben 59
5.5. Visualisierung der Abhängigkeitsstruktur einer großen Applikation 61
5.6. Einstellungs-Dialoge 62
5.7. Auswahl von Klassen bei Reflection-Behandlung 63
5.8. Vorschau über Entwicklung der Abhängigkeiten bei Reflection-Behandlung 63

6.1. GUI für SPI-Behandlung 70

7.1. Eingabe fürs Signatur-Matching 76
7.2. Ergebnis des Signatur-Matchings 81

Abkürzungen

CORBA	Common Object Request Broker Architecture
DTD	Document Type Definition
EAR	Enterprise Application Archive
GIF	Graphics Interchange Format
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JavaEE	Java Platform, Enterprise Edition
JNDI	Java Naming and Directory Interface
JSP	JavaServer Pages
JVM	Java Virtual Machine
RMI	Remote Method Invocation
SAX	Simple API for XML
SPI	Service Provider Interface
WAR	Web Application Archive
XMI	XML Metadata Interchange
XML	Extensible Markup Language