Hochschule für Technik und Wirtschaft Karlsruhe
Fakultät für Informatik

Diplomarbeit

PHP in Java - ein Experiment

von

Michael Bayer

Betreuer: Timm Friebe, Prof. Dr. Thomas Fuchß

Karlsruhe, 2007

Ich erkläre hiermit, dass ich die vorliegende Diplomarbeit selbständig erarbeitet und verfasst habe; aus fremden Quellen übernommene Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

Karlsruhe, den

Michael Bayer

Zusammenfassung

Die Anforderungen an Softwaresysteme werden heute immer komplexer, gleichzeitig aber werden Produktlebenszyklen und somit auch die Entwicklungszeit immer kürzer. Deswegen gewinnen diensteorientierte Architekturen (SOA) immer mehr an Bedeutung, da sie eine Wiederverwendung bereits entwickelter Komponenten erlauben. Innerhalb der 1&1-Firmengruppe werden hierzu Technologien der *Java Enterprise Edition* eingesetzt. Für Entwickler anderer Programmiersprachen ist es daher wichtig, mit solchen Diensten kommunizieren zu können. Die Abteilung *i::Dev*, in welcher diese Diplomarbeit erstellt wurde, setzt für viele Zwecke die Skriptsprache PHP ein, welche keine Möglichkeit bietet mit JavaEE-Diensten zu kommunizieren. Dies hat unter anderem zur Folge, dass sehr viele bestehende Programme und Komponenten keine Möglichkeit besitzen mit Java EE-Diensten zu kommunizieren.

Im Laufe dieser Arbeit wird ein Softwaresystem erstellt, das es ermöglicht PHP-Skripte innerhalb einer Java Virtual Machine auszuführen. Weiterhin wird ein Datenaustausch zwischen den Laufzeitumgebungen der beiden Programmiersprachen, sowie der Zugriff auf Funktionen, Objekte und Methoden der jeweils anderen Sprache ermöglicht, mit dem Ziel Enterprise Java Beans in PHP zu entwickeln und transparent in einem JavaEE Application Server zu installieren.

Zu diesem Zweck werden verschiedene Technologien zum Einbetten einer Skriptsprache in Java evaluiert. Die so gewonnenen Erkenntnisse werden schließlich genutzt, um eine dieser Technologien auszuwählen: den *Java Specification Request 223*. Im zweiten Teil der Arbeit wird eine JSR 223-Implementierung für PHP erstellt, sowie der PHP-Interpreter derart erweitert, dass er auf den vollen Funktionalitätsumfang Javas zugreifen kann. Im dritten Teil der Arbeit wird die erstellte Bibliothek genutzt, um voll funktionsfähige Enterprise Java Beans in PHP zu realisieren. Hierzu werden EJB 3.0 Technologien eingesetzt, und es wird erörtert wie ein PHP-Anwender Stateful- und Stateless Session Beans sowie Message Driven Beans in seiner gewohnten Programmierumgebung entwickeln kann.

Die entwickelte Bibliothek wird unter dem Namen *Turpitude* als Open Source-Projekt veröffentlicht.

Abstract

Enterprise software systems today face ever more complex requirements, while product lifecycles and with them development timespans grow shorter. Therefore the importance of service-oriented architectures grows rapidly, because they allow the re-use of services and software-components. Whithin 1&1 *Java Enterprise Edition* technologies are used to build such software-systems. It is important for developers of other languages to be able to communicate with JavaEE-services. The department *i::Dev* within this thesis was written uses the scripting language PHP for many projects. PHP does not have the capabilities to communicate with such services. This results in many existing programs and components lacking the capablilities to communicate with JavaEE-services.

This thesis investigates different technologies that allow embeding scripting languages into a Java Virtual Machine. One of these technologies - the *Java Specification Request 223* - is then used to implement a library that not only allows users to execute PHP-scripts within a JVM, but also extends the PHP runtime environment to enable PHP-scripts to create and access Java classes, objects and methods. It further enables the Java-user to access PHP objects, methods and functions. Ultimate goal of this thesis is to be able to write *Enterprise Java Beans* in PHP and deploy them transparently in a JavaEE Application Server. The third part of this thesis then shows how the library can be used to implement EJBs in PHP, at the same time evaluating the differences between EJB 3.0 and earlier specifications.

Inhaltsverzeichnis

Danksagung

An dieser Stelle möchte ich mich bei meinen Betreuern von Hochschule und 1&1 für die hervorragende Betreuung und Unterstützung sowohl in fachlichen als auch in methodischen Bereichen bedanken. Diese Diplomarbeit hätte ohne Herrn Prof. Dr. Fuchß und Herrn Timm Friebe in dieser Art nicht erstellt werden können.

Weiterhin gilt mein Dank denjenigen aus der Abteilung i::Dev die mir immer wieder durch Hinweise und Nachfragen Unterstützung gegeben haben, und ohne deren Vorarbeit eine Umsetzung ebenfalls nicht möglich gewesen wäre, insbesondere Alexander Kiesel und Christian Gellweiler.

Schliesslich möchte ich mich bei denjenigen bedanken die mir durch Korrekturlesen oder Anregungen geholfen haben: Ulrich Wenckebach, Claudia Nottebrock, Christoph Schneider, Stephan Kölle und Stephan Dienst.

Kapitel 1

Einleitung

Dieses Kapitel soll die Firma *1&1 Internet AG* in kürze Vorstellen sowie den Kontext, in welchem diese Arbeit erstellt wurde, erläutern. Desweiteren wird die Aufgabenstellung und der Aufbau dieses Dokumentes beschrieben.

1.1 Umfeld der Aufgabe

Die *1&1 Internet AG* (1&1) ist eine einhunderprozentige Tochter der *United Internet AG* (UI) und deckt innerhalb der Firmengruppe die Produktbereiche Informationsmanagement, Internet Access und Webhosting ab. Die Produkte und Gechäftsprozesse als solches werden nicht nur unternehmensintern sondern auch in enger Zusammenarbeit mit anderen Marken der UI entwickelt. Daraus erwächst die Notwendigkeit Softwaresysteme nach außen verfügbar zu machen, was durch verstärkten Einsatz dienstebasierter Softwarearchitekturen (*Serviceorientierte Architektur* - SOA, siehe 2.1) erreicht wird.

Die Abteilung *i::Dev*, in welcher diese Diplomarbeit erstellt wurde, entwickelt hauptsächlich Anwendungen zur internen Verwendung. Sie betreut unter anderem die komplette Rechteverwaltung für das Intranet der UI sowie eine Vielzahl von weiteren Applikationen, von Datenbankfrontends für den Support über eine Dokumentenverwaltung, welche alle Kundenfaxe enthält, bis hin zu Projektmanagement- und Statistikanwendungen.

Hierzu wird hauptsächlich die Skriptsprache PHP (vgl. 2.2.1) eingesetzt, da diese sehr schnelle Entwicklungszyklen erlaubt, und nicht zuletzt weil dadurch sehr viel historisch vorhandener PHP-Quelltext wiederverwendet werden kann. Weiterhin werden im Zuge der Umstellung auf SOA viele neue Dienste in Java implementiert, da Java sich aufgrund schon vorhandener Lösungen und Technologien besser als PHP für diese Art der Entwicklung eignet.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist die Entwickung eines Softwaresystems, das es erlaubt PHP-Skripte innerhalb eines *Java Enterprise Edition Application Servers* als *Enterprise Java Beans* auszuführen. Zum Einen um einem PHP-Anwender die Vorteile eines solchen Application Servers und der Programmiersprache Java mit all ihren existierenden Technologien und Bibliotheken zugänglich zu machen, und zum Anderen um die Kommunikation und den Datenaustausch zwischen diesen beiden Welten zu vereinfachen, was beiden Seiten großen Nutzen bringen kann. Hierzu muss eine geeignete Schnittstelle zwischen Java und PHP, sowie Möglichkeiten des Datenaustausches zwischen den Laufzeitumgebungen der beiden Programmiersprachen gefunden werden. Innerhalb dieser PHP-Skripte soll ein Zugriff auf möglichst den kompletten Funktionsumfang der Java Virtual Machine möglich sein, und auf die derart ausgeführten Skripte soll der Zugriff von außen über die üblichen Enterprise Java Schnittstellen wie *RMI* erfolgen können. Hierzu wird ein Mechanismus benötigt, welcher neben dem Datenaustausch auch den Methodenaufruf sowohl von Java nach PHP, als auch auf umgekehrtem Wege erlaubt. Für den Austausch von Daten muss eine Konvention gefunden werden, die Typkonversionen sowohl simpler (skalarer), als auch komplexer Datentypen zwischen den beiden Programmiersprachen festlegt. Komplexe Datentypen wie Objekte und Arrays sollen in der jeweils anderen Programmiersprache nicht nur als Kopie, sondern als Referenz zugänglich sein, und der Aufruf von Methoden und der Zugriff auf Attribute soll in der jeweiligen Wirtsprogrammiersprache möglichst transparent und intuitiv möglich sein. Besonderen Wert wird auf den einfachen Einsatz dieses Softwaresystems gelegt; so soll der Anwender seine geschriebenen PHP-Beans möglichst ohne zusätzlichen, immer gleichen Code ausbringen können (*easy deployment*). Es sollen die wichtigsten Java Enterprise Technologien wie *Entity-, Session- und Message-Driven-Beans* unterstützt werden, und die Lösung soll unabhängig von der verwendeten PHP-Version funktionieren. Ein wesentlicher Gesichtspunkt ist die Auswahl eines geeigneten Standards zur Skriptintegration in Java, es sollen bestehende Lösungen betrachtet und vorhandene Standards hinsichtlich ihrer Einsatzmöglichkeiten analysiert werden. Diese Aufgabenstellung soll lediglich einen Rahmen der zu erreichenden Ziele stecken, das System soll leicht um weitere Anforderungen erweiterbar sein, was den Forschungscharakter der Diplomarbeit unterstreicht.

1.3 Struktur

An dieser Stelle soll ein Einblick in die Struktur dieser Diplomarbeit gewährt und zu jedem Teilabschnitt eine kurze Erläuterung des jeweiligen Inhaltes gegeben werden. Das vorliegende Dokument unterteilt sich in folgende Kapitel:

1. **Einleitung**

2. **Grundlagen**. Dieses Kapitel bietet eine Einführung in bereits vorhandene Technologien, die entweder für die Lösung der Aufgabe relevant sind, oder aber Alternativen zu den verwendeten Technologien darstellen. Ausserdem werden Begrifflichkeiten für den weiteren Verlauf des Dokumentes eingeführt und erklärt.

3. **Java und Skriptsprachen**. Hier wird erläutert auf welche Arten und Weisen Skriptsprachen prinzipiell in Java eingebettet werden können. Es werden zwei weit verbreitete Standards vorgestellt und es wird erläutert welche Technologie bei der Realisierung der Aufgabe den Vorzug erhielt.

4. **Java und PHP**. In diesem Kapitel soll eine JSR 223-Implementierung für PHP namens *Turpitude* entworfen und erarbeitet werden. Hierzu wird zunächst die Aufgabe analysiert und die Anforderungen die diese stellt erörtert. Diese Anforderungen werden dann genutzt, um das Vorgehen bei der Implementierung zu planen, und um eine Architektur zu entwerfen, die diese Anforderungen möglichst komplett erfüllt. Schliesslich werden die gewonnenen Erkenntnisse genutzt, um die erarbeitete Vorgehensweise in die Tat umzusetzen und eine Bibliothek zu erstellen, die es dem Anwender erlaubt PHP-Skripte in Java auszuführen, Daten an diese Skripte zu senden, aus Java heraus PHP-Funktionen aufzurufen, und aus diesen Skripten heraus widerum Java-Funktionalität zu nutzen.

5. **JBoss und PHP**. Dieses Kapitel soll eine mögliche Anwendung für Turpitude erarbeiten, um einen kleinen Teil der Möglichkeiten die diese Bibliothek bietet aufzuzeigen. Konkret wird die Bibliothek in einem Java Application Server eingesetzt um verschiedene Enterprise Beans in PHP zu implementieren.

6. **Fazit und Ausblick**. Dieser Abschnitt fasst die geleistete Arbeit zusammen und klärt in wie weit die Aufgabenstellung erfüllt wurde, und welche Einsatzmöglichkeiten noch für die entwickelte Technologie denkbar sind.

7. **Anhänge**. In den Anhängen findet der Leser eine ausführliche Dokumentation der Bibliothek, sowohl des Java- als auch des PHP-Teiles, sowie eine Einführung in Verzeichnis- und Dateistruktur des beiliegenden Datenträgers.

Kapitel 2

Grundlagen

Dieses Kapitel stellt bereits vorhandene Technologien vor, welche für die Lösung der Aufgabenstellung relevant sind. Weiterhin werden Technologien angesprochen, die alternative Lösungen darstellen könnten. In diesen Fällen wird kurz darauf eingegangen, warum sie keine Verwendung finden. Außerdem werden Begrifflichkeiten für den weiteren Verlauf des Dokumentes eingeführt und erklärt.

2.1 Serviceorientierte Architektur - SOA

Der Begriff *Serviceorientierte Architektur* oder englisch *Service Oriented Architecture*, auch *diensteorientierte Architektur*, beschreibt ein Softwarearchitekturkonzept welches versucht durch lose Kopplung autarker Module - sogenannter Services - Geschäftsprozesse besser und schneller abbilden zu können als traditionelle Architekturen. Diese Services bilden einzelne Schritte der Geschäftsprozesse nach und bieten standardisierte Schnittstellen an, über welche sie angesprochen werden können. Dienste werden von einem *service provider* angeboten und von einem *service consumer* mittels eines *service request* angesprochen; der Dienst antwortet mittels eines *service response*. Die Koppelung mehrerer Dienste wird über ein oder mehrere Protokolle wie zum Beispiel *SOAP* * oder *RMI* †, aber auch durch den Einsatz ganzer dienstorientierter Technologien wie *CORBA* ‡ oder *Enterprise Java Beans* (vgl. 2.6 und 5) erreicht. Aufgrund der Plattformunabhängigkeit solcher Technologien eignet sich

*Simple Object Access Protocol, XML-basiertes Protokoll zum Nachrichtenaustausch über Rechnernetzwerke, siehe 2.4

†Remote Method Invocation, API zum Aufruf entfernter Methoden, siehe 2.6

‡Common Object Request Broker Architecture - objektorientierte, plattformunabhängige Middleware die Protokolle und Dienste spezifiziert, die das erstellen verteilter Anwendungen in heterogenen Umgebungen vereinfachen soll. CORBA wird von der Object Management Group entwickelt, siehe [OMG06]

eine solche Architektur auch besonders um vorhandene Strukturen miteinander zu verbinden, und die Heterogenität verschiedener Systeme zu überwinden. Ein weiterer wesentlicher Aspekt von SOA ist die Kapselung des Zugriffes auf persistente Daten durch einzelne Services, welche als einzige schreibend auf diese Daten zugreifen können, was die Gewährleistung der Konsitenz solcher persistenter Daten erheblich vereinfacht. Der Begriff SOA fasst eine Vielzahl von Technologien und Herangehensweisen zusammen, und ihn an dieser Stelle komplett zu erläutern würde den Rahmen des Dokumentes sprengen, interessierte Leser sollten eines der zahlreichen Bücher zu diesem Thema konsultieren, beispielsweise [All06] und [Erl04].

2.2 Skriptsprachen

Als Skriptsprachen bezeichnet man Programmiersprachen, welche ursprünglich vor allem zur Lösung kleinerer Programmieraufgaben entwickelt wurden. Sie stellen im Allgemeinen keine so hohen formalen Ansprüche an den Programmierer wie "vollständige" Programmiersprachen, so verzichten sie beispielsweise oft auf den Deklarationszwang von Variablen. Weitere, häufig bei Skriptsprachen anzutreffende Merkmale sind unter anderem die dynamische, beziehungsweise oft auch komplett fehlende Typisierung von Variablen, die automatische Speicherverwaltung, und vor allem das übersetzungsfreie Ausführen - die Programme werden *interpretiert*. Dies führt allerdings dazu, dass Skriptsprachen oft nicht so effizient sind wie direkt übersetzte Programme. Weiterhin treten so viele Fehler die ein Compiler finden, beziehungsweise komplett verhindern würde erst zur Laufzeit und somit möglicherweise im Produktivbetrieb auf. Der Übergang zwischen klassischen Programmiersprachen und Skriptsprachen ist heute jedoch fließend: Sprachen wie beispielsweise Java bieten ebenfalls eine automatisierte Speicherverwaltung und werden nicht direkt in Maschinencode sondern in Bytecode übersetzt, welcher dann interpretiert wird, während Skriptsprachen wie Python oder Ruby teilweise sehr hohe formale Ansprüche stellen und sehr robust sind. Die klassischen Einsatzgebiete von Skriptsprachen werden heute erweitert durch Webanwendungen und *rapid prototyping**. Sie werden auch gerne als sogenannte *glue languages* - als verbindender "Kleber" zwischen verschiedenen Systemen, Programmen und Programmiersprachen - eingesetzt. Skriptsprachen sind außerdem beliebt um Softwaresysteme für weniger versierte Anwender leicht erweiterbar zu machen, indem sie eine kontrollierte, leicht erlernbare und trotzdem ausreichend mächtige Umgebung bieten, in der dieser Anwender der Applikation eigene Funktionalität hinzufügen kann. Im weiteren Verlauf dieses Dokumentes werden Programme, welche in einer Skriptsprache geschrieben sind auch kurz als *Skripte* bezeichnet. Beispiele weitverbreiteter Skriptsprachen sind das in dieser Arbeit behandelte *PHP*, der Klassiker *Perl* sowie die neueren Sprachen *Ruby* und *Python*.

*Das Entwickeln eines Prototyps des zu erstellenden Softwaresystems, ohne die vollen Ansprüche an Umfang und Qualität erfüllen zu wollen.

2.2.1 PHP

Das PHP-Manual [PHP06a] schreibt über PHP: „PHP (Akronym für "PHP: Hypertext Preprocessor") ist eine weit verbreitete und für den allgemeinen Gebrauch bestimmte Open Source Skriptsprache, welche speziell für die Webprogrammierung geeignet ist, und in HTML eingebettet werden kann."

Das heutige PHP entstand aus den ursprünglich 1994 von *Rasmus Lerdorf* in C geschriebenen und über CGI * ausgeführten "Personal Home Page Tools". 1995 wurden diese von Lerdorf erstmals als "PHP/FI" veröffentlicht, nachdem er sie mit einem Interpreter für Formulardaten ausgestattet hatte. 1997 schrieben die beiden israelischen Entwickler *Zeev Suraski* und *Andi Gutmans* den Parser neu und schufen so die Grundlage für PHP 3, welches 1998 veröffentlicht wurde. Diese Version bot auch erstmals grundlegende Möglichkeiten zur objektorientierten Programmierung, allerdings war die gesamte Standardbibliothek noch prozedural angelegt. Suraski und Gutmans fingen daraufhin an den Kern von PHP neu zu schreiben, welcher seit 1999 als *Zend Engine* von der neu gegründeten *Zend Technologies* vertrieben wird. Im Mai 2000 wurde PHP 4 der Öffentlichkeit als erstes PHP vorgestellt, welches die Zend Engine in der Version 1.0 nutzte (siehe auch [ZEN06]). PHP 4 wird bis heute von Zend unterstützt und wird bei 1&1 noch produktiv eingesetzt, obwohl es noch sehr unter den mangelhaften Fähigkeiten zur Objektorientierung von PHP 3 leidet. Es werden weder Kapselung der Daten, zum Beispiel durch private Variablen, noch Destruktoren oder Fehlerbehandlung über Exceptions unterstützt. Im Juli 2004 wurde PHP 5 zusammen mit der Zend Engine II veröffentlicht und bot lang erwartete Features wie robuste Objektorientierung, native SOAP-Unterstützung (siehe 2.4), Iteratorkonstrukte und Exceptions. Zu diesem Zeitpunkt wird an der nächsten PHP Version 6 entwickelt. Diese Version befindet sich allerdings noch in einem sehr frühen Stadium, und es wird sowohl Zend-intern als auch öffentlich noch heftig über die Richtung debatiert, in die PHP sich in Zukunft bewegen soll.

PHP Quelltext wird - wie bei vielen Skriptsprachen, zum Beispiel Perl - nicht kompiliert sondern zur Laufzeit ausgewertet und interpretiert. Die Sprache bietet nur einige wenige Datentypen, welche alle in einer einzigen Art von Variable gespeichert werden (*weak typing*) - sprich Variablen können zur Laufzeit ihren Typ ändern, was von vielen als ein weiterer Kritikpunkt angesehen wird, da es zu Laufzeitfehlern kommen kann die in anderen Sprachen schon von vorneherein durch den Compiler ausgeschlossen sind †. Dieser Variablentyp wird *zval* genannt, und ist auf C-Ebene ein `struct`, der neben einem Feld das den aktuell gespeicherten Typ vorhält einen `union` enthält, in dem der eigentliche Wert der Variable gespeichert wird. Ein weiteres Problem ist, dass der Sprachumfang von PHP durch Dritte mittels *Extensions* genannter Erweiterungen erweitert werden kann, und wird. Dies muss allerdings

*Common Gateway Interface, Schnittstelle zwischen einem Webserver und einem externen Programm, siehe 2.2.2

†siehe hierzu auch den Artikel von Bruce Eckel, zu finden unter [Eck03]

zur Kompilierzeit von PHP geschehen, was wiederum dazu führt, dass verschiedene PHP-Installationen verschiedenen Funktionsumfangs sein können. Dadruch dass die Extensions meist durch Dritte entwickelt werden kann es vorkommen, dass verschiedene gleichartige Extensions - zum Beispiel die Datenbankerweiterungen zu MySQL* und Sybase† - nicht nur unterschiedliche Methodensignaturen, sondern auch unterschiedliches Verhalten ähnlicher Methoden haben.

Der Einsatz von PHP innerhalb von 1&1 ist historisch vor allem dadurch zu erklären, dass zunächst keine konkurrenzfähigen Technologien existierten, und später erstens schon sehr viel in diese Technologie investiert worden war und zweitens etwaige technischen Nachteile von PHP durch im Unternehmen vorhandenes Know-How sehr gut ausgeglichen werden konnten.

Die Tatsache, dass PHP ursprünglich als reine Web-Programmiersprache konzipiert wurde ist heute noch deutlich sichtbar, unter anderem an Variablen und Funktionen, die ohne eine Web-Umgebung keinen Sinn ergeben. Vor allem macht sie sich aber in der Zend-Engine selbst, bemerkbar, da an vielen Stellen noch zwischen Modul (dem Interpreter) und Request (dem einzelnen Zugriff auf eine URL) unterschieden wird. Dies mag auch der Grund dafür sein, dass keine dem Application Server vergleichbare Technologie für PHP existiert. Für PHP ist das Ausführen einer alleinstehenden Applikation eher die Ausnahme als die Regel.

Dennoch erfreut sich PHP heute sehr großer Beliebtheit, wohl vor allem aufgrund der hohen Anzahl von Webservern, die ihren Benutzern die Möglichkeit bieten mittels PHP dynamische Inhalte zu generieren. Obwohl Perl - als größter "Konkurrent" zu PHP - wohl eine ähnlich hohe Verbreitung hat schreckt gerade viele Einsteiger die komplizierte Syntax dieser Sprache ab.

2.2.2 Common Gateway Interface - CGI

Skriptsprachen im Allgemeinen und PHP im Besonderen sind sehr stark mit dem *Common Gateway Interface* verbunden. CGI erlaubt einem Webserver (zum Beispiel dem *apache.org HTTP Server* [APA06]) externe Programme zu nutzen, um neben statischen Inhalten auch die Generierung dynamischer Inhalte zu ermöglichen. Hierzu wird der Zugriff (engl. *Request*) mittels Umgebungsvariablen formalisiert und an ein externes Programm weitergegeben, welches für jeden Zugriff neu gestartet wird. Diese Programme sind oftmals in Skriptsprachen geschrieben, auch weil diese oftmals sehr komfortable Methoden besitzen um Strings zu manipulieren. CGI wurde mit *FastCGI* weiterentwickelt, FastCGI-Programme werden nicht für jeden Request neu gestartet, sondern werden im Speicher gehalten und nur mit den neu ankommenden Requests versorgt. Einem ähnlichen Prinzip folgen Server-Module, nur sind diese keine ausführbaren Programme, sondern Programmbibliotheken und werden direkt in den Webserver geladen.

*Weitverbreitete Open Source Datenbank, siehe [MyS07]
†Bei 1&1 neben MySQL hauptsächlich eingesetzte Datenbank, siehe [SYB07]

2.3 XP-Framework

Das XP-Framework [XPH06] (XP) ist der Hauptgrund weshalb - trotz der sehr mangelhaften Möglichkeiten zur Objektorientierung - bei 1&1 PHP 4 noch an vielen Stellen eingesetzt wird. XP ist ein modulares Framework, welches dem Anwender die Entwicklung jeglicher Anwendung in PHP erleichtern soll; es wird seit 2001 nach streng objektorientierten Kriterien und mittels eines agilen Prozesses hauptsächlich bei 1&1 hausintern entwickelt, ist aber komplett open source und wird unter der *BSD-Lizenz* * vertrieben. Eine zentrale Motivation für die Entwicklung des XP-Frameworks war die Überwindung der angesprochenen Defizite von PHP 4. Daher wurde unter anderem ein eigener Exception-Mechanismus, Methoden zum dynamischen Laden und zur Reflektion von Klassen, OO-Mapping von Datenbankzugriffen, sowie Abläufe zur sauberen Trennung von Logik und Präsentation entwickelt. Desweiteren enthält XP Äquivalente zu sehr vielen Java-Standardklassen. Durch sorgfältige, an Javadoc [JAV06b] angelehnte Dokumentation im Sourcecode wird versucht die mangelnde Typisierung in PHP zu überwinden, was Möglichkeiten zur automatischen Generierung von API-Dokumentation und zum Beispiel WSDLs† für SOAP eröffnet.

Alles in allem erlaubt das XP-Framework die Vorteile von PHP wie kurze Entwicklungszeiten und einfache Programmierung zu nutzen ohne auf die Vorteile der Objektorientierung zu verzichten. Viele dieser Funktionen setzen allerdings voraus, dass der ausführende PHP-Interpreter bestimmte Erweiterungen bereitstellt, ohne die beispielsweise das Transformieren von XML mittels XSL, die FTP- und Mailfunktionalität oder auch der Zugriff auf Datenbanken wie MySQL oder Sybase nicht möglich wären. An einer Portierung des Frameworks nach PHP 5 wird zwar schon länger gearbeitet, allerdings gestaltet sich dieses Vorhaben aufgrund der geänderten Verhaltensweisen der Programmiersprache bezüglich Referenzen sowie der von den Entwicklern der Zend-Engine etwas unbedacht eingeführten Exceptions schwierig. Desweiteren werden PHP ständig neue Standardklassen hinzugefügt was ob der fehlenden Namensräume zu Kollisionen mit Klassen des Frameworks führt. Es existieren fertige, PHP 5 kompatible Versionen aller Basisklassen. Zum Zeitpunkt der Erstellung dieses Dokumentes werden neue Anwendungen in PHP 5 entwickelt, viele ältere Applikationen laufen aber weiterhin in PHP 4 und müssen daher auch weiterhin gewartet werden.

2.4 SOAP

SOAP stand ursprünglich für **S***imple* **O***bject* **A***ccess* **P***rotocol*, heute ist SOAP nur noch ein Eigenname. SOAP ist ein Protokoll, welches den Austausch XML-basierter Nachrichten über ein Rechnernetzwerk - üblicherweise mittels HTTP - erlaubt. SOAP-Implementierungen sind heute für fast jede Programmiersprache verfügbar. SOAP

*Die BSD-Lizenz ist eine sehr liberale Open Source Lizenz, siehe [BSD06]

†WSDL ist eine Webservicebeschreibungssprache, siehe [Wee01]

wird ab PHP Version 5 inhärent unterstützt; auch innerhalb des XP-Frameworks stehen entsprechende Klassen zur Verfügung. SOAP entstand aus einer Zusammenarbeit von *Dave Winer* und *Microsoft*, im Jahre 2000 schlossen sich weitere Firmen, darunter IBM, der Entwicklungsgruppe an und reichten den Entwurf zu SOAP 1.1 beim *World Wide Web Consortium* (W3C) zur Weiterentwicklung ein, welches 2003 den aktuellen Standard SOAP 1.2 veröffentlichte. Eine SOAP Nachricht besteht lediglich aus einem *SOAP-Envelope* welcher wiederum ein (optionales) *Header*-Element und ein *Body-Element* enthäelt und verwendete Namesräume festlegt [Gro04]. Die eigentlichen Nutzdaten sind im Body-Element untergebracht und müssen vom Empfänger der Nachricht interpretiert werden, während im Header-Element Metadaten zur Nachricht enthalten sind, welche ebenfalls von den empfangenden Stationen interpretiert werden müssen. Diese sehr weit gefasste Spezifikation erlaubt es sehr flexible Anwendungen zu entwickeln, allerdings bringt SOAP auch einige Nachteile mit sich:
Aufgrund der Verwendung von XML zur Kodierung der zu übertragenden Informationen vervielfacht sich das Übertragungsvolumen deutlich was nicht nur Probleme mit der verwendeten Bandbreite verursacht, sondern auch erhebliche Mengen an Rechenleistung bei der Generierung und Verarbeitung der Nachrichten erfordert. Ein weiterer Kritikpunkt ist die mangelhafte und zum Teil sich selbst widersprechende Spezifikation selbst, was unter anderem dazu führt dass mittels verschiedener Software (*tool-chains*) erzeugte SOAP-Services und -Anwendungen nicht ohne händisches Eingreiffen des Entwicklers miteinander kommunizieren können. Um dieses Manko zu beheben wurde eigens die *Web Services Interoperability Group* * gegründet, eine Organisation, die versucht den Standard zu standardisieren. Allein Existenz einer solchen Organisation zeugt von der Mangelhaftigkeit der ursprünglichen SOAP-Spezifikation. Da die Spezifikation keine Aussagen über Transaktionalität trifft müsste solches Verhalten - falls gewünscht - von jeder Applikation selbst definiert und vom Benutzer implementiert werden, was dazu führt dass es in der Praxis unmöglich ist Anwendungen mittels SOAP zu verwirklichen welche Transaktionen benötigen würden. Auch ist nicht standardisiert wie dynamisch Daten über bei einem Server verfügbare Dienste erlangt werden können. Zu diesem Zweck wird meist ein *WSDL*-Dokument [Wee01] eingesetzt, allerdings wird auch an diesem Standard viel Kritik geübt, er beschreibt beispielsweise weder wie ein *WSDL*-Dokument für einen speziellen Dienst zu beziehen ist, noch enthält ein solches Dokument Informationen, die über die syntaktische Beschreibung des Dienstes hinausgehen. Ein weiteres SOAP-spezifisches Problem ist, dass kein Mechanismus definiert wird, über welchen Nachrichten aktiv an einen Teilnehmer übermittelt werden können, weshalb dieser ständig überprüfen muss, ob neue Daten für ihn vorliegen (“polling“).

*Die WS-I wird von vielen Unternehmen unterstützt die darauf angewiesen sind, dass Web-Services plattform- und programmiersprachenübergreifend funktionieren, weswegen ihre Vorschläge mittlerweile von weiten Teilen der Industrie als normgebend angesehen werden, siehe [WSI06]

2.5 XML-RPC

XML-RPC (vom englischen **R**emote **P**rocedure **C**all) ist ein Vorgänger von SOAP, und setzt - wie schon aus dem Namen hervorgeht - ebenfalls XML zur Kodierung der zu übertragenden Informationen ein. Im Gegensatz zu den meisten anderen Sytemen die den Aufruf entfernter Methoden erlauben ist XML-RPC sehr simpel gehalten und definiert nur einige wenige Datentypen. Diese sehr schlanke Spezifikation verhindert zwar, dass die zu übertragenden Nachrichten wie bei SOAP zu sehr aufgebläht werden, allerdings schränkt sie auch die mögliche Komplexität der zu versendenden Strukturen stark ein. XML-RPC eignet sich folglich gut um einfache Dienste anzubieten. Für komplexe Anwendungen hingegen würde man sich eher für eine fortgeschrittenere Technologie entscheiden. Nichtsdestotrotz sollte es an dieser Stelle erwähnt werden.

2.6 Java Enterprise Edition - Java EE

Java EE erweitert den Java Standard unter anderem um Funktionalität, die es dem Entwickler erlaubt verteilte, transaktionsbasierte und mehrschichtige (*multitier*) Anwendungen zu entwickeln (siehe auch [JEE06]). Hierzu spezifiziert Java EE mehrere Komponenten, wie unter anderem *Enterprise Java Beans (EJBs)*, *Servlets* und *Java Server Pages (JSPs)* und APIs sowie klare Schnittstellen zwischen diesen Komponenten. EJBs im besonderen sind standardisierte Komponenten innerhalb einer Java EE Architektur, mit denen verschiedene Dienste umgesetzt werden, die für die Geschäftslogik einer Anwendung notwendig sind. EJBs existieren in verschiedenen Ausprägungen, die verschiedene Klassen von Anwendungsfällen abdecken:

Session Bean - Session Beans bilden Vorgänge ab, insbesondere solche die der Nutzer an einem System durchführt. Session Beans implementieren immer ein Service-Interface, über dessen Methoden der Benutzer mit dem System interagieren kann. Man unterscheidet zustandslose (stateless) und zustandsbehaftete (stateful) Session Beans. Eine Stateful Session Bean kann - im Gegensatz zu einer Stateless Session Bean - Daten über mehrere Aufrufe hinweg speichern, wobei diese Daten immer einem bestimmten Nutzer zugeordnet bleiben.

Message Driven Bean - Message Driven Beans ermöglichen eine asynchrone Kommunikation innerhalb eines Java EE Systems, indem sie auf JMS* -Nachrichten, die in bestimmten Warteschlangen vorgehalten werden, reagieren.

Entity Bean - Entity Beans bilden die persistenten Daten, beispielsweise Felder einer Datenbank, innerhalb eines Systems ab und bieten Methoden diese Daten

*Java Message Service, API von Sun zum Austausch von Nachrichten zwischen zwei oder mehr Clients, siehe[JMS07]

aufzufinden, zu verändern und wieder abzuspeichern. In der neuesten EJB-Spezifikation* kommt Entity Beans eine weitaus weniger wichtige Bedeutung zu, da die Datenpersistenz nun mittels ganz normaler Java-Objekte (POJOs), deren Persistenzeigenschaften vom sogenannten *EntityManager* realisiert werden, abgebildet wird. Die Entity Bean als Datentransferobjekt wird somit nicht mehr benötigt.

Diese Technologien benötigen einen sogenannten *Java Application Server* welcher die nötige Infrastruktur für Sicherheit, Transaktionsmanagement, Kommunikation und vielem mehr unabhängig von Betriebssystem, Netzwerk- und Rechnerarchitektur bereitstellt. Der Application Server ist in mehrere Module (sogenannte *Container*) unterteilt, welche als Laufzeitumgebungen für einzelne Komponenten fungieren - so werden zum Beispiel EJBs im EJB-Container ausgeführt. Zum Aufruf von Methoden entfernter Objekte bietet Java RMI (**R**emote **M**ethod **I**nvocation) an, was entweder über ein eigenes Protokoll oder aber über IIOP gefahren werden kann. RMI ist ein binäres Protokoll, was den erzeugten Overhead im Vergleich mit SOAP erheblich reduziert. Ausserdem bietet RMI Transaktionssicherheit. RMI eignet sich deswegen besser für komplexe Anwendungen, bei denen auch die Antwortzeit eine Rolle spielt. RMI hat sich auch innerhalb der 1&1 als bevorzugtes Protokoll etabliert um Dienste miteinander zu verknüpfen. Leider existiert keine PHP-Implementierung, und die Komplexität und die binäre Natur des Protokolls machen eine Eigenentwicklung einer solchen Implementierung fast unmöglich.

2.7 Enterprise Application Server Connectivity - EASC

EASC ist eine hauptsächlich von Christian Gellweiler im Rahmen einer Diplomarbeit für 1&1 entwickelte "Protokollbrücke welche es PHP-Applikationen erlaubt RMI-Aufrufe mittels eines Stellvertreters auszuführen, welcher Daten und Anweisungen über ein generisches Protokoll erhält." [Gel05]
Dieser Proxy kommuniziert mit dem innerhalb des J2EE-Application-Servers ausgeführten sogenannten *EASC Service-MBean* mittels eines speziellen Protokolls, welches die Serialisierung und Übertragung von Daten und Objekten erlaubt. Die Erzeugung der Proxyklassen ("Stub") übernimmt ein spezielles Programm, welches es dem Entwickler erlaubt aus einer in Java geschriebenen Remote Interface Klasse des aufzurufenden Enterprise Beans eine äquivalente PHP-Klasse zu erzeugen. Wird innerhalb von PHP eine Methode des Stubs aufgerufen, so serialisiert dieser diesen Aufruf und übermittelt diesen mittels des EASC-Protokolls an die MBean. Die MBean deserialisiert diesen, und tätigt die seinerseits nötigen RMI Aufrufe. In der Gegenrichtung werden Rückgabewerte und Fehlermeldungen - wiederum über besagtes Protokoll - zurückgeliefert. EASC erlaubt es also einer komplett in PHP geschriebenen

*EJB 3.0 Specifications, siehe [EJB07]

Anwendung auf J2EE-Dienste zuzugreifen, welche andernfalls nur erreichbar wären würde diese Anwendung selbst in einer solcher Umgebung ausgeführt. Allerdings bietet EASC keine Möglichkeit selbst Dienste in PHP zu entwickeln, die ihrerseits aus einer J2EE-Umgebung mittels RMI aufgerufen werden können.

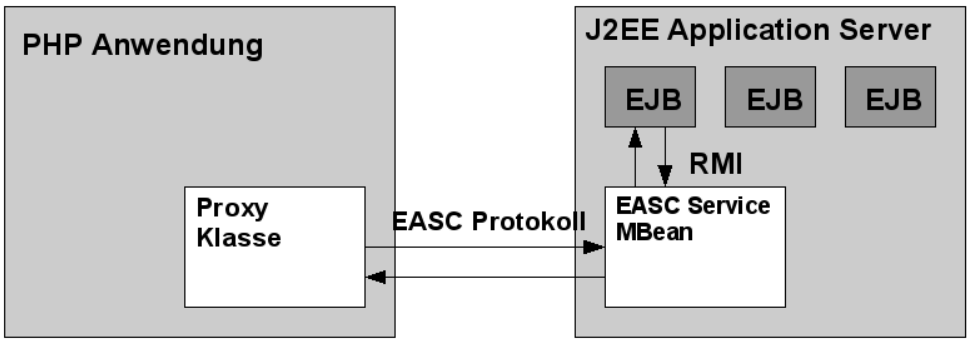

Abbildung 2.1: EASC-Funktionsweise - nach [Gel05]

2.8 php/Java bridge und die Java-Extension

Die php/Java bridge ist ein XML-basiertes Netzwerkprotokoll, das benutzt werden kann um einen PHP-Interpreter mit einer laufenden Java Virtual Machine zu verbinden. Die Java-Extension ist eine PHP-Extension* die dieses Protokoll benutzt um aus PHP heraus in der JVM Objekte zu erzeugen, und die dann Stubs dieser Objekte in PHP verfügbar macht, was es dem PHP-Anwender erlaubt auf diesen Objekten Methoden aufzurufen. Leider scheidet dieses Protokoll zur Implementierung der Aufgabe aus, erstens da es an den selben Nachteilen einer XML-Serialisierung krankt wie beispielsweise auch SOAP, aber vor allem da es - ähnlich wie EASC - den Zugriff von Java auf PHP nicht ermöglicht. Weiterhin ist die Java-Extension nicht mehr Teil des offiziellen PHP-Quelltextes, und wird auch nicht mehr gewartet, so ist es unter anderem nur sehr schwer möglich die Extension in PHP 5 einzubinden. Näheres zur php/Java bridge findet man unter [BRI06].

2.9 Java Native Interface - JNI

Java-Programme erreichen ihre Plattformunabhängigkeit vorrangig dadurch, dass sie eine virtuelle Maschine (*Java Virtual Machine - JVM*) vorraussetzen in welcher sie ausgeführt werden. Diese virtuelle Maschine verhindert allerdings zunächst, dass beispielsweise plattformspezifische Funktionen oder vorhandene, nicht in Java geschriebene Programmbibliotheken aus Java heraus angesprochen werden können. Um dies

*PHP-Erweiterung, siehe 2.2.1

zu umgehen bietet Java dem Entwickler mit dem JNI eine standardisierte Möglichkeit zur Laufzeit betriebssystemspezifische Bibliotheken (.dll unter Windows, .so unter unixoiden Betriebssystemen) einzubinden und Funktionen beziehungsweise Methoden, welche diese bereitstellen, aufzurufen. JNI ermöglicht nicht nur den Aufruf solcher Methoden aus Java heraus, sondern auch Aufrufe von Java-Methoden aus der eingebundenen Bibliothek heraus. Native Funktionen werden in einer eigenen .c oder .cpp (C oder C++) Datei implementiert, wobei C++ eine etwas sauberere Verbindung mit dem JNI erlaubt. Um aus Java heraus eine native Methode aufzurufen, muss diese im Javaquelltext als `native` deklariert werden. Aus der kompilierten Java-Klasse kann mittels des Werkzeuges *javah* eine Headerdatei erstellt werden, die die nötige Funktionsdeklaration enthält. Mit Hilfe dieser Headerdatei kann nun die native Programmbilbiothek entwickelt werden. In der Java-Anwendung muss vor dem Aufruf der nativen Methode die entsprechende Programmbibliothek mittels *System.loadLibray* geladen werden.

JNI wird unter anderem für performance-kritische Zwecke (zum Beispiel in Grafikanwendungen) eingesetzt. Allerdings setzen auch viele im Java-Standard enthaltene Klassen - wie beispielsweise jene, die für die Audioausgabe oder für Dateizugriffe zuständig sind - JNI Implementierungen vorraus um auf diese plattformspezifischen Funktionalitäten in einer sicheren und kontrollierten Art und Weise zuzugreifen. Leider sichert JNI den Anwender nicht gegen sämtliche möglichen Fehler ab: zwar kann beim Aufruf der Bibliotheksfunktionen kein Fehler mehr begangen werden, allerdings können sich Programmierfehler innerhalb der Bibliothek selbst weiterhin negativ auswirken, und zwar nicht nur auf den nativen Teil der Anwendung, sondern unter Umständen sogar auf die JVM selbst.

Java-Typ	Nativer Typ	Beschreibung
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits
void	void	N/A

Tabelle 2.1: Primitive Java-Datentypen und ihre JNI-Äquivalente, nach [JNI06]

JNI definiert für alle primitiven Java-Datentypen ein maschinenabhängiges Äquivalent, zum besseren Verständnis späterer Kapitel wurden in Tabelle 2.1 alle primiten Java-Typen und ihre JNI-Äquivalente aufgeführt. Komplexe Datentypen (auch: Referenztypen) werden auf C-Ebene als Pointer auf Structs repräsentiert, die ähnlich wie die Java-Klassen zueinander im Verhältnis stehen. Abbildung 2.2 zeigt die wichtigsten komplexen Datentypen, ihre JNI-Repräsentationen sowie deren Zusammenhänge.

Mehr Informationen zum JNI sind unter Anderem auf der JNI-Homepage ([JNI06]), und in dem Buch "Java Native Interface" von Sheng Liang ([Lia99]) zu finden.

Abbildung 2.2: Komplexe Java-Datentypen und ihre JNI-Äquivalente - nach [JNI06]

Kapitel 3

Java und Skriptsprachen

In diesem Kapitel wird zunächst besprochen, welche Möglichkeiten prinzipiell existieren um Skriptsprachen wie PHP innerhalb einer Java Virtual Machine auszuführen und um Daten zwischen den Laufzeitumgebungen der beiden Programmiersprachen auszutauschen. Dann werden zwei Technologien vorgestellt, die in diesem Kontext vorrangig benutzt werden, das *Bean Scripting Framework* und der *Java Specification Request 233*, und es wird erläutert welche sich besser für die Lösung der Aufgabe eignet. Außerdem wird erläutert auf welche Weise PHP in Java integriert werden soll und es wird ein simpler Prototyp entwickelt, um die Machbarkeit zu erweisen und um erste Erfahrungen mit der einzusetzenden Technologie zu erlangen. Schlussendlich wird das Projekt in drei Unterprojekte unterteilt und ein Zeitrahmen für die einzelnen Schritte festgelegt.

3.1 Herangehensweisen

Beim Erweitern von Java um die Fähigkeit Skriptsprachen auszuführen exisiteren zwei wesentliche Anwendungsfälle:

Java Objekte in Skriptsprachen: Hierbei werden Java-Klassen benutzt um das Leistungsvermögen einer Skriptsprache zu erweitern. Die Skriptsprache soll fähig sein Java Objekte zu instanziieren und auf deren öffentliche (public) Methoden und Variablen zuzugreiffen. Beispiele für eine solche Integration von Java und einer Skriptsprache sind *Live Connect* von Netscape und proprietäre APIs der Microsoft Java Implementierungen.

Skripte in Java: Java Applikationen soll es möglich sein existierende Skripte auszuführen, diesen Skripten Eingabedaten zur Verfügung zu stellen, deren Rückgabewerte und innerhalb der Skripte vorhandene Daten auszulesen, und gezielt auf Funktionen und Methoden eines laufenden Skriptes zuzugreiffen.

Diese beiden Use-Cases enthalten sich zum Teil gegenseitig und teilen sich viele Anforderungen, um sie zu erfüllen existieren mehrere Herangehensweisen:

Java-Interpreter: Für einige Skriptsprachen sind Interpreter erhältlich, die komplett in Java implementiert sind und welche meist über externe Schnittstellen verfügen, was es einer Java-Applikation erlaubt diese Interpreter einzubetten und Skripte der entsprechenden Sprache auszuführen. Leider folgen diese Schnittstellen keinem gemeinsamen Standard, auch weil zum Entwicklungszeitpunkt dieser Interpreter oftmals kein solcher Standard existierte. Beispiele für diese Art der Einbettung sind unter anderem die Javascript-Implementierung der Mozilla Foundation *Rhino*[*] und *Jacl*[†], ein vollständig in Java geschriebener Interpreter für TCL.

Übersetzung von Skriptcode nach Java Sourcecode: Einige Skriptsprachen verfügen über Sprachkonstrukte und eine Syntax welche stark an Java angelehnt sind, daher ist es möglich Sourcecode solcher Sprachen in Java-Sourcecode zu übersetzen. Allerdings ist das Entwickeln eines solchen Transcoders äußerst aufwändig, so dass es meist einfacher ist Skripte solcher Sprachen manuell nach Java zu portieren. Beispiele solcher Sprachen sind die BeanShell[‡] und Groovy[§].

Übersetzung von Skriptcode nach Java Bytecode: Es existieren einige Compiler die versuchen Skriptcode direkt in Java Bytecode zu übersetzen. Die so generierten Klassen können direkt von der JVM ausgeführt werden. Die Entwicklung eines solchen Compilers ist zwar ähnlich aufwändig wie die Entwicklung eines oben vorgestellten Transcoders, allerdings kann die JVM vom Compiler einfach wie eine weitere Hardwareplatform behandelt werden, und da viele Skriptsprachen schon ausführbaren Code erzeugen können muss nur noch die Codegenerierung ausgetauscht werden, alle anderen Teile des Compilers, wie syntaktische und semantische Analyse, können unverändert weiterbenutzt werden. Ein Beispiel eines solchen Compilers ist *Jython*[¶] für die Skriptsprache Python.

Eingebettete, native Interpreter: Die wohl einfachste Möglichkeit eine Skriptsprache in Java einzubetten ist, diese als native Bibliothek mittels des JNI (siehe **??**) einzubinden und mit einem Adapter zu umhüllen (engl. *wrapping*). Ein solcher Wrapper wird auch *language binding* genannt.

Eine Softwarekomponente, die das Ausführen von Skripten einer bestimmten Sprache erlaubt, und dabei ein standardisiertes Interface implementiert und somit zumindest in

[*] siehe [Boy07]
[†] siehe [JAC07]
[‡] siehe [BEA07]
[§] siehe [GRO07]
[¶] siehe [JYT07]

der Theorie austauschbar ist, soll im weiteren Verlauf dieses Dokumentes *Skriptengine* genannt werden.

Es existieren zwar Ansätze PHP-Interpreter in Java zu implementieren, ein prominentes Beispiel ist *Quercus* von *Caucho Technology* [CAU06]. Allerdings gestaltet sich dieses Vorhaben durch die grosse Menge an PHP "built-in" Funktionen, Funktionen also die nicht mittels PHP verwirklich wurden, sondern die direkt vom Interpreter erkannt und ausgeführt werden, als sehr schwierig. Diese Standard-Bibliothek wird auch mit jeder PHP-Version umfangreicher und unterliegt ständigen Veränderungen. Desweiteren enthalten solche Interpreter in aller Regel keine Möglichkeit PHP-Extensions zu benutzen. Folglich schied diese Art der Integration von PHP in Java aus, und es musste eine Methode gefunden werden den vorhandenen, originalen PHP-Interpreter aus Java heraus anzusprechen.

Um PHP in anderen Umgebungen einzubetten bietet es dem Benutzer die Möglichkeit als dynamisch ladbare Bibliothek (.dll unter Windows, .so unter unixoiden Betriebssytemen) kompiliert zu werden. Java wiederum erlaubt das Einbinden solcher Bibliotheken zur Laufzeit mittels des JNI (siehe 2.9). Um auf den PHP-Interpreter derart zugreifen zu können, muss eine sogenannte *SAPI* implementiert werden. SAPI steht in diesem Kontext für "Server Application Programming Interface" und ist ein PHP-interner Begriff. Der naive Ansatz PHP innerhalb von Java auszuführen wäre nun das simple Einbinden dieser Bibliothek und der Zugriff auf Selbige über JNI gewesen. Hierzu wäre es aber nötig gewesen selbst eine Schnittstelle zu entwickeln die diesen Zugriff formalisiert, allerdings existierten schon zwei unabhängig voneinander entwickelte Methoden Skriptsprachen innerhalb Javas zu verwenden - das *Bean Scripting Framework* des *Apache Jakarta Project* und ein sogenannter *Java Specification Request*. So musste evaluiert werden, ob nicht eine dieser beiden Technologien ein geeigneter Weg wäre dieses Ziel zu erreichen.

3.2 Bean Scripting Framework - BSF

Die aktuelle Version 2.4 des Bean Scripting Frameworks - ursprünglich von IBM entwickelt und nun Teil des Apache Jakarta Projektes [BSF06] - bietet eine API, welche es einer Java-Applikation erlaubt Skriptsprachen einzubinden und die den derart ausgeführten Skripten den Zugriff auf Java-Objekte der ausführenden Applikation ermöglicht. Für jede auszuführende Skriptsprache wird eine Implementierung des Interfaces `BSFEngine` benötigt, welches eine Abstraktion der Fähigkeiten der Skriptsprache darstellt und so der Applikation ein generisches Zugriffsschema auf jegliche Skriptsprache bietet. Alle Instanzen von BSFEngines werden von einer einzigen Instanz der Klasse `BSFManager` verwaltet. Zusätzlich hält dieser ein Register von Objekten (die *object registry*) vor, auf welche der Zugriff aus den laufenden Skripten heraus möglich sein soll. Der BSFManager hält so lange er existiert den Ausführungszustand aller bei ihm registrierten BSFEngines vor, was das Anhalten und spätere Fortsetzen

eines Skriptes erlaubt. Das BSF ist in so gut wie allen Jakarta-Produkten wie Ant, Xalan, und Tomcat schon enthalten - so ist es beispielsweise möglich unter Tomcat JSPs in vom BSF unterstützten Skriptsprachen zu formulieren. Zu diesem Zeitpunkt existieren BSFEngine-Implementierung für eine Vielzahl von Skriptsprachen, darunter Javascript/ECMAScript, Python, Ruby und XSLT, allerdings keine Implementierung für PHP. Die Dokumentation des BSF ist sehr knapp gehalten, allerdings beinhaltet sie alle nötigen Informationen über Architektur und Funktionsweise des Frameworks, ausserdem sind alle Teile des Frameworks unter einer Open Source Lizenz im Quelltext verfügbar.

Eine BSFEngine muss folgende Methoden implementieren:

- `initialize()` - wird zu Begin der Lebenszeit der Skriptengine aufgerufen, um diese zu initialisieren.
- `call()` - zum Aufruf bestimmter Funktionen oder Methoden innerhalb des in der `BSFEngine` geladenen Skriptes.
- `eval()` - mittels `eval()` wird ein als String übergebener Ausdruck ausgewertet und ein Rückgabewert zurückgeliefert.
- `exec()` - mittels `exec()` wird ein komplettes Skript ausgeführt.
- `declareBean()` - mit dieser Methode wird explizit eine Instanz einer bestimmten Klasse innerhalb der `BSFEngine` erzeugt.
- `undeclareBean()` - wird genutzt, um ein vorher mittels `declareBean()` erzeugtes Objekt aus der Engine zu entfernen.

Nachdem dieser Wrapper für die gewünschte Sprache implementiert ist muss mittels der Methode `registerScriptingEngine()` die neu erstellte `BSFEngine` beim `BSFManager` registriert werden. Fortan können Skripte dieser Sprache mittels der normalen BSF-API ausgeführt werden, hierzu bietet der `BSFManager` neben vielen Methoden, die auch in einer `BSFEngine` zu finden sind, unter anderem folgende Methoden:

loadScriptingEngine() - versucht eine BSFEngine für eine gewünschte Sprache zu erstellen.

registerBean() - fügt der object registry ein Objekt hinzu.

lookupBean() - gibt ein Objekt aus der object registry zurück.

Außerdem verfügt der `BSFManager` noch über Methoden, mit denen es möglich ist, Qeulltext-Stücke und ganze Skripte in sogenannte `CodeBuffer`-Objekte zu übersetzen und erst später auszuführen. Das BSF deklariert automatisch innerhalb des ausgeführten Skripts ein Objekt namens *BSF*, welches den mit der BSFEngine assoziierten BSFManager repräsentiert. Über dieses Objekt kann aus dem Skript auf Daten und Objekte aus der Java-Anwendung zugegriffen werden.

3.3 JSR 223 - Scripting for the Java Platform

Ein **J**ava **S**pecification **R**equest ist Teil des Java Community Process (JCP), stellt eine Anforderung den Java Standard (**J**ava **L**anguage **S**pecification - JLS) zu erweitern dar und wird normalerweise von einem Expertenteam geleitet.

Der JSR mit der Nummer 223 [ea06a] beschreibt - ähnlich dem BSF - Mechanismen, welche es Skripten erlauben sollen auf Informationen innerhalb einer Java-Applikation zuzugreiffen, sowie serverseitige Java-Applikationen dazu befähigen sollen Skriptsprachen für Webseiten einzusetzen (*web scripting*). Ursprünglich wurde JSR 223 zwar explizit für diesen serverseitigen Einsatz von Skriptsprachen konzipiert, allerdings wird diese Einschränkung in der neuesten Version (Proposed Final Draft, 10. August 2006) relativiert und auch der Einsatz in anderen Anwendungsgebieten berücksichtigt. In diesem Dokument werden einige Begrifflichkeiten definiert:

Als *scripting engine* wird eine Softwarekomponente bezeichnet, welche in einer Skriptsprache geschriebene Programme ausführt. Eine scripting engine beinhaltet normalerweise einen *interpreter* der die eigentliche Ausführung des Programmes übernimmt und wiederum aus einem *front-end* und einem *back-end* besteht. Das front-end ist für die lexikalische und syntaktische Analyse des Quelltextes zuständig und überführt diesen in eine Zwischenform, den sogenannten *intermediate code*. Das back-end führt diesen intermediate code aus und nutzt Symboltabellen (*symbol table*) um Variablen innerhalb des Skriptes zu speichern. Die Spezifikation geht davon aus, dass alle diese Softwarekomponenten Java-Objekte sind, wobei explizit erlaubt wird, dass einige dieser Objekte native Aufrufe nutzen. Scripting engines die die *General Scripting API* implementieren werden als *Java Script Engines* bezeichnet, eine ob der Existenz einer Programmiersprache gleichen Namens etwas unglücklich gewählten Bezeichnung.

Die JSR 223-Spezifikation unterteilt sich in zwei große Abschnitte: Zunächst werden *Java Language Bindings* besprochen, Mechanismen die es Skripten erlauben Java-Klassen zu laden, Objekte zu erzeugen und Methoden dieser Objekte aufzurufen. Im selben Kapitel wird auch beschrieben wie Methodenaufrufe in Skriptsprachen in Java-Methodenaufrufe umgesetzt, und wie die Argumente und Rückgabewerte dieser Aufrufe zwischen den Laufzeitumgebungen der beiden Sprachen hin- und hergereicht werden. Dieser Teil der Spezifikation ist allerdings nicht normgebend, da es sehr von den Eigenschaften der Skriptsprache abhängt, wie solche Sprachbindungen realisiert werden müssen. Es werden drei Arten von Language Bindings besprochen: *Dynamic Bindings* werden zur Laufzeit des Skriptes erzeugt, als *Programmatic Bindings* werden Sprachbindungen bezeichnet die etwa über Stellvertreterobjekte innerhalb des Skriptes den Zugriff auf Objekte der Java-Anwendung erlauben. Die dritte Art von Sprachbindungen werden *Static Bindings* genannt und beziehen sich auf Funktionen welche die Skriptengine selbst direkt auf Funktionen innerhalb von Java abbildet. Beispielsweise könnte der PHP-Befehl `echo()` direkt einen Aufruf der Java-Methode `System.out.println()` bewirken. Der zweite große Abschnitt beschäftigt sich mit

der *General Scripting API*, einer Sammlung von Interfaces und Klassen, welche es erlauben, dass Skriptengines als Komponenten in Java-Anwendungen eingebettet werden können. Diese API unterteilt sich in einen implementierungsabhängigen und einen implementierungsunabhängigen Teil um eine größtmögliche Portabilität zwischen verschiedenen Implementierungen zu erreichen. Hierzu ist es notwendig zur Laufzeit Informationen über Skriptengines abfragen zu können, diese *Metainformationen* beschreiben Fähigkeiten und Funktionsdetails und werden von jeder Implementierung bereitgestellt. Dieser Mechanismus zum Auffinden und zum Abfragen von ScriptEngineFactories zur Laufzeit wird im Spezifikationsdokument als *discovery mechanism* bezeichnet. Die wichtigsten Klassen der General Scripting API sind der `ScriptContext`, die `ScriptEngine`, die `ScriptEngineFactory` und der `ScriptEngineManager`.

ScriptContext - der ScriptContext wird benutzt um einem Skript den Zugriff (engl. *view*) auf Daten der Wirtsapplikation zu erlauben. Dies geschieht mittels Schlüssel/Wertepaaren, die verschiedenen Gültigkeitsbereichen (engl. *scope*) zugeordnet sind. Die Scopes unterscheiden sich in Sichtbarkeit und Bedeutung. Der ScriptContext bietet den `GlobalScope`, auf welchen von jeder ScriptEngine aus zugegriffen werden kann, die den jeweiligen ScriptContext benutzt, und welcher den Zustand der Hostapplikation darstellt. Außerdem bietet der ScriptContext den `EngineScope`, der nur für eine einzige ScriptEngine gültig ist und vorwiegend dazu genutzt wird Variablen innerhalb dieser auszulesen. Der Anwender kann jederzeit weitere Scopes definieren, sie werden durch Integerwerte identifiziert. Mittels der Methode `getScopes()` kann der Anwender eine Liste der bekannten Scopes erhalten. Innerhalb der einzelnen Scopes können Werte mittels der Methoden des ScriptContext `setAttribute()`, `getAttribute()` und `removeAttribute()` gesetzt, beziehungsweise abgerufen und gelöscht werden, dazu muss jeweils der Schlüssel als String übergeben werden.

Binding - auf Scopes kann mittels des Interfaces `Bindings` zugegriffen werden. `Bindings` implementiert das Interface `Map`, mit der zusätzlichen Anforderung, dass alle Schlüssel nicht leer und nicht `null` sind.

ScriptEngine - das ScriptEngine-Interface ist eine Abstraktion eines Skriptinterpreters und muss von allen Java Script Engines implementiert werden. Es beinhaltet Methoden um Skripte auszuführen und diesen Schlüssel/Wertepaare zu übergeben. Optional kann eine Skriptengine dem Anwender auch die Möglichkeit bieten Skriptquelltexte in intermediate Code zu übersetzen und diesen dann wiederholt auszuführen, was allerdings vorraussetzt dass die Skriptengine explizit den Zugriff auf Front- und Back-end erlaubt. Jede ScriptEngine hat einen Standardkontext, der mittels der Methoden `getContext()` und `setContext()` manipuliert werden kann. Anstatt den Umweg über die Scopes zu gehen kann der Anwender einer ScriptEngine Werte auch direkt mittels

der Methoden `get()` und `put()` übergeben, welche sich auf den EngineScope auswirken. Die eigentliche Ausführung eines Skriptes kann mittels der verschiedenen `eval()` Methoden bewerkstelligt werden, die sich meist nur in der Art und Weise der Quelltextübergabe unterscheiden.

Compilable - Compilable ist ein Interface, das optional von einer ScriptEngine implementiert werden kann, wenn sie in der Lage ist intermediate Code zu produzieren. Eine solche ScriptEngine bietet die Methode `compile()` an, welche eine Instanz von `CompiledScript` zurückgibt. Ein CompiledScript kann von der ScriptEngine wieder mittels `eval()` ausgeführt werden. Es wird davon ausgegangen, dass ein Aufruf von `eval()` das `CompiledScript` nicht verändert, und somit dass mehrere Aufrufe dieser Methode das gleiche Ergebnis liefern.

Invocable - Invocable ist wie Compilable ein weiteres optionales Interface welches von einer ScriptEngine implementiert werden kann. Invocable ScriptEngines bieten dem Anwender die Möglichkeit Prozeduren, Funktionen und Methoden eines Skriptes direkt aufzurufen, deren Aufruf mittels der Methoden `invokeFunction()` und `invokeMethod()` geschieht. Die beiden Ausprägungen der Methode `getInterface()` liefert eine Instanz einer Java-Klasse zurück, welche in der ScriptEngine mittels der Skriptsprache implementiert ist.

ScriptEngineFactory - Die ScriptEngineFactory kann genutzt werden um eine benötigte ScriptEngine zu erstellen, für jede ScriptEngine-Implementierung muss eine korrespondierende Implementierung der ScriptEngineFactory vorhanden sein. Hierzu enthält sie Methoden um auf die oben beschriebenen Metainformationen zuzugreifen. Zusätzlich bietet eine ScriptEngineFactory die Methoden `getOutputStatement()`, `getMethodCallSyntax()` und `getProgram()` an, mit deren Hilfe man Strings erzeugen kann, die von der zugehörigen ScriptEngine als Skript ausgeführt werden können.

ScriptEngineManager - der ScriptEngineManager hilft dem Anwender ScriptEngineFactories für alle verfügbaren ScriptEngines zu finden. Hierzu bietet er die Methoden `getEngineByExtension()`, `getEngineByMimeType()` und `getEngineByName()`. Mittels analoger `register...()`-Methoden kann dieser Auffindungsprozess zur Laufzeit angepasst werden. Zusätzlich verwaltet er - ähnlich wie in der BSF der BSFManager - die Kontexte der ihm unterstellten Skriptengines.

Von der Firma Zend existiert eine JSR 223 Referenzimplementation für PHP5, allerdings ist diese nicht im Quelltext verfügbar, und das mitgelieferte PHP erfüllt wegen fehlender Extensions nicht ganz die Ansprüche die die Aufgabe stellt. Der JSR 223 ist Teil der *Java Language Specification 6* und ist somit in Java 1.6 enthalten, was seit Dezember 2006 erhältlich ist.

3.4 Ergebnis der Analyse

Sowohl das Bean Scripting Framework, als auch der JSR 223 erwiesen sich als geeignete Technologien, um die gestellte Aufgabe zu erfüllen. Allerdings bot der JSR 223 aus Sicht des Autors einige Vorteile: Die verfügbare Dokumentation ist - ganz im Gegensatz zur Dokumentation des BSF - sehr umfangreich, vor allem geht sie auch auf Details wie Methodenaufrufe ein. Die API stellt sich auch etwas kompletter dar, die Interfaces Invocable und Compilable helfen Implementierungsunterschiede sauber zu überwinden. Auch die Tatsache dass der JSR 223 aller Vorraussicht nach Teil des Java Standards werden wird sprach, gepaart mit der Unterstützung durch namhafte Firmen wie Sun, IBM und vor allem auch Zend, eher für als gegen diese Technologie. Die Existenz einer Referenzimplementation für PHP erlaubt nicht nur das schnelle Entwickeln eines Prototypen, sondern bewies eben auch, dass die Vorgehensweisen welche der JSR 223 vorschlägt gangbar und eine Lösung der gestellten Probleme zumindest im Bereich des Möglichen lagen, während alle verfügbaren Skriptengines für das BSF komplett in Java implementiert waren. Somit wurde vom Autor gemeinsam mit den Betreuern entschieden dem JSR 223 den Vorzug zu gewähren.

3.5 Prototyp

Nachdem die Entscheidung über die zu verwendende Technologie getroffen wurde musste diese validiert werden. Hierzu wurde eine Machbarkeitsstudie mit der von Zend zu Verfügung gestellten, und kostenlos herunterladbaren Referenzimplementierung für PHP durchgeführt. Es sollte ein einfaches PHP-Skript ausgeführt werden, welches trotzdem eine gute Abschätzung des Funktionsumfanges des mitgelieferten PHPs erlauben sollte. Um allerdings sicherzustellen, dass zumindest der Implementierungsunabhängige Teil funktioniert wurde ein Programm geschrieben, welches lediglich eine Liste der verfügbaren Skriptengines ausgibt. Hierbei wurde nun festgestellt, dass sich die Referenzimplementierung nicht komplett an den Standard hält - so lautet die Signatur der Methode `getScriptEngineFactories()` des ScriptEngineManagers beispielsweise nicht wie von der Spezifikation verlangt

```
List<ScriptEngineFactory> getScriptEngineFactories()
```

Listing 3.1: Vom Standard verlangte Signatur

sondern

```
ScriptEngineFactory[] getScriptEngineFactories()
```

Listing 3.2: Signatur in der Referenzimplementierung

Dieser erste Eindruck wurde durch weitere Fakten bestätigt, was letztendlich dazu führte, dass die weitere Entwicklung unter Zuhilfename der Referenzimplementation nicht fortgesetzt wurde: Als Test für den Implementierungsabhängigen Teil

wurde die PHP-Funktion `phpinfo()` ausgewählt, welche Text ausgibt, aus dem sich alle geladenen Extensions, sowie alle gesetzten Umgebungsvariablen und PHP-Versionsinformationen ersehen lassen. Leider war es dem Autor unmöglich diesen Protoypen mittels der Zend-Implementierung* zu verwirklichen, da das Laden der mitgelieferten php5.so sowohl und FreeBSD, als auch unter 64- und 32-Bit Linux mit verschiedensten Java-Versionen zum Absturz der Java Virtual Machine führte. Die Tatsache dass die Referenzimplementierung nicht im Quelltext verfügbar ist verhinderte gleichzeitig die Neuübersetzung oder eine effektive Fehlersuche, weiterhin bedeutete dies, dass nur sehr mangelhaft Kontrolle über die benutzte PHP-Version ausgeübt werden konnte.

Weiterhin existiert neben der Referenzimplementierung eine PHP-Extension namens *php/Java bridge*, die das Erzeugen von Java-Objekten und das Aufrufen derer Methoden aus PHP heraus erlaubt, siehe hierzu 2.8. Allerdings fehlt dieser Extension erstens die Möglichkeit aus Java heraus gezielt PHP-Funktionen und -Methoden aufzurufen, ausserdem verwendet sie nicht das JNI zur Kommunikation zwischen den Laufzeitumgebungen der beiden Sprachen, sondern nutzt ein spezielles XML-Protokoll um jeglichen Aufruf und jeden Parameter zu serialisieren und über lokale Sockets an die JVM zu senden. Diese Herangehensweise leidet unter ähnlichen Problemen wie alle Plattform-agnostischen Protokolle zum Aufruf entfernter Methoden im allgemeinen und XML-basierter Protokolle wie beispielsweise SOAP und XML-RPC im speziellen, siehe hierzu 2.4.

Aus diesen Gründen wurde beschlossen eine eigene JSR 223-Implementierung für PHP zu entwickeln.

3.6 Projektplan

Zu diesem Zeitpunkt waren die zur Durchführung des Projekts nötigen Vorarbeiten abgeschlossen, alle nötigen Informationen waren gesammelt und die verfügbaren Technologien waren evaluiert. Nun sollte ein erster, grober Projektplan erstellt werden, der das Projekt in seine Teilbereiche unterteilt. Aufgrund dieser Analyse und der Erfahrungen mit dem Protoypen wurde das Projekt in drei große Teilbereiche aufgeteilt:

Zuerst sollte die Ausführung von PHP-Quelltext innerhalb einer Java Virtual Machine ermöglicht werden. Hierzu sollte mittels einer eigenen Implementierung des JSR 223 ein möglichst austauschbares PHP verwendet werden können. Ziel sollte hierbei insbesondere die Ausführung eines möglichst großen Teils des XP-Frameworks sein. Als Ergebnis sollte am Ende ein einfach zu benutzendes Java-Archiv, zusammen mit einigen Funktionstests entstehen, das dann in weiteren Teilprojekten eingesetzt werden kann. Fuer diesen Teil des Projektes wurden sechs Wochen veranschlagt.

*bezogen über [ea06b]

Im zweiten Teilprojekt sollte der Einsatz von PHP innerhalb eines Java Application Servers - beispielsweise JBoss - erforscht und vereinfacht werden. Großen Wert sollte hierbei auf ein möglichst einfaches Ausbringen (*deployment*) derart entwickelter Anwendungen gelegt werden. Hierzu sollte auch die Notwendigkeit Java-Quelltext zu schreiben auf ein Minumum reduziert, oder wenn möglich komplett eliminiert werden. Hierfür wurden weitere zwei Wochen eingeplant.

Vier Wochen sollten darauf verwendet werden möglichst viele Eigenschaften des Java EE Application Servers für die ausgeführten PHP-Skripte erreichbar zu machen. Dazu zählen die Persistenzanbindung, beispielsweise mittels Java Persistance und besonders die Anbindung an JMI, zum Beispiel als Message Driven Bean.

Schlussendlich sollte die restliche Zeit genutz werden um die Ausarbeitung fertigzuschreiben, und um Dokumentation über die Bibliothek und das Aufsetzen einer Beispielumgebung zu erstellen. Diese Zeit war auch als Puffer für eventuell auftretende Probleme zu verstehen.

Im Laufe der Entwicklung der JSR 223 Implementierung wurde jedoch festgestellt, dass zusätzlich zum reinen Ausführen von PHP-Skripten innerhalb einer Java-VM noch einiges weitere an Funktionalität zu realisieren war. Zum einen sollte aus dem PHP-Skript heraus das Erzeugen von Java-Objekten und der Zugriff auf deren Methoden und Attribute möglich sein, und zum anderen sollte auch umgekehrt aus Java heraus ein Zugriff auf Funktionen und Methoden innerhalb des ausgeführten PHP-Skriptes ermöglicht werden. Aus diesem Grund wurde der Projektplan derart umgestellt, dass der JSR 223-Implementierung mehr Zeit eingeräumt werden konnte. Hierzu wurden die Forderungen nach einem Buildsystem für PHP-EJBs komplett gestrichen, und der JBoss- mit dem eigentlichen EJB-Teil zusammengelegt. Dies ließ acht Wochen Zeit um die JSR 223 Implementierung zu erstellen, und vier Wochen um diese in einem JBoss anzuwenden um Session- und Message Driven Beans in PHP zu schreiben und auszuführen. Abbildung 3.1 zeichnet die Unterteilung des ursprünglichen und des tatsächlichen Projektplanes über die Zeit auf.

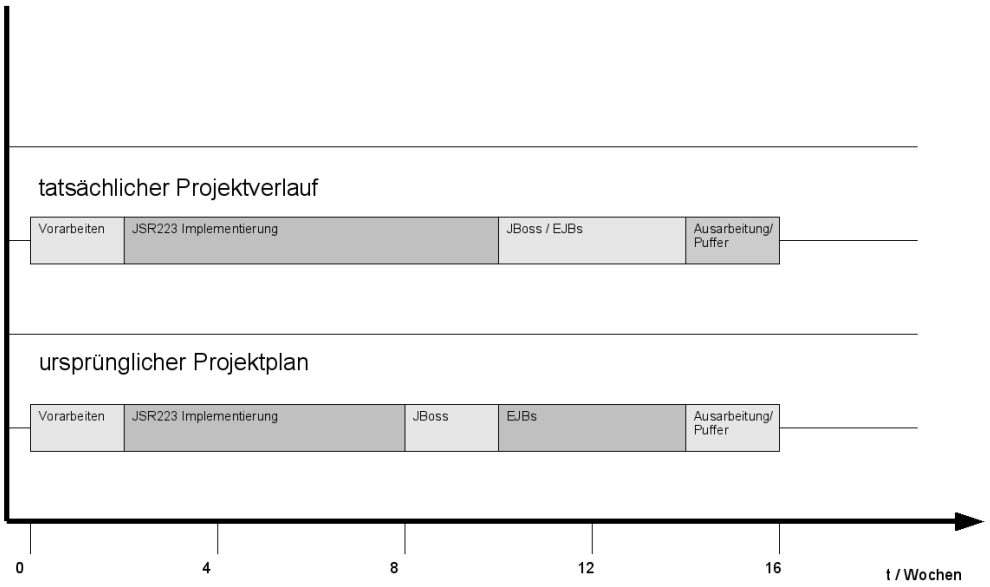

Abbildung 3.1: Projektplan

Kapitel 4

Java und PHP

In diesem Kapitel soll eine JSR 223-Implementierung für PHP entworfen und erarbeitet werden. Hierzu wird zunächst die Aufgabe analysiert und die von ihr gestellten Anforderungen erörtert. Diese Anforderungen werden dann genutzt, um das Vorgehen bei der Implementierung zu planen und um eine Architektur zu entwerfen, die diese Anforderungen möglichst komplett erfüllt. Schließlich werden die gewonnenen Erkenntnisse genutzt, um die erarbeitete Vorgehensweise in die Tat umzusetzen. Es wird eine Bibliothek zu erstellt, die es dem Anwender erlaubt, PHP-Skripte in Java auszuführen, Daten an diese Skripte zu senden, aus Java heraus PHP-Funktionen aufzurufen und aus diesen Skripten heraus Java-Funktionalität zu nutzen.

4.1 Analyse

In diesem Abschnitt wird erläutert, wie der erste Teil der Aufgabenstellung analysiert wurde, welche Use-Cases dabei gefunden wurden und wie aus diesen Use-Cases Arbeitspakete abgeleitet wurden. Dieser erste Teil umfasste hauptsächlich das Ausführen von PHP-Sourcecode innerhalb einer Java-Anwendung. Die Anforderung war, möglichst große Teile des XP-Frameworks innerhalb von Java auszuführen. Vom vollen Umfang des XP-Frameworks ausgenommen waren lediglich die Komponenten welche eine Kommunikation nach außen erlauben, so zum Beispiel die Datenbankkonnektivität, da diese Funktionen später vom Application Server bereitgestellt werden sollten. Idealerweise sollte die verwendete PHP-Version leicht austauschbar sein. Eine weitere Anforderung war, die Interaktion von Java nach PHP und umgekehrt möglichst einfach zu gestalten - Java-Objekte sollen in PHP erzeug- und zugreifbar sein.

4.1.1 Use-Cases

Use-Cases (auch: Anwendungsfälle) definieren die Interaktion zwischen Akteuren und dem zu entwerfenden Softwaresystem. Das Erstellen der Use-Cases soll zu einem besseren Verständnis nicht nur der zu implementierenden Abläufe, sondern der gesamten Aufgabe führen. Zunächst mussten allerdings die möglichen Akteure ermittelt werden. Es ergab sich, dass lediglich zwei unterschiedliche Akteure existieren: der Java-Anwender, der eine Java-Applikation benutzt beziehungsweise entwickelt und den PHP-Anwender, dessen PHP-Skript aus Java heraus ausgeführt wird. Auf oberster Ebene wollen diese beiden Akteure Daten austauschen, beziehungsweise auf Eingaben des jeweils anderen reagieren um Ergebnisse zurückzuliefern, bei genauerer Betrachtung ließen sich im Wesentlichen vier Use-Cases aus der Aufgabe ableiten:

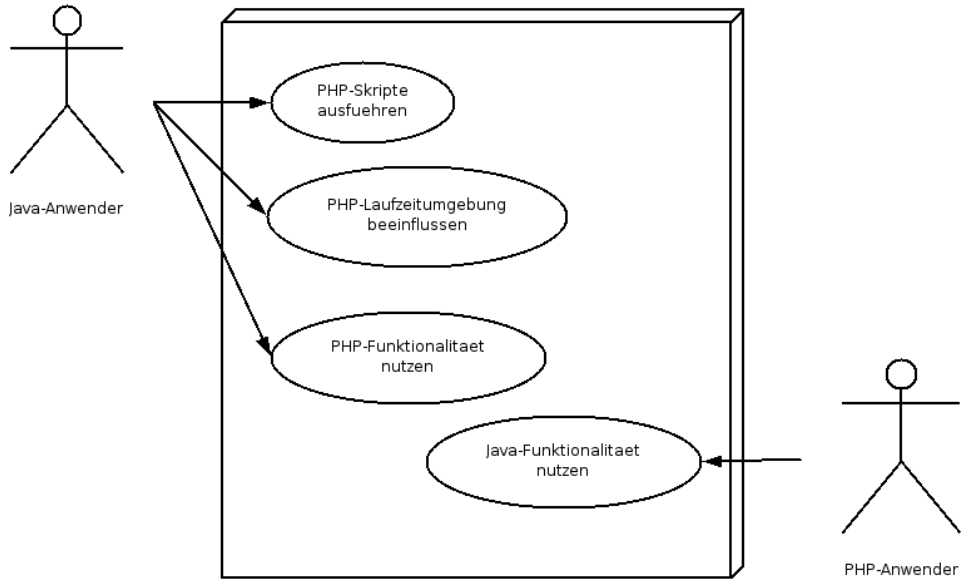

Abbildung 4.1: Use-Cases

UC.01 - Ausführen von PHP-Skripten Ein Java-Anwender möchte ein oder mehrere PHP-Skripte aus einer Java-Applikation heraus ausführen. Hierbei soll es möglich sein das Skript zur Laufzeit der Java-Applikation zu verändern und erneut auszuführen. Weiterhin sollen Daten sowohl aus Java an das PHP-Skript übergeben, als auch Ergebnisse des Skriptes an Java zurückübergeben werden.

UC.02 - Beeinflussen der PHP Laufzeitumgebung Dem Java-Anwender soll möglich sein Einfluss auf die PHP-Laufzeitumgebung zu nehmen. Dies geschieht bei PHP typischerweise durch das Setzen von Initialisierungsparametern, entweder systemweit in der Datei **php.ini**, beim Starten des PHP-Interpreters, oder zur Laufzeit des PHP-Skriptes.

UC.03 - Zugriff auf Java aus PHP Dem Entwickler eines PHP-Skriptes soll Zugriff auf möglichst den kompletten Funktionalitätsumfang der Java-Umgebung gewährt werden. Das umfasst vor allem das Instanziieren von Java-Objekten und den Zugriff auf deren Methoden und Attribute, aber auch weitere Sprachfunktionen wie beispielsweise das Werfen und Fangen von Exceptions. Zusätzlich soll der Zugriff auf Java für den PHP-Anwender möglichst intuitiv ablaufen, im Idealfall sollen sich Java-Objekte in PHP genauso behandeln lassen wie native PHP-Objekte.

UC.04 - Zugriff auf PHP aus Java Es soll einem Java-Entwickler die Möglichkeit eines einfachen Zugriffs auf innerhalb eines PHP-Skriptes vorhandene Funktionen, Klassen und deren Methoden gewährt werden. Auch hier soll dieser Zugriff möglichst intuitiv erfolgen, PHP-Objekte sollen sich wie Java-Objekte benutzen lassen.

4.1.2 Anforderungen

Nach Erstellung derUse-Cases musste festgelegt werden welche Einzelanforderungen sie jeweils an das System stellen. Im Gegensatz zu den Use-Cases, die die Sicht des Anwenders auf das System darstellen, sollte dieser Schritt der Analyse die nötigen Funktionen ergeben, die das System bereitstellen muss, um die Anforderungen des Anwenders zu erfüllen. Weiterhin soll für jede dieser Anforderungen beschrieben werden, wie ein Test aussehen könnte der ermittelt, ob die Anforderung erfüllt ist.

1. **ScriptEngine**. Die PHP-ScriptEngine muss für den Anwender verfügbar sein. Die aktuelle Java-Umgebung muss so konfiguriert sein, dass die ScriptEngine für PHP vom ScriptEngineManager über die in 3.3 beschriebenen Auffindungsmechanismen gefunden und instaziiert werden kann. Diese Anforderung ist eigentlich Grundlage des kompletten Systems, streng genommen wird sie aber nur von *UC.01* gestellt. **Test**: Durch einfache Ausgabe auf der Konsole soll schnell ermittelbar sein, welche ScriptEngines geladen und einsatzbereit sind.

2. **Übersetzen von PHP-Quelltext**. Dem System übergebener PHP-Quelltext soll gemäß des in `javax.script` vorhandenen Interfaces `Compilable` in eine ausführbare Form übersetzt werden. Das Ergebnis eines solchen Übersetzungsvorgangs soll mehrfach ausgeführt werden können.Beim Übersetzen auftretende syntaktische oder semantische Fehler sollen dem Anwender als Java-Exception zurückgegeben werden. Die Umsetzung dieser Anforderung ist eine Vorraussetzung für die Realisierung fast aller weiteren Anforderungen, weswegen sie ebenfalls *UC.01* zugeordnet wird. **Test**: die Übergabe fehlerhaften PHP-Quelltextes soll zu einer Java-Exception führen.

3. **Ausführen beliebiger PHP-Skripte**. Es soll dem Anwender möglich sein, beliebige PHP-Skripte auszuführen. Diese Skripte sollen in vielfacher Form an das

System übergeben werden können, auch ohne dass der Anwender selbst Java-Quelltext schreiben muss. Zur Laufzeit des PHP-Skriptes auftretende Fehler sollen ebenfalls als Java-Exception an den Anwender weitergereicht werden. Aufbauend auf die beiden vorherigen Anforderungen realisiert diese Anforderung die in *UC.01* geforderte Funktion. **Test**: Es soll auf der Kommandozeile der Name einer Datei übergeben werden, deren Inhalt als PHP-Skript ausgeführt wird.

4. **Datenaustausch**. Zwischen den beiden Programmiersprachen soll der Austausch beliebiger Daten möglich sein. Simple (skalare) Datentypen sollen in ihre jeweilige Entsprechung in der anderen Sprache umgewandelt werden, komplexe Datentypen sollen in der anderen Sprache als Referenz verfügbar sein. Auch die Datenübergabe mittels des vom JSR 223 definierten ScriptContext soll möglich sein. Sowohl *UC.03* als auch *UC.04* stellen diese Anforderung. **Test**: Ein Mittels des Kontexts übergebenes Java-Objekt soll in PHP verändert werden und diese Änderungen sollen in Java ausgelesen werden.

5. **Erzeugen von Java-Objekten**. Innerhalb eines ausgeführten PHP-Skripts soll es möglich sein, Java-Objekte zu erzeugen und auf deren Methoden und Attribute zuzugreiffen. Dieser Zugriff soll für den PHP-Anwnder möglichst einfach und intuitiv möglich sein. Diese Anforderung ist ein wesentlicher Teil von *UC.03*. **Test**: Es soll ein Skript aufgerufen werden, welches ein Java-Objekt erzeugt, eine Methode dieses Objektes aufruft und deren Ergebnis an die Java-Applikation zurückgibt.

6. **Java-Exceptions**. Neben dem Zugriff auf Java-Objekte soll es dem PHP-Anwender auch möglich sein aus einem PHP-Skript heraus beliebige Java-Exceptions zu werfen. Dem Anwender soll ermöglicht werden diese Exceptions entweder wie andere Java-Objekte vor dem Werfen selbst zu erzeugen, oder er soll beim Werfen einen Klassennamen übergeben können, aus dem dann eine Exception erzeugt werden. Der PHP-Anwender soll ebenfalls in die Lage versetzt werden überprüfen zu können, ob Java-Exceptions aufgetreten sind, und aufgetretene Java-Exceptions abarbeiten zu können. Diese Anforderung wird explizit vom *UC.03* gestellt. **Test**: Es soll ein PHP-Skript ausgeführt werden das eine Java-Exception provoziert und auf diese Exception reagiert.

7. **Weitere Java-Funktionalität**. Implizit fordert *UC.03* ebenfalls, dass das System dem PHP-Anwender erlaubt grundlegende Funktionalität der Java Laufzeitumgebung zu nutzen. Er soll beispielsweise überprüfen können, ob ein Java-Objekt identisch mit einem anderen ist (`equals()`), ob es Instanz einer bestimmten Klasse ist (`instanceof`), oder ob dieses sicher auf eine andere Klasse zu casten ist. **Test**: Innerhalb eines PHP-Skripts sollen auf einem Java-Objekt jeweils die oben genannten Tests durchgeführt werden.

8. **Aufrufen von Methoden in übersetzten Skripten**. Das zu entwickelnde Softwaresystem soll das Aufrufen sowohl von globalen Funktionen als auch von Objektmethoden in übersetzten PHP-Skripten aus Java heraus erlauben, hierzu soll das Interface `javax.script.Invocable` implementiert werden. Die übergebenen Funktions- und Methodenargumente sollen wie schon beim Datenaustausch in ihre PHP-Entsprechungen umgewandelt werden. Diese Anforderung stellt einen wesentlichen Teil des *UC.04* dar. **Test**: Es soll aus einer Java-Applikation heraus eine PHP-Funktion aufgerufen werden, die ein PHP-Objekt zurückgibt, welches ein Java-Interface implementiert. Auf diesem Objekt soll dann eine Interface-Methode aufgerufen werden.

9. **Setzen von php.ini Parametern**. Dem Anwender soll die Möglichkeit geboten werden gezielt php.ini-Werte* zu setzen oder zu verwenden, um das Laufzeitverhalten von PHP wie gewohnt zu beeinflussen. Diese Werte sollen aus den Java-System-Properties ausgelesen werden, da diese sowohl zur Laufzeit der Java-Applikation programmatisch als auch vor dem Start der Java-VM beispielsweise über die Kommandozeile gesetzt werden können. Die Datei **php.ini** ist der übliche Weg das Laufzeitverhalten des PHP-Interpreters zu beeinflussen. Somit werden die von *UC.02* gestellten Anforderungen komplett erfüllt. **Test**: Es soll der php.ini-Wert `include_path` über die Kommandozeile gesetzt und in einem PHP-Skript ausgelesen werden.

Natürlich decken die oben aufgelisteten Anforderungen nur die grundlegende Funktionalität des Systems ab. Im Lauf der Implementierung wurden noch weitere notwendige und nützliche Funktionen gefunden und realisiert.

*Werte, die das Laufzeitverhalten des PHP-Interpreters beeinflussen werden auch *php.ini-Werte* genannt, da sie in der gleichnamigen Datei festgehalten werden

4.2 Design

Eine JSR 223-Implementierung für PHP besteht gezwungenermassen aus zwei Teilen: Zum einen einem Java-Teil, welcher im Wesentlichen die ScriptEngineFactory enthält. Dieser ist komplett in Java ohne die Verwendung von nativen (JNI-) Aufrufen implementiert und deckt sämtliche Funktionalität des Auffindungsmechanismus ab. Zum anderen einem nativen Teil, welcher aus einem C- oder C++ - Programm, das die nötigen Aufrufe an PHP vornimmt, besteht. Die eigentliche ScriptEngine verbindet diese beiden Teile, indem sie aus Java heraus dieses in nativen Code übersetzte Programm mittels JNI anspricht. Um allerdings JSR 223 überhaupt nutzen zu können, müssen alle Klassen aus dem Package `javax.script` im Java-Classpath liegen, damit dem Anwender der Zugriff auf die ScriptEngine über den ScriptEngineManager möglich ist. Hierzu kann entweder eine Java-Runtime der Version 6 oder aber eine eigene Implementierung dieser Klassen genutzt werden. Die im folgenden beschriebene JSR 223 Implementierung trägt den Namen "Turpitude".

4.2.1 Java-Teil

Eine JSR 223-Implementierung muss sich um für den ScriptEngineManager auffindbar zu sein gemäß der *Jar File Specification* [JAR03] als sogenannter *Service Provider** registrieren. Hierzu muss sich im Verzeichnis `META-INF/services` des Archivs eine Datei mit dem Namen der Service-Klasse (in diesem Fall `ScriptEngineFactory`) befinden, in welcher verfügbare ScriptEngineFactories zeilenweise aufgelistet werden.

Das Package `net.xp_framework.turpitude` enthält alle Klassen der Implementierung. Die ScriptEngineFactory-Implementierung heißt `PHPScriptEngineFactory` und implementiert direkt das Interface `ScriptEngineFactory` aus `javax.script`. Im Gegensatz dazu implementiert die Klasse `PHPScriptEngine` nicht direkt das Interface `ScriptEngine`, sondern erbt von der abstrakten Klasse `AbstractScriptEngine`. Diese bringt für viele Varianten der `eval()`-Methode schon eine Realisierung, sowie mit der Klasse `SimpleScriptContext` schon einen `ScriptContext` mit. Um den übergebenen Skriptcode tatsächlich auszuführen, muss dieser an den mittels der Java-Methode `System.loadLibary()` geladenen nativen Teil der Implementierung weitergegeben werden. Die Kommunikation mit diesem nativen Teil findet mittels JNI-Methoden statt. Weiterhin implementiert die PHPScriptEngine die beiden in `javax.script` enthaltenen Interfaces `Compilable` und `Invocable`, deren Funktionalität zur Erffüllung einiger der Use-Cases benötigt wird. Für die Methoden aus Compilable wird noch eine weitere Klasse - `PHPCompiledScript` - erstellt, welche direkt von der abstrakten Klasse `javax.script.CompiledScript` erbt. Der Standard schreibt zwar vor, dass die ScriptEngine das Interface Invocable implementiert, allerdings ergibt das aus Sicht des Autors nur wenig Sinn. Es bleibt unklar auf welchem der

*Ein *Service Provider* ist eine spezifische Implementierung eines Interfaces oder einer abstrakten Klasse

übersetzten Skripte Methoden und Funktionen aufgerufen werden sollen. Aus diesem Grund wird neben der `PHPScriptEngine` auch das `PHPCompiledScript` dieses Interface implementieren und Aufrufe der Interfacemethoden bei der `PHPScriptEngine` werden an das zuletzt übersetzte Script weitergeleitet.

Die Kommunikation zwischen Java und PHP findet über zwei unterschiedliche Kanäle statt: Bei der Rückgabe von Daten aus einem PHP-Skript werden diese einfach kopiert, wobei primitive Datentypen (bool, long, double und string) einfach in ihre Java-Äquivalente (java.lang.Boolean, java.lang.Long, java.lang.Double und java.lang.String) umgesetzt werden. Da alle Arrays in PHP assoziativer Natur sind, werden sie als untypisierte `java.util.HashMap` zurückgegeben, was bedeutet dass sowohl Schlüssel als auch Wert vom Typ `java.lang.Object` sind. Einzig für PHP-Objekte wird eine gesonderte Java-Klasse benötigt: `PHPObject`. Sie enthält neben dem Klassennamen auch alle Eigenschaften eines PHP-Objektes. Diese werden in einer HashMap gespeichert, mit dem Namen der Eigenschaft als Schlüssel.

Leider lässt die JSR 223-Spezifikation sehr viele Fragen unbeantwortet, so wird beispielsweise zur Typkonversion nur lapidar auf Implementierungsdetails verwiesen. Da der Datenaustausch zwischen PHP und Java allerdings einen nicht unwesentlichen Bestandteil einer JSR223-Implemenation darstellt, muss gezwungenermaßen eine Konvention gefunden werden. Diese wird in den Tabellen 4.1 und 4.2 erläutert, sowohl von Java nach PHP als auch umgekehrt. Weiterhin wird bei der Typkonversion von PHP nach Java zwischen Methodenaufrufen und Werterückgaben unterschieden. Bei Methodenaufrufen wird versucht, in simple Datentypen wie `int` und `double` umzuwandeln. Weil der aber Standard oft verlangt dass Objekte zurückgegeben werden, wird bei der Rückgabe von Werten in den entsprechenden Objektyp wie zum Beispiel `java.lang.Boolean` konvertiert. Einige Java-Klassen und PHP-Objekte werden speziell konvertiert was in den Tabellen gesondert aufgeführt ist. Beispielsweise wird der `java.lang.String` nicht zu einem `TurpitudeJavaObject`, sondern zu einem String in PHP. Die Tabellen enthalten auch Verweise auf spezielle PHP-Klassen, die später im Kapitel 4.2.2 erläutert werden.

Java-Typ	PHP-Typ
jlong	long
jint	long
jshort	long
jchar	long
jbyte	long
jboolean	bool
jfloat	double
jdouble	double
jstring	string
jobject	TurpitudeJavaObject
jclass	TurpitudeJavaClass
type[]	TurpitudeJavaArray

Tabelle 4.1: Typkonversion Java nach PHP

Die Eingabe von Daten aus Java heraus in ein PHP-Skript geschieht - gemäß der JSR223-Spezifikation - mittels des Kontextes der ScriptEngine. Hierzu wird dieser - zusammen mit einigen Umgebungsvariablen und -informationen - in eine globale Variable injiziert, deren Name sich in der ScriptEngine setzen lässt. Im Unterschied

PHP-Typ	Methodenaufruf	Werterückgabe
long	jlong	java.lang.Long
double	jdouble	java.lang.Double
bool	jboolean	java.lang.Boolean
array	java.util.HashTable	java.util.HashTable
object	PHPObject	PHPObject
constant	jstring	jstring
string	jstring	jstring
TurpitudeJavaClass	jclass	jstring
TurpitudeJavaObject	jobject	jobject
TurpitudeJavaArray	jarray	jarray

Tabelle 4.2: Typkonversion PHP nach Java, Methodenaufruf und Werterückgabe

zu den oben beschriebenen Rückgabewerten stellen die Kontext-Objekte Referenzen dar, weshalb sich Änderungen an diesen unmittelbar auf die entsprechenden Java-Objekte auswirken.

Die Abbildung 4.2 zeigt die beteiligten Klassen und ihre Abhänigkeiten untereinander, zusammen mit einigen wichtigen Methoden und Attributen.

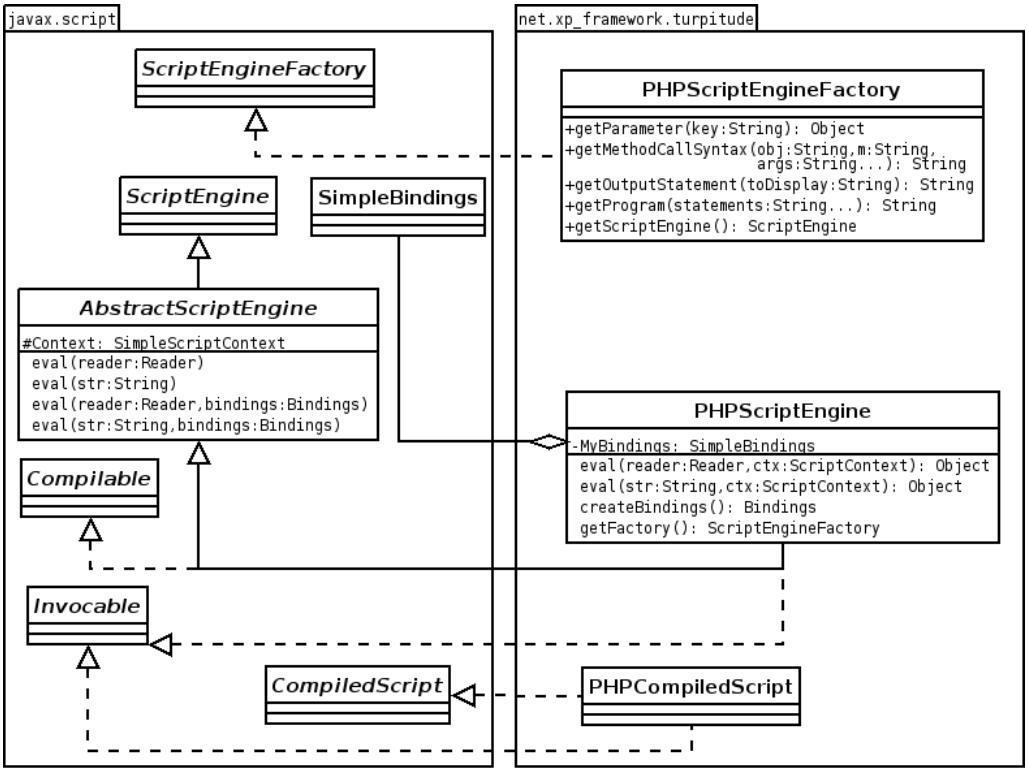

Abbildung 4.2: JSR 223 Implementierung - Architektur

Der JSR 223 spezifiziert lediglich eine einzige Exception (`ScriptException`) um alle möglichen Laufzeitfehler innerhalb einer Implementierung abzudecken. Um dem Anwender trotzdem eine Möglichkeit einer differenzierten Fehlerauswertung zu geben, werden drei zusätzliche Exceptions eingeführt: die `PHPScriptException`, die `PHPCompileException` und die `PHPEvalException` (siehe Abbildung 4.3). Die `PHPScriptException` dient zum Einen als Oberklasse, zum anderen kann der Anwender durch sie erkennen, ob das aufgetretene Problem implementationsspezifisch ist. Die beiden anderen neuen Exceptions werden genutzt um Fehler beim Übersetzen (`PHPCompileException`) oder beim Ausführen (`PHPEvalException`) eines Skripts voneinander abzugrenzen.

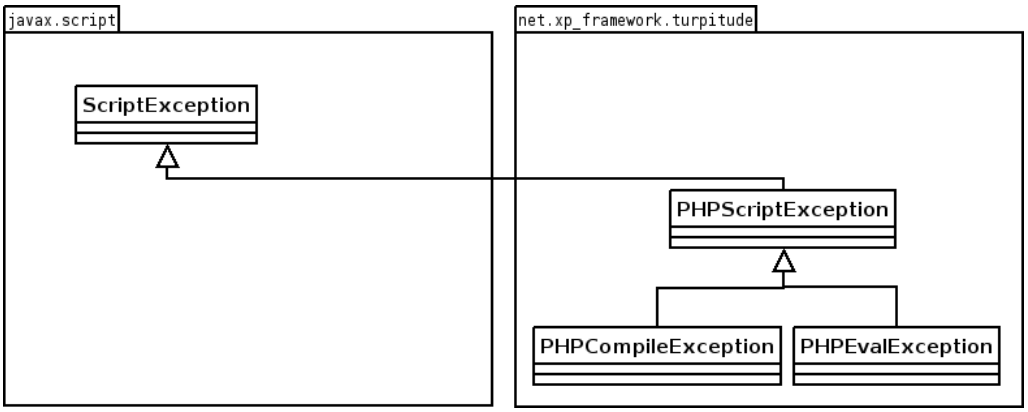

Abbildung 4.3: JSR 223 Implementierung - Exceptions

4.2.2 Nativer/PHP-Teil

Der native Teil der Implementierung teilt sich wiederum auf in JNI-Methoden, für welche die Headerdateien automatisch aus der Java-Klasse der PHPScriptEngine generiert werden können, und in die Implementierung einer SAPI. Hierzu muss im Wesentlichen ein sogenanntes `sapi_module_struct` angelegt und befüllt werden, welches neben einigen Strings hauptsächlich Funktionspointer auf Rückruffunktionen enthält, welche vom PHP-Interpreter aufgerufen werden. Da es kaum möglich ist, dieses Konstrukt sinnvoll in eine objektorientierte Architektur einzufügen, wird auf eine solche vollständig verzichtet. Das `sapi_module_struct` und die dazugehörigen Funktionen werden auf traditionelle, prozedurale Art und Weise implementiert.

Die zu implementierenden SAPI-Funktionen ergeben sich aus den Anforderungen der Zend-API und sollen an dieser Stelle nur sehr kurz erläutert werden: die Funktionen `turpitude_read_cookies`, `turpitude_flush`, `turpitude_send_headers`, `turpitude_send_header` und `turpitude_log_message` können leer, beziehungsweise auf triviale Art und Weise implementiert werden, da sie für den geplanten Einsatz der PHP-Umgebung keine Relevanz haben. Die Funktionen `turpitude_startup` und

turpitude_register_variables leiten den Aufruf einfach an die entsprechenden Zend-API Funktionen php_module_startup respektive php_import_environment_variables weiter, lediglich die Funktion turpitude_error_cb, welche die Rückruffunktion für PHP-Laufzeitfehler darstellt, bedarf einer komplizierteren Implementierung, da sie den PHP-Fehler als passende Java-Exception an den Anwender weitergeben soll.

Die JNI-Methoden ergeben sich aus dem Java-Quelltext, da sie Implementierungen von in Java als native deklarierten Methoden darstellen. Ihre Hauptaufgabe ist das Ausführen von PHP-Quelltext beziehungsweise von PHP-Anweisungen. Ausnahmen bilden lediglich die Methoden startUp() und shutDown() der PHPScriptEngine, welche die Initialisierung beziehungsweise die kontrollierten Dekonstruktion der PHP-Laufzeitunmgebung und er Turpitude-spezifischen Klassen übernehmen.

Um die Implementierung in PHP nutzen zu können muss auch hier eine Schnittstelle geschaffen werden. Leider schreibt die Spezifikation keine API für die Skriptseite vor, was zu einer Nichtaustauschbarkeit der Implementierung führen kann. Nichtsdestotrotz werden auf PHP-Seite folgende Klassen eingeführt, welche im Wesentlichen JNI-Funktionen abbilden, und in vielen Fällen auch die JNI-Syntax, zum Beispiel für Java-Methoden, verwenden:

TurpitudeEnvironment - bietet neben der Methode getScriptContext(), die den JSR223 ScriptContext zurückgibt, auch Schnittstellen an um grundlegende Java-Funktionalität aus PHP heraus anzusprechen: FindClass() erzeugt die Repräsentation einer Java-Klasse in PHP, instanceOf() überprüft, ob ein Java-Objekt Instanz einer bestimmten Java-Klasse ist. Weiterhin werden mit throw(), throwNew(), exceptionOccurred() und exceptionClear() Methoden angeboten, um Java-Exceptions zu werfen und zu fangen. TurpitudeEnvironment implementiert das bekannte *Singleton-Entwurfsmuster*, kann also nicht mehrfach existieren und wird zusätzlich unter einem konfigurierbaren Namen in das PHP-Superglobal* $_SERVER eingeführt, in welchem laut dem PHP-Manual [PHP06a] SAPI-Informationen über die Laufzeitumgebung gespeichert werden sollen.

TurpitudeJavaClass - PHP-Repräsentation einer Java-Klasse, wird mittels der Methode TurpitudeEnvironment::findClass() erzeugt. Bietet mit findMethod(), findStaticMethod() und findConstructor() Möglichkeiten um gewöhnliche Methoden, statische Methoden und Konstruktoren der repräsentierten Java-Klasse zu erzeugen. Weiterhin können mittels der Methode create() Instanzen der Klasse erzeugt, und mittels der Methode invokeStatic() können statische Methoden der Klasse aufgerufen werden. Enthält als Attribut den JNI-formatierten Klassennamen.

TurpitudeJavaMethod - bildet eine Java-Methode in PHP ab. TurpitudeJavaMethod bietet selbst keine für den Anwender aufrufbaren Methoden an und wird

*Superglobals sind Variablen, auf die innerhalb eines PHP-Skriptes aus jedem Scope heraus zugegriffen werden kann. Superglobals können nicht vom PHP-Anwender erzeugt werden, sondern werden von der PHP-Laufzeitunmgebung verwaltet.

nur als Parameter für `TurpitudeJavaClass::create()` und beim Methodenaufruf auf Objekten und Klassen benötigt. Als Attribute werden der Methodenname, deren Signatur sowie ein Indikator, ob die Methode statisch aufgerufen werden kann, vorgehalten.

TurpitudeJavaObject - repräsentiert ein Java-Objekt, wird mittels der Methode `TurpitudeJavaClass::create()` erzeugt. Neben Methoden die den sowohl lesenden als auch schreibenden Zugriff auf Attribute der Java-Klasse erlauben (`javaGet()` und `javaSet()`), können auch direkt Methoden des gekapselten Java-Objektes mittels `javaInvoke()` aufgerufen werden. Falls möglich soll der Methodenaufruf aber intuitiv nach folgendem Schema erlaubt werden:

```
$object->method($param1, $param2, ...);
```

Listing 4.1: angestrebte Syntax zum Aufruf von Java-Methoden in PHP

Rückgabewerte von Methodenaufrufen werden entweder auf skalare Typen in PHP oder wieder als TurpitudeJavaObject-Objekte abgebildet. Eine Besonderheit bilden allerdings Java-Arrays: Diese könnten zwar als einfache Java-Objekte abgebildet werden, allerdings würde das nur den Zugriff auf Attribute erlauben, die allen Java-Arrays gemeinsam sind - wie beispielsweise `length`. Folglich muss eine weitere Klasse eingeführt werden, um dem Anwender auch den Zugriff auf die Elemente eines Java-Arrays zu ermöglichen:

TurpitudeJavaArray - erlaubt den direkten Zugriff auf Elemente eines Java-Arrays in PHP. Allerdings können keine neuen Einträge erzeugt werden, da Java-Arrays - anders als PHP-Arrays - nicht als HashMap, sondern als "echte" Arrays im Speicher abgebildet sind, und eine Erweiterung dieses Bereiches nicht ohne Weiteres möglich ist. Der Zugriff auf diese Arrays soll nicht nur über Funktionen wie `get()` und `set()`, sondern auch über den Klammern-Operator [ ] möglich sein, um den Anwender einen intuitiven Umgang zu bieten. Außerdem soll das iterieren über Java-Arrays sowohl über die objektorientierte, in PHP5 enthaltene Schnittstelle `Iterator`, als auch über den normalen PHP-Operator `foreach` möglich sein. Das `TurpitudeJavaArray` implementiert das PHP-Interface `IteratorAggregate`.

TurpitudeJavaArrayIterator - erlaubt das Iterieren über Java-Arrays in PHP. Der `TurpitudeJavaArrayIterator` implementiert das PHP-Interface `Iterator` um das objektorientierte Iterieren über Java-Arrays zu ermöglichen. Intern wird die aktuelle Position im Array in einer einfachen Zählvariable gespeichert. Wird diese größer oder gleich der Länge des Java-Arrays wird der Iterator ungültig und muss mittels `rewind()` zurückgesetzt werden. `next()` inkrementiert diesen Zähler um eins, die Methoden `current()` und `key()` liefern das aktuelle Element beziehungsweise die aktuelle Position.

Der Zusammenhang zwischen Java-Arrays, den Turpitude-Klassen und PHP-Interfaces ist in der Abbildung 4.4 dargestellt. Das von Java bereitgestellte Array wird in PHP in einem `TurpitudeJavaArray` gekapselt. Ein Aufruf der Methode

`getIterator()` erzeugt einen `TurpitudeJavaArrayIterator`, der intern eine Referenz auf das `TurpitudeJavaArray` vorhält.

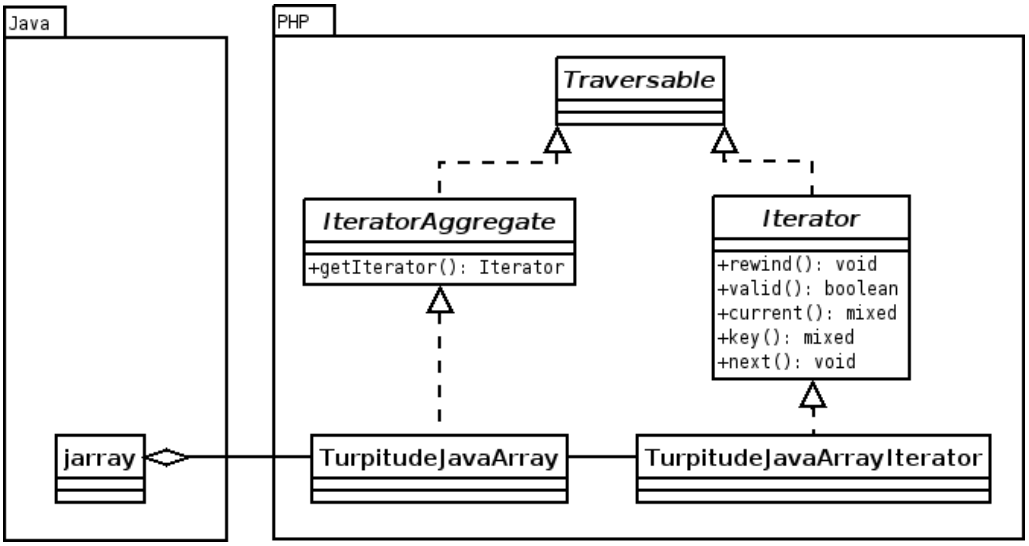

Abbildung 4.4: Java-Arrays in PHP

Ein typischer Ablauf eines PHP-Skriptes, welches auf Java-Objekte und Methoden zugreift, soll wie in Abbildung 4.5 aufgezeigt aussehen: Das `TurpitudeEnvironment` besteht über die komplette Laufzeit des Skriptes. Der Anwender erzeugt mittels der Methode `findClass()` des `TurpitudeEnvironment` eine `TurpitudeJavaClass` und ruft auf dieser die Methode `findConstructor()` auf, welche ihm eine Instanz der Klasse `TurpitudeJavaMethod` zurückgibt. Diese wiederum kann der Anwender der TurpitudeJavaClass beim Aufruf von `create()` übergeben, um schlussendlich eine Instanz der Klasse zu erzeugen. Um nun auf dem Java-Objekt eine Methode aufzurufen, muss diese zunächst wieder per `findMethod()` erzeugt werden, um dann der Methode `javaInvoke()` zusammen mit den geforderten Parametern übergeben zu werden.

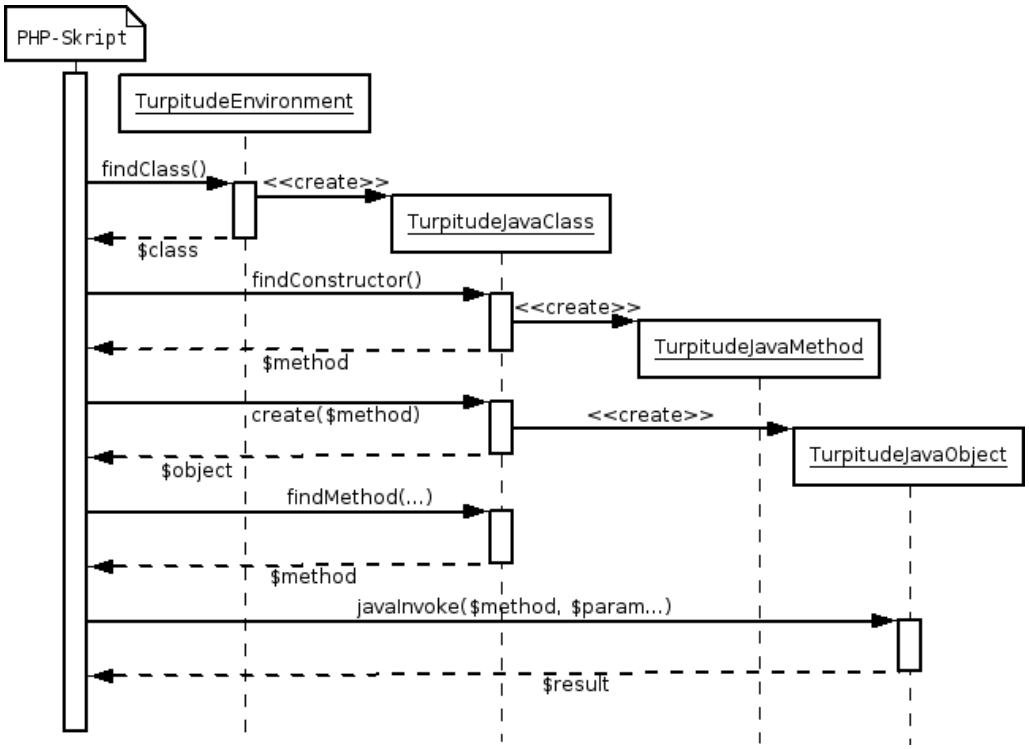

Abbildung 4.5: Ablauf eines PHP-Skriptes mit Zugriff auf Java-Objekte

4.3 Implementierung

Im Folgenden wird beschrieben, wie die in den beiden vorherigen Kapiteln 4.1 und 4.2 gewonnenen Erkenntnisse umgesetzt wurden. Jede in 4.1.2 gefundene Anforderung wird - soweit sinnvoll möglich - in einem eigenen Unterkapitel aufgeführt.

4.3.1 Infrastruktur und ScriptEngine

Um mit der Implementierung beginnen zu können, musste zunächst ein als dynamisch ladbare Bibliothek verfügbares PHP erzeugt werden. Hierzu wurde von [PHP06b] ein PHP in der Version 5.2.0 im Quelltext heruntergeladen und übersetzt, nachdem es mittels des Kommandozeilenparameters "–enable-embed=shared" konfiguriert wurde. Die so erzeugte *libphp5.so* konnte, zusammen mit den vorhandenen Headerdateien, zur Entwicklung genutzt werden.

Nun musste noch dafür gesorgt werden, dass sich die vom JSR 223 verlangten Klassen aus `javax.script` im Classpath befinden. Nachdem allerdings die Klassen aus der Referenzimplementierung aus den zum Teil in 3.5 beschriebenen Gründen nicht in Frage kamen, wurde von [JAV06a] ein *Java Development Kit (JDK)* der Version 6 heruntergeladen. Da dieses JDK nicht als System-VM genutzt werden kann, wurde ein Makefile erstellt, welches die zur Übersetzung und Ausführung nötigen Kommandos vereinfacht.

Nachdem diese Vorarbeiten geleistet waren, konnte mit der eigentlichen Implementierung begonnen werden. Der erste Schritt war die Abbildung der ersten Anforderung (verfügbare ScriptEngines auflisten) erstens um sicherzustellen, dass die Entwicklungsumgebung den Anforderungen gerecht wird, und zweitens um einen ersten Eindruck der JSR 223 API zu erhalten. Also wurde im Package `samples` eine Klasse namens `EngineList` geschrieben, die einen ScriptEngineManager erzeugt, und mittels der Methode `getEngineFactories()` eine Liste aller verfügbaren ScriptEngines erstellt. Der Inhalt dieser Liste wird dann mittels `System.out.println()` ausgegeben. Nachdem dem Makefile ein Target namens "list" hinzugefügt wurde, ergab ein erstes Ausführen dieser Klasse zum einen keine Fehler, und zum andern, dass dem JDK 6 mit *Mozilla Rhino* bereits eine ScriptEngine beiliegt:

```
# make list
found 1 available ScriptEngines:
Engine: Mozilla Rhino
#
```

Listing 4.2: erste Tests

Im Folgenden wurde die Klasse `PHPScriptEngineFactory` erstellt, welche - wie in Kapitel 4.2.1 beschrieben - das Interface `ScriptEngineFactory` implementiert. Hierbei erwähnenswert sind die Methoden `getProgram()` und `getMethodCallSyntax()`. Erstere erstellt aus als Strings vorliegenden einzelnen Anweisungen ein ausführbares

PHP-Programm, zweitere erzeugt aus den übergebenen Parametern einen der PHP-Syntax entsprechenden Methodenaufruf. Weiterhin enthält sie neben den weiter unten beschriebenen nativen Methoden `startUp()` und `shutDown()` die Methode `read()`, die aus einem übergebenen java.io.Reader eine Zeichenkette erstellt, die dann als Sourcecode verwendet werden kann. Auf diese Weise kann der Anwender beispielsweise leicht seinen PHP-Quelltext in Dateien verwalten, ohne jedesmal eigens einen Java-String erzeugen zu müssen. Wurde nun die EngineList ausgeführt, stellte sich heraus, dass der ScriptEngineManager die neue ScriptEngine schon erkannte:

```
# make list
found 2 available ScriptEngines:
Engine: Mozilla Rhino
Engine: XP-Framework Turpitude PHP Engine
#
```

Listing 4.3: Neue ScriptEngine

Nachdem dieser Schritt getan war, wurde mit der Erstellung der eigentlichen ScriptEngine begonnen. Deren Implementierung gestaltete sich sehr einfach, da durch das Ableiten von der Klasse `AbstactScriptEngine` fast alle Varianten der `eval()`-Methode schon vorimplementiert waren. Lediglich das Auslesen des Skriptquelltextes aus einem übergebenen `Reader` musste selbst umgesetzt werden. Die Tatsache, daß PHP ursprünglich ausschließlich als CGI ausgeführt wurde*, merkt man dem Kern der Sprache noch deutlich an. So muss eine SAPI Funktionen aufgerufen werden, welche den Beginn des sogenannten Requests, respektive dessen Ende anzeigen. Auch müssen Funktionen zum Setzen und Auslesen von HTTP-Headern und Cookies bereitgestellt werden. Die nötigen Aufrufe der Zend-Funktionen `php_module_startup()` und `php_request_startup` werden in der PHPScriptEngine von zwei privaten, nativen Methoden erledigt: `startUp()` und `shutDown()`. Zusätzlich wurde im Konstruktor der PHPScriptEngine ein sogenannter "ShutdownHook" eingefügt, welcher dafür sorgt, dass zumindest beim Herunterfahren der Virtual Machine `shutDown()` aufgerufen wird.

4.3.2 Übersetzen und Ausführen von Skripten

Nachdem auf der Java-Seite die Grundlagen gelegt worden waren, konnte mit der Realisierung des nativen Teiles begonnen werden. Die lückenhafte - an vielen Stellen sogar gänzlich fehlende Dokumentation - der Zend-Engine machte es unmöglich herauszufinden was der von den Zend-Entwicklern vorgesehene Weg ist, um PHP auf die beschriebene Art und Weise einzubetten. Folglich musste durch reines Ausprobieren ermittelt werden, wie vorgegangen werden sollte. Zunächst musste ein zur Übergabe an die Zend-Engine geeignetes `sapi_module_struct` definiert werden. Hierbei handelt es sich um ein C-struct, welches im Wesentlichen Pointer auf Callback-Funktionen

*siehe hierzu auch 2.2.1

enthält. Viele dieser Funktionen, wie `read_cookies` und `send_headers`, wurden leer implementiert, da sie für die geplante Art des Einsatzes nicht sinnvoll sind. Lediglich `error_callback` soll hier erwähnt werden: innerhalb dieser Methode werden zwei globale Variablen gesetzt, eine zeigt an ob die Fehlermethode aufgerufen wurde, die andere hält die beim Aufruf ausgelesene Fehlermeldung vor, um sie später an den Benutzer weiterzugeben. Ein erster Ansatz PHP-Code auszuführen war die Zend-API Funktion `zend_eval_string`. Nachdem der nötige "Boilerplate"-Code geschrieben war, zeigten sich auch Erfolge: eine in Java geschriebene HelloWorld-Klasse, welche den Quelltext "echo 'Hello World';" direkt an die ScriptEngine weiterreicht, erzeugte die gewünschte Ausgabe. Eingehendere Tests unter Zuhilfenahme der `ScriptExec` Klasse, die ein Skript aus einer ihr übergebenen Datei ausliest und an die ScriptEngine weitergibt, förderten allerdings einen gravierenden Nachteil von zend_eval_string zutage: Sobald als Parameter "retval_ptr" nicht mehr `NULL` übergeben wird, wird nicht mehr das komplette Skript, sondern nur noch die erste Zuweisung innerhalb des Quelltextes ausgeführt. Wurde dieses Skript

```
$var = "Test";
for ($i=0; $i<10; $i++) {
    printf("Hello %s\n", $var);
}
```

Listing 4.4: Testscript für zend_eval_string()

mit retval_ptr == NULL ausgeführt, so wurde wie erwartet der Text "Hello Test" zehnfach auf der Konsole ausgegeben. Wurd allerdings ein Pointer auf ein valides zval-struct übergeben, wurde nichts mehr ausgegeben, und jene struct enthielt als Wert den String "Test", und zusätzlilch gab zend_eval_string einen Fehlercode als eigenen Rückgabewert zurück. Dieses Phänomen ließ sich auch mit anderen, beliebig komplizierten Zuweisungen reproduzieren. Aus diesen Tests wurde geschlossen, dass sobald ein valider Pointer auf einen Rückgabewert übergeben wird, die Funktion das Skript nur bis einschließlich der ersten Zuweisung ausführt, und dann abbricht. Aus diesem Grund war `zend_eval_string` ungeeignet, die gestellten Anforderungen zu erfüllen.

Obwohl der Versuch `zend_eval_string` zu nutzen fehlschlug, konnten einige geschriebene Komponenten für weitere Ansätze weiterverwendet werden, insbesondere die Funktionen `zval_to_jobject` zur Konvertierung des Zend-Engine Datentyps `zval` in den JNI-Datentyp `jobject`, und `java_throw`, die das Werfen von Java-Exceptions aus dem nativen Teil heraus vereinfacht. Die Entwicklung der Funktion `zval_to_jobject` erwies sich aufgrund der fehlenden Dokumentation der Zend-Engine als unerwartet kompliziert, insbesondere das Auslesen von Objekten aus zvals erfordert intimes Wissen über Interna der Zend-Engine. So werden beispielsweise Eigenschaften, *Properties* genannt, analog zu den Elementen von Arrays intern in einer `Hashtable` gespeichert, wobei die Schlüssel-Namen den Namen der Eigenschaft darstellen. Allerdings wird dieser Schlüssel im Falle von privaten und geschützten

(protected) Eigenschaften gesondert kodiert. In diesem Fall beginnt er mit einem Nullbyte (\0), gefolgt vom Namen der Objektklasse und einem weiteren Nullbyte und dem eigentlichen Namen des Propertys. Verständlicherweise sorgt diese Kodierung in C, wo das Nullbyte als Endzeichen für Strings verwendet wird, für einige Probleme. Über die Beweggründe der Zend-Entwickler diese Kodierung zu wählen, kann nur spekuliert werden, aber die Vermutung liegt nahe, dass mit einem Nullbyte beginnende Strings in internen Funktionen, da sie für C-Stringfunktionen die Länge 0 haben, als ungültig behandelt werden, und so einfach nirgendwo auftauchen. Da die Java-Klasse PHPObject, die laut Design an die JVM zurückgegeben werden sollte, wenn ein PHP-Objekt als Rückgabewert auftaucht, allerdings alle Attribute des jeweiligen PHP-Objektes enthalten sollte, mussten diese kodierten Attributnamen "entwirrt" werden. Ein weiteres Problem, das gelöst wurde war, dass alle Zend-API Funktionen, die Quelltexte interpretieren, erwarten, dass keine PHP-Tags mehr in diesem enthalten sind. PHP-Skripte werden üblicherweise innerhalb der Zeichenfolge "<?php" und "?>" eingeschlossen, um sie von den umgebenden Inhalten (meist HTML) abzugrenzen. Die PHPScriptEngine filtert diese Zeichen automatisch aus dem Quelltextstrom.

Da zend_eval_string offensichtlich ungeeignet war die Anforderungen zu erfüllen, wurde nach einer Alternative gesucht, und die Funktion compile_string gefunden. Da diese - wie bereits der Name andeutet - den Quelltext lediglich in einen zend_opcode_array übersetzt und dieser später ohnehin seperat ausgeführt werden muss, wurde an dieser Stelle dem geplanten Projektablauf vorgegriffen und das Interface Compilable gleich mitimplementiert. Im Rahmen dieses Schrittes wurde die Klasse PHPCompiledScript wie geplant realisiert. Diese enthält die native Methode execute, von welcher der von der PHPScriptEngine erzeugte und in einen im PHPCompiledScript enhaltenen ByteBuffer geschriebene Opcode letztendlich ausgeführt wird. Hierzu sollte ursprünglich die Zend-API Methode execute() verwendet werden. Allerdings führte jeglicher Aufruf dieser Methode ausschließlich zu Abstürzen und Speicherzugriffsfehlern, über deren Ursachen nur Vermutungen angestellt werden können. Schlussendlich gelang es trotz aller Hindernisse Zend-Opcode fehlerfrei auszuführen: die Zend-API Methode zend_call_function() lässt sich zur Ausführung jeglichen Opcodes missbrauchen. Es ist lediglich notwendig, einige Typfelder des Opcode-Arrays mit "falschen" Informationen zu füllen. Zusätzlich löst dieser Ansatz ein weiteres Problem: Jede Funktion hat einen Rückgabewert, aber im Gegensatz zu zend_eval_string wird dieser nicht direkt der Methode übergeben, sondern in einem C-struct des Types zend_fcall_info gespeichert und lässt sich von dort mühelos auslesen. Außerdem beeinflusst er das Ausführverhalten von zend_call_function nicht. Allerdings ergaben sich aus dieser Art und Weise Zend-Opcode auszuführen auch einige Abweichungen zu einem "Standard-PHP":

Es war nicht mehr möglich Kommandozeilenparameter an das so ausgeführte Script zu übergeben. Das ist vernachlässigbar, da der JSR 223 diese Art der Datenübergabe nicht vorsieht. Weiterhin erhielt der ansonsten innerhalb eines gewöhnli-

chen Skriptes nutzlose PHP-Befehl `return` plötzlich einen Sinn: das Skript beendete sich und der Rückgabewert wurde mit der zurückgegebenen Variablen befüllt. Getreu dem Motto "It's not a bug, it's a feature" wurde dies allerdings nicht als störend empfunden, da der normale `exit`-Befehl sich weiterhin wie gewohnt verhielt und so nur zusätzliche Funktionalität verfügbar wurde.

4.3.3 Datenaustausch Java nach PHP

Der nächste Schritt war nun die Übergabe von Parametern aus Java an den PHP-Interpreter. Idealerweise sollten die zu übergebenden Objekte nicht nur hin- und herkopiert werden, sondern sie sollten als Repräsentation innerhalb der PHP-Umgebung verfügbar sein. Dies bringt den Vorteil mit sich, dass nicht nur die Daten der Objekte, sondern auch deren Funktionalität dem PHP-Entwickler zur Verfügung steht. Außerdem sollte es dem Anwender möglich sein, Java-Objekte aus PHP heraus zu erzeugen und Methoden auf diesen Aufuzrufen.

Um dies zu erreichen, war es nötig Klassen "von außen" - also ohne PHP-Quelltext zu nutzen - in den Interpreter zu injizieren, und Aufrufe auf Objekte dieser Klassen auf C-Ebene abzuhandeln, da nur auf C-Ebene auf JNI-Funktionalität zugegriffen werden konnte. Diese Art von Klasse wird von Zend *internal class* genannt, da sie nicht im PHP-Userspace, sondern direkt im Interpreter oder in von ihm geladenen Bibliotheken implementiert werden. Leider existiert auch zu dieser Seite der Zend-Engine keinerlei Dokumentation, und so musste viel ausprobiert werden um herauszufinden, wie dies zu bewerkstelligen wäre. Schließlich wurde das Zend-Makro `INIT_OVERLOADED_CLASS_ENTRY` entdeckt, und nach vielen weiteren Versuchen wurde die Bedeutung des Makros und die restlichen Schritte herausgefunden, die nötig sind, um eine solche interne Klasse zu erstellen:

Um der Zend-Engine eine Klasse unterzuschieben muss zunächst ein C-struct definiert werden, das neben einem struct vom Typ `zend_object` namens `std` beliebige weitere Daten enthalten kann. Dieses struct wird für jede Instanz der Klasse vorgehalten. Weiterhin wird ein Array vom Typ `function_entry[]` benötigt, welches die Callbacks - im wesentlichen Funktionspointer auf C-Funktionen - für verschiedene Ereignisse auf dem Objekt enthalten. Im Gegensatz zu im Userland* implementierten Methoden werden diese *built-in functions* genannt. Besonders erwähnenswert sind hier die Funktionen für `__call`, `__get` und `__set`. `__call` wird bei jedem Methodenaufruf aufgerufen, und erhält neben dem Funktionsnamen auch alle übergebenen Parameter, und bietet so die Möglichkeit PHP-Methoden direkt in C zu implementieren, ohne für jede Methode einen eigenen Eintrag im Funktionspointerarray zu benötigen. Allerdings kann es bei solchen mittels `__call` implementierten Methoden zu Überschneidungen mit "echten" Methoden, solchen also die einen Eintrag im

*Als Userland bezeichnet man im Allgemeinen den Bereich den der Anwender beeinflussen kann. Bei PHP umfasst er alles, was der Benutzer implementieren kann: Klassen, Funktionen, Methoden und Variablen.

Funktionspointerarray haben, kommen. In solchen Fällen wird immer zuerst überprüft ob im Array ein Eintrag passenden Namens existiert und gegebenenalls diese Methode aufgerufen, bevor der Aufruf an `__call` weitergereicht wird. `__get` und `__set` werden bei jedem lesenden beziehungsweise schreibenden Zugriff auf ein nicht gefundenes Attribut des Objektes aufgerufen, und erhalten neben dem neuen Wert auch den Namen des Attributs. Normale Attribute eines Objektes werden in einer HashTable gespeichert und auch hier gilt: Ist in der HashTable ein Attribut gesuchten Namens vorhanden wird dieses zurückgeliefert, ansonsten wird je nachdem `__get` oder `__set` aufgerufen.

Weiterhin gibt es noch die beiden Funktionen `__construct` und `__destruct`, die beim Erzeugen beziehungsweise Zerstören des Objektes aufgerufen werden, die Funktionen `__sleep` und `__wakeup`, die beim Serialisieren beziehungsweise Deserialisieren des Objektes zum Einsatz kommen, sowie die Funktion `__cast`, die das Casten des Objektes auf einen anderen Datentyp implementiert. Diese Funktionspointer werden zusammen mit dem Klassennamen an das Makro übergeben, um so einen sogenannten `zend_class_entry` zu erzeugen, der für jede Klasse ein mal global gespeichert wird. Auf diese Art und Weise konnten die in 4.2.2 geforderten Klassen und deren Methoden implementiert werden. Alle Callback-Methoden haben keinen (`void`) Rückgabewert und müssen die Argumente entgegenehmen, die im Makro `INTERNAL_FUNCTION_PARAMETERS` der Zend-API definiert werden. Es enthählt unter anderem einen Pointer auf den Rückgabewert (`return_value`), einen Pointer auf das Objekt (`this`), vor allem aber ein Array der beim Aufruf der PHP-Methode übergebenen Argumente (`argv`) und einen Ganzzahlwert, der die Anzahl der Argumente enthält (`argc`). Um allerdings überhaupt JNI-Funktionen nutzen zu können musste zunächst die native Implementierung der startUp-Funktion der PHPScriptEngine erweitert werden: beim Aufruf dieser Funktion wird der JNI Environment-Pointer als globale Variable gespeichert, auch werden hier die Funktionen aufgerufen, die die Turpitude-Klassen in die PHP-Umgebung injizieren.

4.3.4 TurpitudeEnvironment

Als erstes wurde die Klasse `TurpitudeEnvironment` implementiert. Das Objektstruct enthält neben dem zend_object noch Pointer auf den aktuellen ScriptContext sowie die JNI-Umgebung. Eigene, für PHP lesbare Attribute besitzt die Klasse nicht, also mussten nur die geforderten Methoden umgesetzt werden. Zuvor wurde allerdings getestet, ob das injizieren der Klasse funktioniert hat, und die ScriptEngine wurde derart angepasst, dass jedem Aufruf eines PHP-Skriptes eine Zeile vorangestellt wird, die eine Instanz der Klasse erzeugt und in das PHP-Superglobal `$_SERVER` eingefügt wird. Superglobals sind eine Besonderheit von PHP, normale globale Variablen müssen im Scope einer Funktion oder Methode mittels des Schlüsselwortes `global` bekannt gemacht werden, Superglobals hingegen sind immer und von überall erreichbar. Der Name, unter dem das TurpitudeEnvironment dort eingefügt wird, ist

in der ScriptEngine mittels der Methoden `setVarName()` und `getVarName()` konfigurierbar. Ein `var_dump($_SERVER)` zeigte, dass das injizieren und initialisieren der Klasse funktioniert hatte.

```
$_SERVER["TURP_ENV"] = new TurpitudeEnvironment();
```

Listing 4.5: TurpitudeEnvironment in $_SERVER einfügen

In allen Turpitude-Klassen wird hierzu in der Callback-Methode `__call` der Methodenname ausgelesen, der sich in `argv` an der Stelle mit dem Index 0 befindet. Außerdem wird stets ein Pointer auf die zweite Stelle von `argv`, sowie ein Integer der um eins kleiner ist als `argc`, an die die eigentlichen Methoden implementierenden Funktionen übergeben, zusammen mit `return_value` und eventuell dem dem Objekt-eigenen C-struct. Diese PHP-Methoden implementierenden Funktionen werden im weiteren Verlauf auch *Handler* genannt. Danach wird in jeder `__call`-Funktion mittels `strcmp` der Name der aufgerufenen Methode mit den Namen der zu implementierenden Methoden verglichen, und falls eine Übereinstimmung gefunden wurde wird die jeweilige Funktion aufgerufen. Wird keine Übereinstimmung gefunden, wird mittels des Makros `php_error` ein PHP-Laufzeitfehler erzeugt. Da diese Fehler von Turpitude von der Funktion `turpitude_error_cb` als `PHPEvalException` an die JVM weitergeleitet werden stellen sie ein probates Mittel zum Abbruch der PHP-Programmausführung dar, und können gefahrlos eingesetzt werden. Diese Gefahrlosigkeit rührt daher, dass Turpitude alle Aufrufe von Zend-Funktionen durch einen speziellen Try-Catch Block einschliesst:

```
zend_first_try {
    // Aufrufe
} zend_catch {
    // Fehlerbehandlung
} zend_end_try();
```

Listing 4.6: Zend Try-Catch Block

Ohne diese Zeilen würden schwerwiegende PHP-Fehler einen sofortigen Abbruch des laufenden Prozesses durch den C-Systemaufruf `_exit` mit dem Fehlercode 255 zur Folge haben, was in diesem Falle nicht nur den PHP-Interpreter, sondern auch die gesamte einbettende Java Virtual Machine beenden würde. Im Catch-Block findet die Fehlerbehandlung statt, im Normalfall wird hier eine Java-Exception mit dem Text der PHP-Fehlermeldung geworfen und der Interpreter beendet. Leider führen auch einige PHP-Funktionen (genauer: alle Funktionen, die die Zend-Funktion `zend_bailout` aufrufen) zu einem Sprung in den Catch-Block, weswegen in der oben beschriebenen Funktion `turpitude_error_cb` eine globale Variable gesetzt werden muss, die anzeigt, ob ein echter Fehler aufgetreten ist.

4.3.5 Java-Klassen in PHP

Die erste Methode des TurpitudeEnvironment, die implementiert wurde, trägt den Namen `findClass()`. In der zugehörigen C-Funktion wird zunächst lediglich überprüft, ob die Anzahl der Parameter stimmt, und ob der übergebene Parameter vom Typ `String` ist. Dann wird die zur Klasse `TurpitudeJavaClass` gehörende Funktion `make_turpitude_jclass_instance` aufgerufen, die eine TurpitudeJavaClass zurückgibt. Um eine Instanz dieser Klasse erzeugen zu können, musste diese allerdings zunächst dem Interpreter bekannt gemacht werden. Dies geschah analog zum Bekanntmachen von TurpitudeEnvironment. `make_turpitude_jclass_instance` selbst erwartet drei Argumente: eine `jclass` der zu repräsentierenden Klasse, deren Klassennamen und einen Pointer auf den zval, der das Objekt enthalten soll. Da die TurpitudeJavaClass tatsächlich Attribute (den Klassennamen) enthalten soll, muss die Zend-HashTable initialisiert werden, die diese Attribute enthalten soll. Dies geschieht mittels der Zend-Funktion `object_init_ex`, die neben einem zval-Pointer den class_entry entgegennimmt. Nachdem im entsprechenden `object_struct` die jclass gesetzt wurde, konnte schlussendlich über die Funktion `zend_hash_update` der Klassenname in die HashTable eingefügt werden. Ein var_dump einer solchen Klasse sah nun wie folgt aus:

```
object(TurpitudeJavaClass)#4 (1) {
  ["ClassName"]=>
    string(16) "java/lang/String"
}
```

Listing 4.7: Dump einer TurpitudeJavaClass

Um aus Klassen Objekte erzeugen zu können, muss ein Konstruktor aufgerufen werden. Konstruktoren werden im JNI wie ganz normale Methoden behandelt, nur dass der Methodenname stets `<init>` ist. Um mittels JNI eine Methode aufrufen zu können, muss diese zuerst in Form einer Variablen des Types `jmethodID` vorhanden sein, die wiederum mit der JNI-Umgebung und einer `jclass` erzeugt werden kann. Hierzu muss neben dem Methodennamen allerdings auch die Signatur der aufzufindenden Methode angegeben werden, und zwar in einer speziellen Syntax. Diese zu erläutern würde an dieser Stelle den Rahmen sprengen, deswegen hier nur der Verweis auf die JNI-Dokumentation, zu finden unter [JNI06]. Um eine jmethodID für den PHP-Anwender hantierbar zu machen wurde die Klasse TurpitudeJavaMethod eingeführt - wieder analog zu den vorherigen Klassen. Eine TurpitudeJavaMethod hat weder Attribute noch eigene Methoden. Sie wird ausschliesslich als Parameter in Methoden anderer Klassen benötigt. Erzeugt wird ein solches Objekt mit der Methode `findMethod()` der Klasse TurpitudeJavaClass, die zwei String-Argumente erwartet: den Methodennamen und die JNI-kodierte Methodensignatur. Die Validität dieser beiden Argumente wird nicht explizit geprüft. Wird keine entsprechende Methode gefunden, wird ein PHP-Fehler erzeugt. Die TurpitudeJavaClass wird durch die

Funktion `make_turpitude_jmethod_instance` erzeugt. Sie speichert neben ihrem Namen und ihrer Signatur noch einen Indikator, der anzeigt, ob sie statisch aufgerufen werden kann. Außerdem wird im object_struct die jclass und der im enum `turpitude_java_type` definierte Typ des Rückgabewerts gespeichert. Dieser wird durch Parsen der Signatur ermittelt und später beim Aufruf der Methode benötigt. Neben den primitiven Java-Datentypen wie `int`, `boolean` und `double` werden noch `void` und vor allem `Object` unterschieden, wobei der eigentliche Typ des Objektes keine Rolle spielt. Neben der Methode `findMethod()` wurde in der TurpitudeJavaClass noch die Methoden `findStaticMethod()` für statisch aufrufbare Methoden und `findConstructor()`, die dem Anwender die Übergabe von "<init>" als Methodenname erspart, definiert.

4.3.6 Java-Objekte in PHP

Derart erstellte Konstruktoren konnten nun zur Erzeugung von Java-Objekten genutzt werden. Diese sollten in PHP durch die Klasse `TurpitudeJavaObject` dargestellt werden, und so musste diese zunächst dem Interpreter auf bewährte Weise bekanntgemacht werden. Neben dem `jobject`, welches das Java-Objekt auf C-Ebene repräsentiert, wird im object_struct des TurpitudeJavaObject noch die `jclass` mitgespeichert, was später klassenspezifische Operationen erleichtern soll. Das Objekt selbst wird schon im call-Handler der TurpitudeJavaClass erzeugt, und zwar mittels der JNI-Funktion `NewObject`, falls der Konstruktor keine Argumente erfordert, und der Funktion `NewObjectA`, falls er Argumente erwartet. Alle JNI-Funktionen mit dem Suffix "A" erwarten die an die Java-Methode weiterzureichenden Argumente als Array des Typs `jvalue`, einem Union aller JNI-Typen. Um ein solches Array füllen zu können wurde die Funktion `zval_to_jvalue` geschrieben, die über den Typen des zval ermittelt, welches Attribut des jvalue befüllt werden soll. Leider enthält ein jvalue keinerlei Typinformation über den gespeicherten Wert, was später bei der Rückumwandlung eines jvalue in einen zval zu Problemen führen könnte. Im Gegensatz zur oben beschriebenen Funktion `zval_to_jobject` wandelt `zval_to_jvalue` skalare PHP-Typen nicht in deren Objektäquivalente in Java, sondern in die entsprechenden primitiven Java-Typen um. Somit können auch Konstruktoren (und später auch Methoden) aufgerufen werden, die solche Typen als Argumente verlangen, ohne sich auf das Autoboxing* der JVM verlassen zu müssen. Um aus einem jobject eine Instanz der Klasse TurpitudeJavaObject zu konstruieren wird es, zusammen mit der zugehörigen Klasse - sowohl die jclass als auch die TurpitudeJavaClass - an die Funktion `make_turpitude_jobject_instance` übergeben, in der die eigentliche Instanz angelegt und die Turpitude-Klasse als Attribut "Class" hinzugefügt wird. Eine so erzeugte Repräsentation eines Java-Objektes vom Typ java.util.Date sieht für den PHP-Anwender wie folgt aus:

*Ermöglicht das Benutzen von Hüllenklassen (wie `java.lang.Integer`) in der von primitiven Datentypen (wie `int`) gewohnten Form, siehe [**?**].

```
object(TurpitudeJavaObject)#6 (1) {
  ["Class"]=>
  object(TurpitudeJavaClass)#4 (1) {
    ["ClassName"]=>
    string(14) "java/util/Date"
  }
}
```

Listing 4.8: Dump eines TurpitudeJavaObject

4.3.7 Java-Methodenaufrufe in PHP

Nun konnte endlich der Aufruf von Java-Methoden implementiert werden. Der call-Handler des TurpitudeJavaObjects überprüft zuerst, ob es sich bei der aufgerufenen Methode um `javaInvoke` handelt. Ist dies der Fall, wird die Methode des Java-Objektes aufgerufen, die im ersten Parameter übergeben wurde. Danach wird - ähnlich wie bei Konstruktoren - unterschieden, ob neben der aufzurufenden Methode noch andere Argumente übergeben wurden. In diesem Fall wird - wieder wie bei Konstruktoren - ein Array aus jvalues erstellt und je nach Typ des Rückgabewertes der Methode die entsprechende JNI-Funktion (Call<type>MethodA) aufgerufen. Wurden außer der Methode keine weiteren Parameter mitübergeben, wird eine entsprechende Funktion der Signatur Call<type>Method (ohne das nachgestellte A) aufgerufen. Der Rückgabewert der aufgerufenen JNI-Funktion wird in einem jvalue gespeichert, der zusammen mit dem Typen des Rückgabewertes der Funktion `jvalue_to_zval` übergeben wird. Diese erzeugt letztendlich den zval, der an PHP zurückgegeben wird. Im Zuge der Implementierung von Methodenaufrufen bei Objekten wurde auch gleich der Aufruf statischer Methoden bei Klassen implementiert. Dieser funktioniert exakt gleich, nur dass JNI-Funktionen der Signatur CallStatic<Type>Method(A) aufgerufen werden müssen. Zu diesem Zweck wurde die `TurpitudeJavaClass` um die Methode `invokeStatic` erweitert.

Da das Erzeugenmüssen von Methoden bevor man sie aufrufen kann doch eher mühsam erscheint wurde call-Handler noch derart erweitert, dass wenn keine der explizit definierten Methoden aufgerufen wurde nicht einfach ein Fehler erzeugt wird, sondern es wird versucht eine Methode gleichen Namens zu finden und diese aufzurufen. Dazu muss der Anwender allerdings als erstes Argument die Methodensignatur angeben, da es das JNI leider nicht ermöglicht alle Methoden eines Namens zu finden, um dann eventuell anhand der weiteren Argumente die richtige Methode zu finden. Ist der erste Parameter also ein String wird versucht eine Methode zu finden die den Namen der in PHP aufgerufenen Methode trägt, und die den übergebenen String als Signatur hat. Ist dies der Fall wird einfach eine Instanz von `TurpitudeJavaMethod` erzeugt und an Stelle des Signaturstrings in das Argumentenarray geschrieben. Nun muss nur noch die Funktion aufgerufen werden die auch die mittels `javaInvoke`

aufgerufenen Methoden abhandelt. Folglich kann die Java-Methode `toString()` der Klasse `java.util.Date` auf diese beiden Arten aufgerufen werden:

```
$method = $class->findMethod('toString', '()Ljava/lang/String;');
$string = $date->javaInvoke($method);
oder
$string = $date->toString('()Ljava/lang/String;');
```

Listing 4.9: Zwei Arten die gleiche Methode aufzurufen

4.3.8 Zugriff auf Java-Attribute in PHP

Weiterhin sollte es dem Anwender ermöglicht werden auf Attribute eines Objektes zuzugreifen. Für Attribute müssen mit dem JNI - wie bei Klassen und Objekten auch - Handler erzeugt werden. Allerdings wurde hier auf eine eigene PHP-Klasse verzichtet, stattdessen wurden einfach dem TurpitudeJavaObject zwei weitere Methoden hinzugefügt: `javaGet()` und `javaSet()`. `javaGet()` erwartet analog zu Methoden zwei Argumente, den Namen des Attributes und dessen Signatur, beides als Strings, letzteres wieder JNI-kodiert. Wie bei Methodensignaturen auch wird der Feldsignaturstring geparst, um später dem Rückgabewert den richtigen Typen zuweisen zu können, aber auch weil um ein Attribut eines Java-Objektes auszulesen die richtige JNI-Funktion aufgerufen werden muss. Diese Funktionen haben die Signatur Get<Type>Field, und erwarten alle das Objekt dessen Attribut ausgelesen werden soll, sowie eine sogenannte `jfieldID`, die zuvor mittels der Funktion `GetFieldID` generiert werden muss. Wie bei Methodenaufrufen der Rückgabewert wird hier der Wert des Attributes in einem jvalue gespeichert, der wieder in einen zval umgewandelt wird. `javaSet()` erwartet neben den Argumenten die auch `javaGet()` erwartet noch den Wert auf den das Attribut gesetzt werden soll, dafür gibt diese Methode natürlich keinen Wert zurück. Auch `javaSet()` entscheidet anhand der Signatur welche JNI-Funktion zum Setzen des Attributes aufgerufen werden soll und versucht ebenfalls anhand des übergebenen Attributnamens und -signatur die entsprechende jfieldID zu finden. Die entsprechenden JNI-Funktionen haben die Signatur `Set<Type>Field`.

Analog zu Feldern von Instanzen sollte auch auf statische Felder in Java-Klassen zugegriffen werden können. Hierzu wurden in der `TurpitudeJavaClass` zwei Methoden implementiert, `getStatic()` und `setStatic()`. Wie bei nicht-statischen Attributen auch, müssen zum Setzen beziehungsweise Auslesen je nach Feld-Typ spezielle JNI-Funktionen aufgerufen werden, diese haben die Signatur `Get-` beziehungsweise `SetStatic<Type>Field`. Die Umwandlung des zu setzenden Wertes beziehungsweise des Rückgabewertes geschieht analog zu nicht-statischen Attributen.

4.3.9 Java-Exceptions in PHP

Nachdem man nun Klassen konstruieren, deren Methoden aufrufen und auf deren Attribute zugreifen konnte, wurden die Möglichkeiten der Fehlerbehandlung verbessert.

Java arbeitet hauptsächlich mit Exceptions, und auch die Turpitude-intern erzeugten Fehler treten in der JVM als Exceptions auf. Folglich mussten dem Anwender Werkzeuge zur Hand gegeben werden, geworfene Exceptions in PHP zu fangen und zu behandeln, und selbst Java-Exceptions zu erzeugen. Das Werfen von Exceptions kann der Anwender mittels der Methoden `throw` und `throwNew` des TurpitudeEnvironment bewerkstelligen. Mit `throw` können Instanzen der Klasse `java.lang.Throwable` geworfen werden, die der Anwender zuvor wie oben beschrieben erzeugt hat. `throwNew` hingegen erwartet zwei Parameter, zum einen den Namen der zu werfenden Exception, zum anderen die Nachricht, die die Exception enthalten soll. Will der Anwender also komplizierte Exceptions werfen, die zusätzlich zur eigentlichen Nachricht noch andere Informationen tragen soll, wird er `throw` nutzen, für den Großteil der Fälle hingegen reicht `throwNew` völlig aus. Um Exceptions auch fangen zu können wurde die Methode `exceptionOccurred` eingeführt, die die gleichnamige JNI-Funktion aufruft, und die zuletzt geworfene und noch nicht gefangene Exception zurückgibt. Wird keine solche Exception gefunden wird `null` zurückgegeben, derart gefundene Exceptions verbleiben allerdings in diesem Status. Um erfolgreich behandelte Exceptions aus diesem Zustand zu entfernen wurde die Methode `exceptionClear` implementiert, die ebenfalls keine Parameter entgegennimmt, und ebenfalls die gleichnamige JNI-Funktion aufruft.

Alle hier beschriebenen Methoden erzeugen PHP-Laufzeitfehler sollte beispielsweise die Anzahl der übergebenen Argumente nicht stimmen, oder falschen Typs sein sollte. Wichtig ist noch zu bemerken, dass sowohl das Aufrufen von Java-Methoden, als auch das Lesen beziehungsweise Schreiben von Attributen auch auf nicht-öffentlichen Methoden und Attributen funktioniert. Einerseits da JNI keinerlei Möglichkeiten bietet Zugriffsmodifikatoren zu erkennen, andererseits da man eigentlich davon ausgehen können sollte, dass ein Anwender nicht böswillig versucht Inkonsistenzen in den von ihm benutzten Java-Objekten zu erzeugen..

4.3.10 Der JSR 223 ScriptContext

Mit der geleisteten Vorarbeit war es nun einfach die Anforderungen des JSR nach einem ScriptContext umzusetzen. Die von der `PHPScriptEngine` erweiterte Klasse `AbstactScriptEngine` bringt eine Kontextimplementation namens `SimpleContext` mit. Dieser Kontext wird beim Aufruf der nativen Methode `exec` des PHPCompiledScripts als jobject übergeben und global gespeichert. An diesen Kontext kommt der Anwender in PHP über die Methode `getScriptContext` des TurpitudeEnvironment. Im entsprechenden Handler wird die Klasse des ScriptContext ermittelt, damit eine TurpitudeJavaClass erstellt, und mit dem Kontext-jobject zu einer Instanz von TurpitudeJavaObject vereinigt. Somit kann der Anwender einfach die entsprechenden Java-Methoden dieses Objektes aufrufen, um an die im Kontext enthaltenen Daten zu gelangen. Getestet wurde dies mit einem Programm das einen java.util.StringBuffer erzeugt, in den Kontext einsetzt, und dann ein PHP-Skript ausführt, das diesem

StringBuffer einen Text anhängt. Nachdem das Skript ausgeführt wurde enthält der StringBuffer sowohl die per Java, als auch die in PHP angehängten Zeichen.

4.3.11 Implementieren des Interfaces Invocable

Bis zu diesem Zeitpunkt konnte die Java-Applikation, die das PHP-Skript einbettet, dieses nur ausführen, es konnten keine gezielten Aufrufe von Funktionen und Methoden abgesetzt werden, obwohl diese Fähigkeit eine Bibliothek wie Turpitude erst wirklich interessant macht. Deswegen wurde das für diese Anwendungsfälle vorgesehne JSR 223-Interface `javax.script.Invocable` implementiert, das genau an diesem Punkt ansetzt und Abhilfe schaffen soll. Leider zeigte sich auch hier, dass die JSR 223-Spezifikation an einigen Stellen etwas undurchdacht wirkt: laut Spezifikation sollte die jeweilige ScriptEngine das Interface implementieren. Nun implementiert die PHPScriptEngine aber schon das Interface `javax.script.Compilable`, es ist also nicht eindeutig auf welchem Skript die Aufrufe durchgeführt werden sollen, und die Spezifikation schweigt sich zu diesem Thema ebenfalls aus. Um dieses Problem zu lösen wurde zunächst das PHPCompiledScript derart abgewandelt, dass dieses `Invocable` implementiert, da der Kontext hier eindeutig ist. Der PHPScriptEngine wurde ein weiteres Attribut `lastScript` hinzugefügt, in welchem das zuletzt übersetzte PHP-Script gespeichert wird. Dieses Attribut wird bei jedem Aufruf der Methode `compile` gesetzt, womit auch ausgeführte Skripte abgedeckt werden. Die Invocable-Methoden der PHPScriptEngine leiten nun einfach alle Aufrufe an die entsprechenden Methoden des `lastScript` weiter.

`javax.script.Invocable` beinhaltet vier Methoden, von denen sich jeweils zwei sehr ähnlich sind. Die Methode `invokeFunction` ruft sogenannte `top-level functions` des Skriptes auf, also im globalen Scope definierte Funktionen. Da die in der nativen Methode `execute`, die wie oben beschrieben beim Ausführen eines übersetzten Skriptes aufgerufen wird, benutzte Zend-API Funktion `zend_call_function` ein auszuführendes Op-Array erwartet, und da das Finden des richtigen Abschnittes im Op-Array des PHPCompiledScripts sehr viel Aufwand bedeutet hätte, wurde ein anderer Weg gewählt: die Funktion `call_user_function`, die direkt eine im PHP-Skript vorhandene Funktion aufruft. Damit dies allerdings funktionieren kann, muss zunächst in den sogenannten *Executor Globals*, erreichbar über das Makro `EG`, das Feld `active_op_array`, welches das gerade aktive Op-Array enthalten soll, auf das im PHPCompiledScript gespeicherte Op-Array gesetzt werden. Um den Rückgabewert einer Funktion an die Java-Applikation weiterreichen zu können wurde noch ein zval initiert, und ein Pointer auf dessen Pointer wurde in den *Executor Globals* an die Stelle `return_value_ptr_ptr` geschrieben. Nach dem erfolgreichen Ausführen der Funktion enthält dieser zval den Rückgabewert, und konnte folglich wie alle anderen Rückgabewerte behandelt und an die JVM zurückgegeben werden. Um die von Java aus als Array von Objekten übergebenen Funktionsargumente an die PHP-Funktion weiterleiten zu können wird Speicher für ein Array von zval-Pointern reser-

viert, und dieses Array mittels der Funktion `jobject_to_zval` gefüllt. Das so erstellte Array wird schlussendlich der die PHP-Funktion aufrufenden Zend-API Funktion `call_user_function` übergeben, zusammen mit einem den Funktionsnamen enthaltenden zval vom Typ String, der zu nutzenden Funktionstabelle und dem Pointer auf den Rückgabewert. Gibt `call_user_function` nun ein `FAILURE` zurück ist ein Fehler aufgetreten, und es wird eine Java-Exception geworfen, ansonsten hat das Aufrufen funktioniert. Ganz ähnlich wurde die Invocable-Methode `invokeMethod` implementiert, allerdings musste zunächst das PHPObject noch um ein Feld erweitert werden, welches eine Referenz auf den entsprechenden zval innerhalb des PHP-Interpreters hält. Hierzu wurde - ganz ähnlich dem Speichern des Op-Arrays im PHPCompiledScript - ein `java.nio.ByteBuffer` verwendet, in welchen einfach der zval-Pointer gespeichert wird. Beim Aufrufen einer Methode wird dieser Zeiger nun ausgelesen und der Funktion `call_user_function` als zweiter Parameter übergeben. Da so stets die richtige Methode der richtigen Instanz aufgerufen wurde, beschreibt dieser Parameter offensichtlich das Objekt, auf dem die Funktion aufgerufen werden sollte. Ansonsten verläuft ein Methodenaufruf analog zu einem Funktionsaufruf.

Für den Java-Entwickler stellt sich das Aufrufen von Funktionen und Methoden also wie folgt dar:

```
Invocable inv = (Invocable)engine.compile(/* PHP-Source */);
Object phpobj = inv.invokeFunction("foo", "function call");
Object retval = inv.invokeMethod(phpobj, "bar", "method call");
```

Listing 4.10: Aufrufen von PHP-Funktionen und -Methoden

Zunächst muss natürlich PHP-Quelltext zu einem PHPCompiledScript übersetzt, und dieses dann auf Invocable gecastet werden. Dann wird die PHP-Funktion “foo“ aufgerufen, und ihr ein String als einziges Argument übergeben. Diese Funktion gibt in diesem Fall ein PHPObject zurück, auf welchem in der letzten Zeile die Methode “bar“, wieder mit einem String als einzigem Argument, aufgerufen wird.

Unter Zuhilfenahme von `invokeMethod` konnten schließlich die anderen beiden Methoden des Interfaces `Invocable` implementiert werden, die beide den Namen `getInterface` tragen. Beiden Varianten dieser Methode wird eine Java-Klasse oder -Interface übergeben, und zurückgegeben werden muss eine Instanz dieser Klasse, deren Methoden in der Skriptsprache implementiert sind. Der Unterschied zwischen beiden Methoden ist, dass einer von beiden zusätzlich ein Objekt übergeben werden kann, welches als implementierendes Objekt genutzt werden soll. Dies macht im Falle von Turpitude nur Sinn, wenn das übergebene Objekt ein PHPObject ist, von dem der Anwender weiß, dass es das geforderte Java-Interface implementiert. Letztere Methode wurde zuerst implementiert, da hierzu schon alle Voraussetzungen geschaffen waren, und keine einzige Zeile C-Code mehr geschrieben werden musste. Als erstes wird in der Methode geprüft, ob das übergebene Objekt wirklich eine Instanz von PHPObject ist. Danach wird eine Instanz der Klasse `java.lang.reflect.Proxy` erzeugt, die innerhalb Javas als ganz normale Instanz der übergebenen Klasse auf-

tritt, intern aber alle Methodenaufrufe an einen sogenannten `InvocationHandler` weiterreicht. Ein InvocationHandler ist ein Interface, welches in diesem Falle von der Klasse `PHPInvocationHandler` implementiert wird, die im Konstruktor ein PHPCompiledScript, in dessen Kontext die Methodenaufrufe stattfinden sollen, und ein PHPObject auf dem die Methodenaufrufe durchgeführt werden sollen erwartet. Die eigentlichen Methodenaufrufe werden sämtlich durch die oben beschriebene Methode `invokeMethod` des PHPCompiledScripts erledigt. Die Implementierung der Variante von `getInterface` die kein Objekt mitbringt stellte allerdings ein Problem dar - wie sollte ein das Java-Interface implementierendes PHP-Objekt gefunden werden? Auch hier schweigt sich die JSR 223-Spezifikation leider aus, so wurde nach einigem Überlegen entschlossen einzig auf den Namen der Klasse zu prüfen. Folglich wurde dem PHPCompiledScript eine weitere native Methode namens `createInstance` hinzugefügt, die als einziges Argument den Klassennamen in Form eines Strings erwartet. Sie sollte versuchen eine PHP-Klasse des übergebenen Namens zu finden, und eine Instanz dieser Klasse neu zu erzeugen. Da Invocable hierzu keine Parameterübergabe vorsah, konnten auf diese Weise nur PHP-Objekte erzeugt werden, die einen Konstruktor ohne Argumente besitzen, ist dies nicht der Fall wird ein Laufzeitfehler erzeugt. Zur Implementierung dieser Methode wurde die Zend-Funktion `zend_lookup_class` gefunden, die zu einem Klassennamen einen `zend_class_entry` zurückgibt. Mit diesem Klasseneintrag kann mittels der schon vorher genutzten Funktion `object_init_ex` eine Instanz der Klasse erzeugt werden, die dann an die JVM als PHPObject zurückgegeben wird. Wird auf diese Weise ein PHPObject gefunden, wird mit diesem als zweitem Argument einfach die schon fertige Variante der Interfacemethode aufgerufen. Da PHP keine Punkte in Klassennamen erlaubt, und zumindest in Version 5 keine Namespaces kennt, wurde beschlossen den sogenannten *SimpleName*, den Namen also wie er in der Quelltextdatei vorkommt, der zu findenden Klasse zu benutzen. Abbildung 4.6 soll den Ablauf eines solchen Aufrufes einer in PHP implementierten Methode darstellen.

4.3.12 Java-Arrays in PHP

Eigentlich wären zu diesem Zeitpunkt alle Anforderungen an die Bibliothek erfüllt, in vielen Bereichen sogar übererfüllt gewesen, aber da während der Entwicklung festgestellt wurde, dass sowohl innerhalb als auch auch außerhalb von 1&1 erhebliches Interesse an Turpitude besteht, wurde beschlossen einige Erweiterungen vorzunehmen, um Entwicklern ohne intimes Wissen über die Bibliothek das Entwickeln mit Turpitude zu vereinfachen. Zum einen mussten einige Randbedingungen beim Aufrufen von Java-Methoden abgefangen werden, die sonst zu unerwarteten Ergebnissen führen konnten, zum anderen sollten Java-Arrays in PHP nicht nur als TurpitudeJavaObject dargestellt, sondern ein intuitiverer Umgang mit ihnen ermöglicht werden.

Um Java-Arrays angemessen repräsentieren zu können musste allerdings zunächst eine neue Klasse eingeführt werden, das `TurpitudeJavaArray`. Nach bewährtem

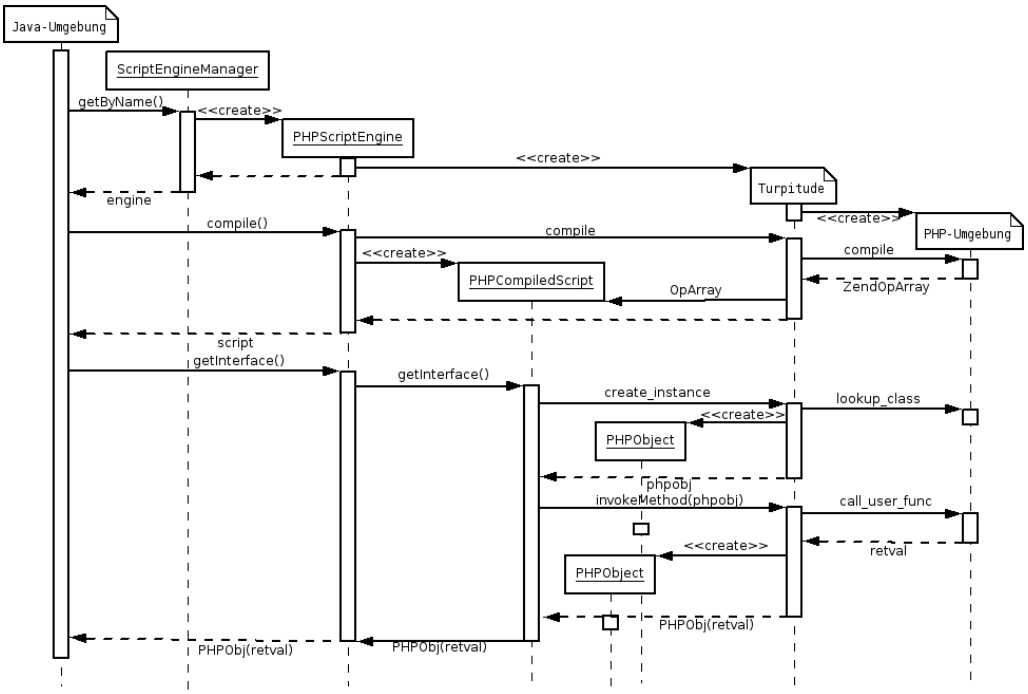

Abbildung 4.6: Ablauf eines in PHP implementierten Java-Methodenaufrufes

Muster wurden dieser Klasse drei Methoden gegeben, zum einen `getLength`, die die Länge des Arrays zurückgibt, und zum anderen `get` und `set`, um auf Elemente des Arrays zuzugreifen. Für den Zugriff auf Java-Arrays bietet das Java Native Interface Funktionen an, die den Inhalt des Arrays in einen zusammenhängenden Speicherbereich kopieren, diese Funktionen haben die Signatur `Get<Type>ArrayElements`. Diese Speicherbereiche können dann in C ausgelesen und verändert werden. Derartige Änderungen können beim Freigeben der Speicherbereiche entweder verworfen oder gespeichert werden. Da diese Funktionen allerdings - wie fast alle JNI-Funktionen - für jeden Typ andere sind, musste das oben beschriebene Parsen von JNI-Typstrings derart erweitert werden, dass nicht nur Arrays sondern auch die verschiedenen Typen von Arrays erkannt werden. Das Freigeben der Speicherbereiche geschieht mittels der Funktionen `Release<Type>ArrayElements`, deren letzter Parameter angibt, ob die Änderungen übernommen oder verworfen werden sollen. Eine Ausnahme zu dieser Regel bilden Arrays von Objekten, auf deren Elemente direkt mittels der Funktionen `GetObjectArrayElement` und `SetObjectArrayElement` zugegriffen werden kann. Arrays in PHP sind eigentlich HashMaps, sie können also nicht nur numerische Werte, sondern jeglichen skalaren Typ als Index haben, folglich mussten für alle Zugriffe auf das `TurpitudeJavaArray` Tests eingebaut werden die nur numerische Indizes zulassen. Somit konnte zwar von PHP aus auf Java-Arrays zugegriffen werden, allerdings eben nur mit den beschriebenen Methoden. Ein PHP-Anwender jedoch ist gewohnt mit dem Klammern-Operator `[index]` auf Arrays arbeiten zu können. Da-

mit auf eine Klasse in PHP wie auf ein Array zugegriffen werden kann, bietet die Zend-Engine die Möglichkeit das interne Interface `ArrayAccess` zu implementieren, welches auf C-Ebene unter dem Symbol `zend_ce_arrayaccess` verfügbar ist. Dieses Interface enthält die Methoden `offsetGet` und `offsetSet` um auf Arrays zuzugreifen, `offsetUnset` um Zeilen aus dem Array zu löschen und `offsetExists` um anzuzeigen ob zu einem bestimmten Schlüsselwert ein Eintrag vorhanden ist. Die zu implementierenden Methoden wurden als *built-in functions* in das bei der Initialisierung der Klasse übergebene Funktionsarray eingetragen. Die Implementierung von `offsetUnset` erwies sich als trivial, da aus einem Java-Array keine Zeilen gelöscht werden können, es wird also lediglich ein PHP-Laufzeitfehler geworfen. Aufrufe von `offsetGet` und `offsetSet` werden lediglich an die bereits implementierten Methoden `get` und `set` weitergereicht.

```
$array = $instance->getStringArray('()[Ljava/lang/String;');
$length = $array->getLength();
$val = $array->get(0);
$array[0] = 'Wert';
```

Listing 4.11: Zugriff auf ein Java-Array

Es wurde beschlossen ein weiteres PHP-internes Interface zu implementieren, um dem PHP-Anwender den Umgang mit Java-Arrays weiter zu vereinfachen. Dieses Interface trägt den Namen `IteratorAggregate`, und beschreibt eine einzige Methode, `getIterator`, allerdings muss sie ein Objekt zurückgeben, das das PHP-Interface `Iterator` implementiert. Ist dies der Fall kann der Anwender nicht nur mittels der `Iterator`-Methoden auf dem Array arbeiten, sondern auch mit dem `foreach`-Operator. Nachdem das `TurpitudeJavaArray` so verändert wurde, dass es zusätzlich das Interface `IteratorAggregate`, auf C-Ebene unter dem Symbol `zend_ce_aggregate` zu finden, implementiert wurde also eine weitere Klasse eingeführt, der `TurpitudeJavaArrayIterator`. Ein `Iterator` muss folgende Methoden implementieren: `rewind`, `valid`, `key`, `current` und `next`, die alle keinerlei Argumente erwarten, und deren Funktionalität sich aus den Namen erschließen sollte. Intern hält der `TurpitudeJavaArrayIterator` neben dem `TurpitudeJavaArray` über welches er iterieren soll noch einen Integer vor, der den gerade aktuellen Index in das Array speichert. `rewind` setzt diesen Index auf 0 zurück, während ein Aufruf von `next` ihn um eins erhöht. `valid` gibt genau dann `true` zurück wenn der Index größer als 0 und kleiner als die Anzahl der Elemente des Arrays ist. `key` gibt den aktuellen Index, `current` das aktuelle Array-Element zurück, wozu einfach die `get`-Methode des entsprechenden Arrays aufgerufen wird. Hiernach konnte also in PHP folgendermaßen über Java-Arrays iteriert werden:

```
$iterator = $array->getIterator();
while ($iterator->valid()) {
    $row = $iterator->current();
    $ley = $iterator->key();
    printf("key: %d, val: %s\n", $key, $row);
    $iterator->next();
}
```

Listing 4.12: Iterieren über ein Java-Array

Laut Zend-Dokumentation sollte es eigentlich so sein, dass man über alle Objekte die entweder `Iterator` oder `IteratorAggregate` implementieren zusätzlich mit dem `foreach`-Operator itererieren können sollte. Dies stimmt wohl auch für im Userland geschriebene Klassen, allerdings verursachte der Versuch auf diese Weise über `TurpitudeJavaArray` oder `TurpitudeJavaArrayIterator` Objekte lediglich Abstürze und Fehlermeldungen. Nach ausführlichem Lesen von Zend-Quelltext wurde entdeckt, dass jeder `zend_class_entry` ein Feld enthält, in dem ein Pointer auf eine Funktion gespeichert wird, die wiederum einen Pointer auf ein struct vom Typ `zend_object_iterator` zurückgeben muss. Solch ein struct besteht im Wesentlichen aus einem Voidpointer (`void*`), in dem Daten gespeichert werden können, und einem Pointer auf ein weiteres struct, welches die Iteratorfunktionen enthalten muss. Folglich müssen interne Klassen nicht nur die Methoden implementieren mit denen der Benutzer umgeht, sondern zusätzlich noch Funktionen die die Zend-Engine benutzt um mit `foreach` über die Objekte zu iterieren. Nachdem sowohl die interne `get_iterator` Funktion als auch die internen Iteratorfunktionen, die fast eins-zu-eins Kopien der bereits implementierten Iteratormethoden sind - geschrieben waren konnte nun auch mit dem `foreach`-Operator auf Java-Arrays zugegriffen werden:

```
$iterator = $array->getIterator();
foreach ($iterator as $key => $val) {
    printf("key: %d, val: %s\n", $key, $row);
}
```

Listing 4.13: Java-Array und foreach

4.3.13 Java-Methodenaufrufe, die 2.

Eigentlich konnten zu diesem Zeitpunkt wie oben beschrieben schon Java-Methoden aus PHP heraus aufgerufen werden, allerdings hatte diese Implementierung zwei Nachteile: Einerseits musste der PHP-Entwickler die genaue JNI-Signatur der Methode kennen, andererseits - und dies war der erheblich störendere Nachteil - konnten nur Methoden aufgerufen werden die tatsächlich in der Java-Klasse definiert sind, auf der `findMethod()` aufgerufen wurde, also keine Methoden die in Oberklassen definiert wurden. Folglich musste der PHP-Anwender unter Umständen mehrere Java-Klassen

in PHP erstellen um unterschiedliche Methoden auf ein und demselben Java-Objekt aufzurufen. Zwar mag es unter bestimmten Umständen wünschenswert sein die genaue Methode angeben zu können die aufgerufen werden soll, allerdings will man in den meisten Fällen doch von den Java-Sprachfeatures wie Polymorphismus profitieren. Aus diesem Grund wurde beschlossen die Methodenaufrufe über `javaInvoke()` nicht zu verändern, aber die Methodenaufrufe "direkt" auf dem Objekt so anzupassen, dass sie sich wie ein normaler Java-Methodenaufruf verhalten. Dies zu verwirklichen wäre mit JNI-Mitteln alleine sehr schwer gewesen, weswegen auf die Java Reflection API zurückgegriffen werden musste.

Zu diesem Zweck wurde eine neue Java-Klasse erstellt, der `ReflectHelper`, die drei statische Methoden enthält: `signatureMatchesArguments()`, `findMethod()` und `callMethod()`. `callMethod()` wird vom nativen Code aus aufgerufen, und erwartet als Parameter ein Objekt, einen Methodennamen und ein Array, welches Methodenargumente enthält, und versucht dann mit Hilfe der anderen beiden Methoden eine zum Namen und den Argumenten passende Methode auf dem übergebenen Java-Objekt zu finden und aufzurufen, und das Ergebnis dieses Aufrufes zurückzugeben. Um die passende Methode zu finden nutzt sie `findMethod()`, welche aus der Klasse die verfügbaren Methoden ausliest, und zunächst anhand des aufzurufenden Methodennamens, und schliesslich mit Hilfe von `signatureMatchesArguments()` anhand der Argumententypen versucht die passende Methode zu finden. Eventuelle beim Aufruf auftretende Exceptions werden von `callMethod()` einfach an den Aufrufer weitergeleitet, welcher diese Fehler dann selbst behandeln muss.

Nun musste der native Code des TurpitudeJavaObjects derart angepasst werden, dass jeder Methodenaufruf der nicht die explizit definierten Methoden betrifft, in einen Aufruf von `callMethod()` umgewandelt wird. Hierzu wurde eine C-Funktion geschrieben, in der die zum Aufruf von `callMethod()` nötigen jclass und jmethodID erzeugt, und die in PHP übergebenen Parameter in ein Java-Array gewandelt werden. Ist dies geschehen wird `callMethod()` aufgerufen, und das Ergebnis, wie bei jedem anderen Java-Methodenaufruf auch, in einen zval umgewandelt. Diese C-Funktion gibt "true" zurück wenn die Methode gefunden und aufgerufen werden konnte, ansonsten wird "false" zurückgegeben. Dieser Bool'sche Wert wird dann im `__call`-Callback des TurpitudeJavaObjects ausgewertet, und es wird entweder das Ergebnis zurückgegeben, oder ein PHP-Laufzeitfehler erzeugt. Der eigentliche Aufruf der Java-Methode geschieht über die Methode `invoke()` der gefundenen `Method`-Klasse, die neben dem Objekt auf dem die Methode aufgerufen werden soll noch die zu übergebenden Argumente entgegennimmt.

Nun konnten Java-Methodenaufrufe in PHP auf die Übergabe der genauen Signatur verzichten, und es konnten auch Methoden aufgerufen werden die in übergeordneten Klassen definiert wurden:

```
$date = $obj->getDate();
$string = $date->toString();
```

Listing 4.14: Verbesserter Aufruf einer Java-Methode

Allerdings brachte diese Art des Methodenaufrufes auch Nachteile mit sich, die zum Teil mit der Reflection-API von Java, und zum Teil mit Turpitude selbst zusammenhängen. So werden beispielsweise Methodenargumente immer gemäß der Tabelle **??** konvertiert, was bei primitiven Datentypen wie `int` und `float` unter Umständen zu Problemen führen kann, weil keine passende Methode gefunden wird. Beispielsweise führt der folgende Code zu einem Fehler, da der PHP-Wert (5) in einen java.lang.Long konvertiert wird, die Methode aber einen Integer erwartet.

```
$date = $obj->setIntval(5);
```

Listing 4.15: Typkonversionsfehler

Auch ist es so nur möglich öffentliche (`public`) Methoden aufzurufen, bei privaten Methoden wirft `Method.invoke()` eine Exception. In Fällen in denen der Anwender private oder ganz bestimmte Methoden aufrufen will, läßt es sich also leider nicht vermeiden, dass der Anwender auf `findMethod()` und `javaInvoke()` zurückgreifen muss. In den meisten Fällen bringt diese Änderung dem PHP-Programmierer aber eine erhebliche Vereinfachung, und rechtfertigt die entstehenden Nachteile, zumal in speziellen Fällen auf die oben beschriebenen Methoden zurückgegriffen werden kann.

4.3.14 Java-Attribute, die 2.

Nachdem nun der Aufruf von Java-Methoden deutlich vereinfacht wurde, sollte eine ähnliche Verbesserung beim Zugriff auf Java-Attribute entwickelt werden, sprich der direkte Zugriff ohne den Aufruf einer speziellen Methode und vor allem ohne die Notwendigkeit die Signatur des Attributes anzugeben. Weiter oben wurde beschrieben, dass PHP-Klassen Callbacks für das Setzen und Auslesen von Attributen besitzen. Diese sollten nun im TurpitudeJavaObject und in der TurpitudeJavaClass implementiert werden. Wie schon im vorherigen Kapitel sollte hierzu die Java-Reflection API zur Hilfe genommen werden. Die Callbacks `__get` und `__set` haben die selben Parameter die alle Methoden einer Zend-internen Klasse erwarten, und somit musste auch hier zunächst der Feldname und der Feld-Wert aus der Parameter-HashMap ausgelesen und zwischengespeichert werden. Der erste Parameter ist immer der Feldname, und im Falle von `__set` ist der zu setzende Wert an zweiter Stelle zu finden. Der Pointer auf den Rückgabewert macht offensichtlich nur bei `__get` Sinn, speichert aber in diesem Fall den ausgelesenen Wert. Beide Callbacks rufen statische Methoden des in 4.3.13 beschriebenen `ReflectHelper` auf, `getFieldValue()` und `setFieldValue()`, die beide die Methode `getField()` der Klasse des Java-Objektes benutzen um das entsprechende `java.lang.reflect.Field` des Attributes zu finden, um dann auf diesem die Methode `get()` beziehungsweise `set()` aufzurufen.

getField() geht nach folgendem Algorithmus vor: Wenn die Klasse ein Feld mit dem gesuchten Namen definiert wird dieses zurückgegeben, ist dies nicht der Fall und die Klasse implementiert mindestens ein Interface, werden diese Interfaces nach einem passenden Feld durchsucht. Wird auch hier kein Feld gefunden, wird der Algorithmus rekursiv auf die Vaterklasse angewandt, hat die Klasse keine Vaterklasse wird ein Fehler erzeugt. Die involvierten PHP-Variablen, beim schreibenden Zugriff der zu schreibende Wert, beim lesenden Zugriff der ausgelesene Wert, werden wie immer nach den Tabellen ?? und ?? konvertiert. Treten beim Zugriff Fehler auf, so wird die Ausführung des PHP-Codes abgebrochen und ein entsprechender Fehler als Java-Exception geworfen. Nun konnte auf die Felder eines Java-Objektes wie folgt zugegriffen werden:

```
$obj->stringval = 'wert';
$str = $obj->stringval;
```

Listing 4.16: Verbesserter Zugriff auf Java-Attribute

Im Zuge dieser Verbesserung wurde auch der direkte Zugriff auf statische Attribute von Java-Klassen ermöglicht, indem die Callbacks __get und __set in der TurpitudeJavaClass entsprechend implementiert wurden, sie unterscheiden sich von den Callbacks des TurpitudeJavaObject lediglich darin, dass sie beim ReflectHelper die entsprechenden Methoden getStaticFieldValue() und setStaticFieldValue() aufrufen. Diese Methode des Zugriffs auf Java-Attribute hat die selben Nachteile wie der direkte Zugriff auf Java-Methoden, sprich es können wieder nur öffentliche Felder gesetzt und ausgelesen werden, und die Typkonversion kann in manchen Fällen zu Problemen führen, aber auch hier überwiegen nach Meinung des Autors die Vorteile.

4.3.15 Erzeugen von Java-Arrays in PHP

In 4.3.12 wurde beschrieben, dass Arrays die von Java an PHP übergeben werden, als spezielles Objekt mit speziellen Methoden abgebildet werden. Was allerdings noch fehlte, war eine Möglichkeit Java-Arrays in PHP zu erzeugen, beispielsweise um sie als Argument an eine Methode zu übergeben. Das JNI bietet mit den Funktionen New<Type>Array die Möglichkeit beliebige Arrays zu erstellen, im Folgenden wird beschrieben wie diese Funktionen durch Turpitude in PHP abgebildet und dem Anwender zugänglich gemacht wurden.

Die Methode um neue Arrays zu erzeugen sollte newArray() heißen, und da sie sich keiner Klasse zuordnen ließ wurde sie dem TurpitudeEnvironment hinzugefügt. Sie erwartet zwei Argumente, zum einen den JNI-kodierten Typ der Arrayelemente als String, und zum anderen die Länge, die das zu erzeugende Array haben soll. Intern wird das Erzeugen von der Funktion turpitude_env_method_newarray() übernommen, die die selben Parameter entgegennimmt wie die meisten PHP-Methoden implementierende Funktionen. Innerhalb dieser Funktion werden zunächst die Anzahl und Typen der Argumente überprüft, und mit der Funktion get_java_field_type()

der Typstring geparst. Anhand des Typs wird entschieden welche JNI-Funktion aufgerufen werden soll, wird der Typ nicht erkannt wird ein Laufzeitfehler geworfen. Ein Sonderfall sind Objektarrays, zum einen weil hier nicht überprüft wird ob die Klasse valide ist, und zum anderen weil auch ein Objektarray erzeugt wird wenn es sich beim verlangten Typ um ein Array handelt, und somit ein Array von Arrays erstellt werden soll. Trat bis hierhin kein Fehler auf wird das Array noch in ein TurpitudeJavaArray verpackt, und an den Anwender zurückgegeben, der dann wie gewohnt mit dem Array verfahren kann.

4.3.16 Implementierung fehlender Java-Funktionalität

Zur Abrundung der Bibliothek fehlte dem PHP-Teil noch einiges an Funktionalität, leider konnten aus zeitlichen Gründen nicht alle Features des JNI dem PHP-Entwickler zugänglich gemacht werden, dennoch wurden dem TurpitudeEnvironment am Ende des Entwicklungszyklus noch einige Methoden hinzugefügt. Zunächst die Methode `instanceOf()`, die den Java-Operator `instanceof` nachbildet. Diese Methode erwartet zwei Argumente, zum einen das zu testende Objekt, und zum anderen die Klasse von der der Anwender wissen will ob das übergebene Objekt eine Instanz dieser ist.

Ausserdem wurde die TurpitudeJavaClass um die Methode `isCastable()` erweitert, die true zurückgibt wenn das übergebene TurpitudeJavaObject entweder eine Instanz der Klasse ist, oder wenn sich das Objekt problemlos auf die Java-Klasse casten läßt, die von der TurpitudeJavaClass gekapselt wird. Das Casten von Objekten selbst wurde nicht implementiert, da der Aufruf von Methoden und der Zugriff auf Attribute entweder unter Verwendung des genauen Klassennamens, oder aber über Reflection geschieht, es also unerheblich ist Instanz welcher Klasse das Objekt genau ist.

4.3.17 Setzen von PHP.ini Parametern

Als letztes Feature stand nun noch das Setzen von Parametern aus, die normalerweise über die Datei "php.ini" gesteuert werden. Schnell war klar dass die geeignete Zend-API Funktion dies zu erreichen `zend_alter_ini_entry()` war. Diese Funktion erwartet als Argumente den Namen des INI-Parameters, dessen Länge, und den zu setzenden Wert sowie dessen Länge. Außerdem erwartet sie zwei weitere Parameter, den "*modify_type*" und die "*stage*", deren Bedeutung sich zunächst nicht ohne weiteres erschloss. Naive Versuche mittels dieser Funktion Parameter zu setzen waren erfolglos, und da weder die Funktion noch ihre Parameter dokumentiert waren, konnte abermals nur durch ausführliches Studium des Zend-Quelltextes ein Erfolg erzielt werden.

Wie sich herausstellte hat jeder php.ini-Wert eine Bitmaske die anzeigt wann er verändert werden darf, und das oben angesprochene Argument *stage* wird mit

dieser Bitmaske verundet, um festzustellen ob eine Veränderung zu diesem Zeitpunkt zuläßig ist. Die ersten vier Bits der Bitmaske stehen in dieser Reihenfolge für folgende Zeitpunkte:

1. `STARTUP` - nach dem Initialisieren des Interpreters
2. `SHUTDOWN` - vor dem Herunterfahren des Interpreters
3. `ACTIVATE` - nach dem Starten des sog. Requests *
4. `DEACTIVATE` - vor der Beendigung des Requests, aber nach Ausführung des Skriptes
5. `RUNTIME` - zur Laufzeit des PHP-Skriptes

Der Parameter *modify_type* hingegen gibt an wer die Änderung vornehmen will, und auch hierzu enthält jeder php.ini-Wert wieder eine Bitmaske. Die Bedeutung der einzelnen Bits stellt sich wie folgt dar:

1. `USER` - Diese Werte kann der PHP-Benutzer, sprich das ausgeführte Skript setzen
2. `PERDIR` - Diese Werte können pro Verzeichnis † gesetzt werden.
3. `SYSTEM` - Diese Werte dürfen nur systemweit verändert werden

Nachdem die Bedeutung der beiden Bitmasken klar war, funktionierte das Setzen von php.ini-Werten immer noch nicht, was daran lag, dass die Längenangabe des Parameternamens mysteriöserweise um eins gößer sein muß als er wirklich ist. Diese Lösung wurde nur durch Zufall gefunden, und der Grund hierfür konnte nicht ermittelt werden.

Nun musste noch ein Weg gefunden werden der es dem Anwender erlaubt diese Werte auf möglichst einfache Weise zu setzen, und es wurde beschlossen dieses Ziel über Java-Systemproperties zu erreichen, da diese beim Start der JVM als Kommandozeilenparameter übergeben, oder aber vor dem Start der ScriptEngine programmatisch gesetzt werden können. Folglich wurde die C-Implementierung der `startUp()`-Methode der PHPScriptEngine derart angepasst, dass vor dem Start des Requests eine Java-Methode aufgerufen wird, die aus den Systemproperties diejenigen ausliest, die mit einem bestimmten Präfix beginnen, welcher als statisches Feld

*an dieser Stelle ist wieder einmal sehr deutlich zu merken wie sehr PHP mit den Konzepten der Web- und CGI-Welt verbunden ist, in der PHP oftmals als Modul in einem Webserver läuft, und jeder Seitenzugriff (Request) unabhängig behandelt werden muss.

†Auch dies ist ein Hinweis auf die CGI-Abstammung von PHP, wo oftmals unterschiedliche Benutzer Skript im gleichen Webserver ausführen, und unabhängig voneinander php.ini-Werte setzen wollen.

`PropertyPrefix` in der ScriptEngine gespeichert ist. Standardmäßig ist der Prefix `net.xp_framework.turpitude.ini`, und die Namen der zu setzenden php.ini-Werte werden mit einem Punkt separiert an diesen Präfix angehängt. Jedes der so gewonnenen Schlüssel/Wertepaare wird wieder an eine native Methode übergeben, die schließlich `zend_alter_ini_entry()` mit den nötigen Argumenten aufruft, um den Wert zu setzen. Diese Vorgehensweise mag etwas kompliziert erscheinen, aber dadurch, dass die php.ini-Werte vor dem Start des Requests gesetzt werden, können alle Werte verändert werden, gleich welchen Wert ihr `modify_type`-Attribut hat.

4.4 Fazit

Im Laufe dieses Kapitels wurde eine mächtige Bibliothek entwickelt, die nicht nur das Ausführen von PHP-Skripten aus einer Java-Laufzeitumgebung heraus ermöglicht, sondern viel weiter geht und die beiden Programmiersprachen eng miteinander verbindet. Dem PHP-Entwickler wird die Möglichkeit geboten den vollen Funktionsumfang von Java inklusive aller verfügbaren Java-Bibliotheken zu nutzen, Objekte zu erzeugen, auf deren Felder und Methoden zuzugreifen und diese auf intuitive und einfache Weise in PHP zu nutzen. Der Java-Entwickler kann PHP-Skripte nicht nur ausführen, ihnen Daten übergeben und von ihnen erzeugte Daten zurückerhalten, er kann nicht nur gezielt Funktionen und Methoden innerhalb eines PHP-Skriptes aufrufen, sondern er kann sogar Java-Interfaces direkt in PHP implementieren, oder bereits bestehende PHP-Klassen in ein Java-Interface wrappen und transparent mit ihnen wie mit jedem anderen Java-Objekt umgehen.

Dieser Funktionsumfang macht Turpitude einzigartig. Es existieren zwar zwei "Konkurrenzprodukte" auf dem Markt - die JSR223-Implementierung von Zend und die PHP/Java-Bridge [BRI06] - allerdings implementieren beide weder den vollen Umfang des JSR223, noch halten sie die aktuelle Spezifikation ein. Weiterhin ist die Zend-Implementierung nicht Queloffen, sprich es ist einem Anwender nicht möglich ein eigenes PHP in Java zu benutzen, mit allen Extensions die er benötigt, und es ist ebenfalls unmöglich sie auf nicht explizit unterstützten Plattformen einzusetzen. Die PHP/Java-Bridge ist zwar im Quelltext verfügbar, aber sie benutzt eine TCP-Verbindung um mit einer laufenden Java-Umgebung zu kommunizieren. Diese Kommunikation basiert außerdem noch auf XML, die Nachteile einer solchen Kommunikation wurden in Kapitel 2 hinlänglich beschrieben. Die Tatsache, dass der verwendete PHP-Interpreter aus einer externen Bibliothek geladen wird stellt sicher, dass der Anwender jegliches PHP-Programm ausführen kann. Insbesondere wird so der komplette Funktionsumfang des XP-Frameworks unterstützt.

Somit wurden alle am Anfang des Kapitels gesteckten Ziele nicht nur erreicht sondern in vielen Fällen sogar übererfüllt.

Die Implementierung der Bibliothek selbst wurde hauptsächlich durch zwei Faktoren erschwert: Zum einen ist die JSR 223-Spezifikation in vielen Detailfragen nur äußerst ungenau und erscheint außerdem in einigen Bereichen etwas undurchdacht. Dies lies sich allerdings in den meisten Fällen durch eine angemessene Portion Pragmatismus sehr schnell und zur Zufriedenheit Aller lösen. Deutlich mehr Zeit kostete die mangelhafte Dokumentation der Zend-Engine. Viele Probleme konnten nur durch langwieriges Ausprobieren und das Lesen des Zend-Quelltextes gelöst werden, was oft zu unnötigen Verzögerungen des Projektes führte. Abgesehen von den veralteten und nicht gepflegten Dokumentationsbruchstücken auf der PHP-Seite ([PHP06b]) war die einzig wirklich hilfreiche "Dokumentation" die, die durch das Programm *LXR* (siehe [Gje05]) erzeugt wird und unter [PHP07] verfügbar ist. Der Quelltext von PHP selbst stellte weitere Herausforderungen, so ist er nur sehr unzureichend kommentiert, und

verwendet einige eher “interessante“ Konzepte. Das verdeutlichen am besten einige Beispiele, zum einen die Implementierung einer verketteten Liste:

```
typedef struct _zend_llist_element {
    struct _zend_llist_element *next;
    struct _zend_llist_element *prev;
    char data[1]; /* Needs to always be last in the struct */
} zend_llist_element;
...
zend_llist_element *le;
...
opline_ptr = (zend_op *)le->data;
```

Listing 4.17: Verkettete Liste im Zend-Code

Hier wird genau ein Byte Speicher reserviert, es werden allerdings Pointer an diese Stelle geschrieben, die mindestens vier Bytes lang sind - ein Voidpointer wäre hier passender gewesen. Zum anderen noch ein Ausschnitt aus der De-serialisierungsroutine. Hier wurden zwar Kommentare verwendet, allerdings zeugen sie an dieser Stelle nicht unbedingt von einer guten Organisation:

```
yy13:    ++YYCURSOR;
    goto yy14;
yy14:
{
    /* this is the case where we have less data than planned */
    php_error_docref(
        NULL TSRMLS_CC,
        E_NOTICE,
        "Unexpected end of serialized data");
    return 0; /* not sure if it should be 0 or 1 here? */
}
```

Listing 4.18: Zend-Engine: De-serialisierung

Trotz dieser Probleme konnte ein benutzbares Softwaresystem erstellt werden, auch wenn Turpitude sicherlich noch keinen Produktionsstatus erreicht hat, hierzu muss der Quelltext noch hinreichend auf Speicherlecks überprüft werden, da der Einsatz in einem Web- oder Application-Server immer lange Laufzeiten mit sich bringt, und Speicherlecks in diesen Fällen besonders gravierend sind. Weiterhin kann an vielen Stellen unnötig gewordener Quelltext entfernt oder ähnliche Quelltextstellen zu eigenen Funktionen refaktoriert werden. Die Begrenzte Zeit die für diese Arbeit zur Verfügung stand ließ solche “Schönheitsoperationen“ aber leider nicht zu.

Nach Einschätzung des Autors kann Turpitude zu diesem Zeitpunkt aber schon guten Gewissens eingesetzt werden um nicht unternehmenskritische Anwendungen zu entwickeln, vor allem weil eventuelle qualitative Verbesserungen sich nicht mehr auf die Schnittstellen auswirken werden. Es gilt allerdings zu beachten, dass die Bibliothek unter keinen Umständen genutzt werden sollte um nicht vertrauenswürdige

PHP-Skripte auszuführen, zum einen weil nicht alle Grenzfälle ausgetestet wurden, und zum anderen weil durch den Zugriff auf Java-Klassen die PHP-eigenen Schutzmechanismen ausgehebelt werden könnten. Ein solcher Einsatz beispielsweise um Kunden auf einem Java-Webserver PHP anzubieten wäre - zumindest zu diesem Zeitpunkt - grob fahrlässig.

Im Laufe der Implementierung wurde deutlich, dass eine - im Gegensatz zur ursprünglichen Planung - vollständige Implementierung des JSR223 und vor allem die Implementierung des Zugriffs auf Java-Funktionalität aus dem PHP-Interpreter heraus dem Unternehmen und der Abteilung viele Vorteile bringen würde, weswegen deutlich mehr Zeit in die Entwicklung der Bibliothek investiert wurde als eigentlich geplant. Dies führte nun dazu dass für den Rest der Aufgabe deutlich weniger Zeit zur Verfügung stand, weswegen in diesen Bereichen an Umfang gekürzt werden musste.

Kapitel 5

JBoss und PHP

Die ursprüngliche Planung sah vor, diesem Teil der Aufgabe, der Integration von Turpitude in einen J2EE Application Server und der Umsetzung der verschiedenen Enterprise Java Bean-Typen in PHP, einen erheblichen Teil der zur Verfügung stehenden Zeit zu widmen. Wie allerdings in Kapitel 4.4 beschrieben, verschoben sich diese Prioritäten im Laufe der Implementierung des ersten Teils der Aufgabenstellung, wodurch erheblich weniger Zeit für dieses Kapitel zur Verfügung stand. Trotz dieser Kürzung soll an dieser Stelle eine mögliche Anwendung für Turpitude erarbeitet werden, um einen kleinen Teil der Möglichkeiten, die diese Bibliothek bietet, aufzuzeigen. Weiterhin soll durch das Entwickeln einer solchen Beispielanwendung die Einsatzfähigkeit Turpitudes unter möglichst realen Bedingungen nachgewiesen werden. Außerdem sollen eventuelle Fehler in der Bibliothek auf diese Weise gefunden und behoben werden.

5.1 Aufgabe

Ziel ist es, eine Möglichkeit zu schaffen die verschiedenen Enterprise Java Bean Typen mit PHP zu implementieren. Hierzu muss zuerst ein Weg gefunden werden der JVM, die den Application Server ausführt die nötigen Parameter zu übergeben die das Laden von Turpitude zur Laufzeit ermöglichen. Wünschenswert hierbei wäre, dass die nötigen Bibliotheken nur einmal beim Start des Application Servers geladen werden, anstatt jeder installierten Anwendung mitgegeben werden zu müssen. Weiterhin muss gewährleistet werden dass Fehler in der geladenen nativen Bibliothek und in PHP auftretende Fehler möglichst nicht zum Absturz des Application Servers führen, um eine Beeinträchtigung anderer, im selben Container laufender Applikationen ausschließen zu können. Ein weiteres Ziel der ursprünglichen Aufgabenstellung war die Entwicklung eines Buildsystems das dem Entwickler das unnötige Schreiben von sogenanntem “Boilerplate-Code“, Quelltext der für jede Applikation immer gleich ist,

abnimmt. Der PHP-Entwickler, der EJBs schreiben möchte, sollte nach Möglichkeit keinerlei Java-Quelltext schreiben müssen. Leider kann ein solches Buildsystem aufgrund des kleiner gewordenen Zeitrahmens nicht implementiert werden, allerdings soll bei der Entwicklung darauf geachtet werden, dass die spätere Entwicklung eines solchen Systems möglich bleibt. Abbildung 5.1 zeigt das Schema eines Aufrufs einer in PHP implementierten Enterprise Java Bean.

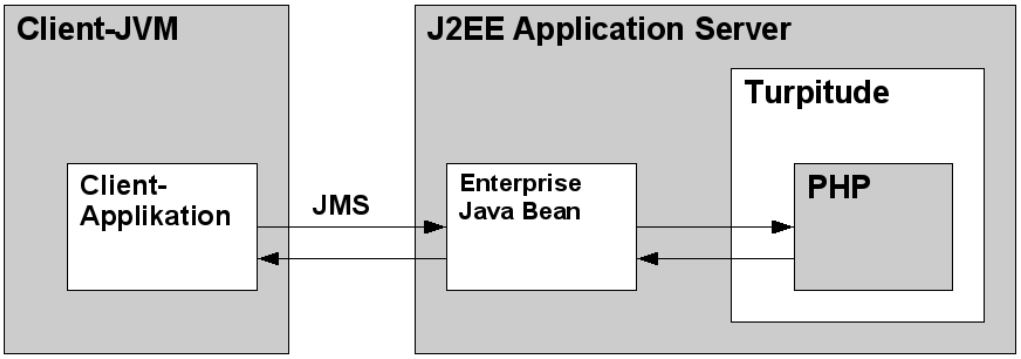

Abbildung 5.1: Aufruf einer PHP-EJB

5.2 EJB 3.0

Obwohl innerhalb der Firma - im Gegensatz zu EJB 2.0 - nur sehr wenig Erfahrung mit der *Enterprise Java Bean Specification* in der neuesten Version 3.0 (siehe [EJB07]) vorhanden war, wurde beschlossen das Projekt unter Einsatz dieser durchzuführen. Diese Entscheidung fiel nicht nur um Erfahrungen mit dieser neuen Technologie zu sammeln, sondern auch um das Projekt zukunftssicherer zu machen. Aus diesem Grund sollen an dieser Stelle kurz die Unterschiede zwischen EJB 2.0 und 3.0 umrissen werden, es wird dabei davon ausgegangen, dass der Leser mit den Programmierkonzepten von EJB 2.0 zumindest im Groben vertraut ist.

Die früheren J2EE 1.4 and EJB 2.1 Spezifikationen sind sehr komplex, Entwickler mussten sich mit dem EJB-Komponentenmodell, verschiedenen APIs und Entwurfsmustern sowie XML Metainformationsdateien vertraut machen bevor sie anfangen konnten benutzbare Softwaresysteme zu entwickeln. Diese Komplexität verhinderte nicht nur die schnelle Adaption von J2EE, sondern führte auch dazu, dass J2EE oft falsch eingesetzt wurde, was wiederum schlechtere anstatt bessere Software zur Folge hatte. Man erkannte, dass das ursprünglich vorrangige Ziel von EJB - die Gewährleistung transaktioneller Integrität über verteilte Applikationen hinweg - von “Enterprise Applications“ gar nicht benötigt wurde. Die EJB 3.0 Spezifikation versucht nun sich diesem Komplexitätsproblem anzunehmen, und aus den Erfahrungen erfolgreich über

längere Zeiträume eingesetzter Opensource-Projekte wie beispielsweise *Hibernate*[*] oder *XDoclet*[†] zu lernen. Sie basiert auf Java-Annotationen und POJOs [‡] , und ist sehr viel leichter zu verstehen als frühere Versionen, ohne jedoch an Mächtigkeit einzubüßen.

Mit Java-Annotationen (siehe [JAV04]) wurde Java erstmals in der Version 5 mit der Möglichkeit ausgestattet, den Quelltext mit Metadaten zu versehen. Durch die Verwendung von Annotationen wird in EJB 3.0 zum einen die Anzahl von benötigten Klassen und Interfaces reduziert, und zum anderen werden so Deploymentdeskriptoren weitestgehend unnötig. Dies wird durch eine sinnvolle Vorbelegung von Konfigurationsparametern möglich, davon abweichende Werte werden durch Annotationen direkt im Quelltext der Java-Klassen untergebracht. Weiterhin werden Annotationen verwendet um Abhängigkeiten zur Umgebung sowie JNDI-Zugriffe zu spezifizieren und durch Dependecy Injection [§] automatisch aufzulösen. Ausserdem wurde die Notwendigkeit abgeschafft von EJB-spezifischen Interfaces zu erben oder diese zu implementieren, Session Beans benötigen kein Home-Interface mehr, Entity Beans sind nun einfach Java-Klassen (POJOs). Persistenzeigenschaften werden realisiert indem die neue *Java Persistence Architecture*, welche bessere Möglichkeiten zur Abfrage, für Mengenoperationen und Vererbung bietet, verwendet wird. Die Implementierung von in früheren Standards vorgeschriebenen Lebenszyklusmethoden ist jetzt optional. Solche Methoden wurden früher benötigt, um zu bestimmten Zeitpunkten des Bestehens einer EJB Aktionen durchführen zu können. Ein weiteres Merkmal von EJB 3.0 sind die sogenannten "Interceptors". Hierbei handelt es sich um Methoden oder Klassen die mittels Annotationen an Session- und Message Driven Beans sowie an deren Methoden angehängt werden können. Sobald das annotierte Element (englisch "target") aufgerufen wird ruft der Application Server automatisch den angehängten Interceptor auf.

5.3 Infrastruktur

Bevor mit der Bearbeitung der Aufgabe begonnen werden konnte musste zunächst die nötige Infrastruktur geschaffen werden, allem voran die Auswahl des zu verwendenden Application Servers. Da bei 1&1 ausschließlich der Application Server JBoss [JBO06] eingesetzt wird, war der Einsatz dieses Application Servers eine Vorgabe.

[*] Hibernate ist ein O/R Mapper, eine Middleware die das Abbilden von relationen Datenbanken auf objektorierte Datenstrukturen erlaubt, siehe [HIB06]

[†] XDoclet ermöglicht das attributorientierte Arbeiten in Java vor Version 5, indem es die annotierten Attribute vor dem eigentlichen Übersetzen mittels eines Präprozessors in Java-Quelltext umwandelt, siehe [XDO05]

[‡] POJO steht für "Plain Old Java Object" und bedeutet "ganz normales Java-Objekt". Insbesondere unterscheiden sich diese von mit vielfältigen externen Abhängigkeiten belasteten Objekttypen.

[§] Entwurfsmuster das dazu dient in einem objektorientierten System Abhängigkeiten zwischen Komponenten oder Objekten zu minimieren, siehe [DEP07]

Folglich wurde ein JBoss in der Version 4.0.5 installiert, wobei darauf zu achten war dass der nicht standardmäßig mitinstallierte Deployer* für EJB 3.0 Applikationen zusätzlich ausgewählt wurde. Direkt nach der Installation konnte der Application Server problemlos gestartet werden.

Nun wurde für das Projekt eine Verzeichnisstruktur aufgebaut wie Sun sie für EJB-Projekte vorschlägt, mit eigenen Unterverzeichnissen für die Quelltexte (**src/**), benötigte Bibliotheken (**lib/**), zusätzliche Deploymentdeskriptoren (**dd/**) und die beim Buildprozess erzeugten Dateien (**build/**). Da es sich um ein reines Java-Projekt handelte wurde als Buildsystem Apache Ant [ANT06] und nicht wie für Turpitude selbst *make* gewählt. Ant benutzt die XML-Datei "build.xml" wie *make* das "Makefile" benutzt. Der Wurzelknoten muss `project` heißen. Darunter werden die einzelnen Targets angelegt, die wiederum andere Targets als Abhängigkeiten haben können. Als Name für das Projekt wurde "phpejb" gewählt, die benötigten Klassen liegen im Package `net.xp_framework.phpejb`. Bevor mit der Programmierung begonnen werden konnte mussten jedoch Targets für das Übersetzen, Packen und Deployen der Anwendung angelegt, sowie am Anfang der build.xml einige Konfigurationsparameter gesetzt werden. Das Übersetzen geschieht im Target `compile` und bereitete keine Schwierigkeiten. Die so erzeugten .class-Dateien werden im Target `package-ejb` in ein JAR gepackt, das den Namen *phpejb.ejb3* trägt, damit der JBoss erkennen kann welchen Deploymechanismus er verwenden muss. Dieses JAR wird zusammen mit dem Deploymentdeskriptor **application.xml** - in dem lediglich beschrieben wird welche Module enthalten sind - in das *Enterprise Archive*† **phpejb.ear** zusammengepackt, dies geschieht im Target `assemble-app`. Ein weiteres Target namens `deploy` kopiert dieses in das Verzeichnis in dem der JBoss die auszubringenden Applikationen erwartet. Schlussendlich wurde noch das Target `clean`, das die während des Buildprozess erzeugten Dateien löscht, sowie das Target `all` angelegt, welches als erstes Target in der build.xml steht und deswegen ausgeführt wird wenn Ant ohne Parameter aufgerufen wird, und die nötigen Schritte enthält die zur Erzeugung der *phpejb.ear* nötig sind. Somit waren die Grundlagen für die weitere Entwicklung gelegt, und es konnte ein zwar leeres, aber dennoch valides .ear erzeugt werden.

5.3.1 Beispielanwendung

Um erste Erfahrungen mit einer EJB 3.0 Anwendung zu sammeln sollte nun ein einfacher Service entwickelt werden. Damit diese Arbeit auch später von Nutzen ist wurde beschlossen einen Service zu schreiben der das Verhalten der in 4.3 vorgestellten An-

*Ein Deployer ist eine Softwarekomponente, die das eigentliche Ausbringen eines bestimmten Archivtyps übernimmt

†auch: EAR-Datei. Java EE packt Applikatinen in eine EAR-Datei, um sie leichter ausbringen zu können. EAR-Dateien enthalten .JAR- und .WAR-Dateien.

wendung `EngineList` nachempfindet, und eine Liste der zur Verfügung stehenden JSR 223 Implementierungen zurückgibt.

Zu jedem EJB Service gehört ein beschreibendes Interface, das Interface für diesen Service ist denkbar einfach, es besteht aus nur einer Methode, `getList()`. Durch die Annotation `@Remote` wird angegeben, dass dieses Interface auch von remote* aufgerufen werden kann.

```
@Remote
public interface EngineList {
    public List<String> getList();
}
```

Listing 5.1: Testservice Interface

Zu diesem Interface musste nun eine implementierende Klasse geschrieben werden. Dadurch, dass der Service auch Ziel eines entfernten Objektaufrufs sein kann, muss keine lokale Implementierung vorhanden sein, ein Remote-Bean kann sowohl lokal als auch remote aufgerufen werden. Die simpelste EJB ist die Stateless Session Bean; die Annotation `@Stateless` gibt an dass es sich bei dieser Bean um eine solche handelt.

```
@Stateless
public class EngineListBean implements EngineList {
    public List<String> getList() {
    ...
    }
}
```

Listing 5.2: Testservice Bean

Eine einfache Methode zum Testen einers EJB-Service ist es, einen Client zu schreiben, der in einer anderen JVM ausgeführt wird als der Application Server. Ein solcher Test zeigt nicht nur ob der Service funktioniert, sondern auch ob das Deployment korrekt abgelaufen ist, und ob der Service von aussen über JNDI aufgefunden und angesprochen werden kann. Ein solcher JNDI-Lookup geschieht immer über einen Kontext, welcher die nötigen Service-Provider kennt. Der Lookup resultiert in einem Stub-Objekt, das Aufrufe über die Netzverbindung an den Application Server weiterleitet.

```
InitialContext ctx = new InitialContext(env);
EngineList list =
    (EngineList)ctx.lookup("phpejb/EngineListBean/remote");
List<String> lst = list.getList();
```

Listing 5.3: Testservice Client

*ugs. für "entfernter Objektaufruf", im Gegensatz zu zum lokalen Aufruf

Der Aufruf ergab als einzige innerhalb des JBoss verfügbare JSR 223 Implementierung die standardmäßig in Java 6 enthaltene Javascript-Engine Rhino von Mozilla. Nun mussten der den JBoss ausführenden JVM die nötigen Parameter mitgegeben werden, damit Turpitude innerhalb des Application Servers verfügbar wird (siehe auch B.2). Hierzu wurde ein Shell-Skript geschrieben, das die benötigten Umgebungsvariablen (LD_LIBRARY_PATH, JAVA_HOME) setzt, und dann das eigentliche JBoss-Startskript *run.sh* mit den nötigen Parametern ausführt. Danach erschien Turpitude in der Liste der verfügbaren ScriptEngines, und es konnte mit der eigenlichen Aufgabe begonnen werden.

```
#!/bin/sh
TURP_HOME=<PFAD>
PHP_HOME=<PFAD>
export JBOSS_HOME=/home/nsn/jboss-4.0.5.GA
export JAVA_HOME=/home/nsn/jdk1.6.0
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PHP_HOME/libs:$TURP_HOME
./run.sh --classpath=$TURP_HOME/turpitude.jar
```

Listing 5.4: JBoss Startskript

5.4 Stateless Session Beans

Die einfachste Enterprise Java Bean ist die Stateless Session Bean, weshalb sie als erstes umgesetzt werden sollte. Stateless Session Beans haben - wie der Name bereits vermuten läss t - keinen eigenen Zustand, sie können also keine Informationen über mehrere Anfragen hinweg vorhalten.

5.4.1 Interface

Wie für jede EJB muss auch für die Stateless Session Bean zunächst ein Interface definiert werden. Das Interface für diese Beispielanwendung ist sehr einfach, es besteht lediglich aus einer einzigen Methode, welche einen Namen als String entgegennimmt und einen String zurückgibt, der eine Grußbotschaft enthält.

```
public interface SLHelloWorld {
    public String sayHello(String s);
}
```

Listing 5.5: Stateless Hello World Interface

5.4.2 PHP-Implementierung

Die Implementierung des Bean-Interfaces besteht aus zwei Teilen: zum einen dem Java-Teil, der die `ScriptEngine` instanziiert, den PHP-Quelltext lädt und übersetzt sowie die entsprechenden PHP-Funktionen aufruft, und zum anderen den PHP-Teil, der die eigentliche Interface-Implementierung enthält.

```
public String sayHello(String s) {
...
    ScriptEngineManager mgr = new ScriptEngineManager();
    ScriptEngine eng = mgr.getEngineByName("turpitude");
    Compilable comp = (Compilable)eng;
    CompiledScript script = comp.compile(/* Source */);
    Invocable inv = (Invocable)script;
    SLHelloWorld hw =
        inv.getInterface(SLHelloWorld.class);
...
    return hw.sayHello(s);
}
```

Listing 5.6: Java-Teil

Obwohl Turpitude viele Möglichkeiten bietet, die nötige Funktionalität zu implementieren, wurde eine sehr komplizierte Variante gewählt: das Implementieren des EJB-Interfaces in PHP, sowie das Aufrufen der PHP-Methoden über die Methoden des JSR 223-Interfaces `javax.script.Invocable`. Diese Entscheidung wurde vor allem vor dem Hintergrund getroffen, dass die Entwicklung einer PHP-Implementierung

eines Java-Interfaces intuitiver erscheint, als das Entwickeln von PHP-Skripten die sich an eine nicht trivial ersichtliche Konvention halten müssen um vom Java-Teil aufgerufen werden zu können. Außerdem ist so ein generischer Adapter denkbar, der beliebige Java-Interfaces auf passende PHP-Implementierungen adaptiert, oder zumindest ein Automatismus, der aus einem gegebenen Java-Interface und einer PHP-Implementierung den nötigen Java-Quelltext automatisch erzeugt.

```
<?php
class SLHelloWorld {
  function sayHello($s) {
    return 'The PHP-implementation says hello to '.$s;
  }
}
?>
```

Listing 5.7: PHP-Implementierung

Der Aufruf des SLHelloWorld-Service unterscheidet sich nicht von dem Aufruf des EngineList-Services, siehe 5.3.1.

5.5 Stateful Session Beans

Eine Stateful Session Bean ist ein EJB die ihren internen Status über mehrere Aufrufe hinweg beibehalten kann. Methodenaufrufe des Clients an einen bestimmten Stub werden immer an die selbe Bean-Instanz weitergeleitet, somit behalten alle Felder der Bean ihre Werte solange der Client den Stub hält. Zwar geschieht das Erstel-

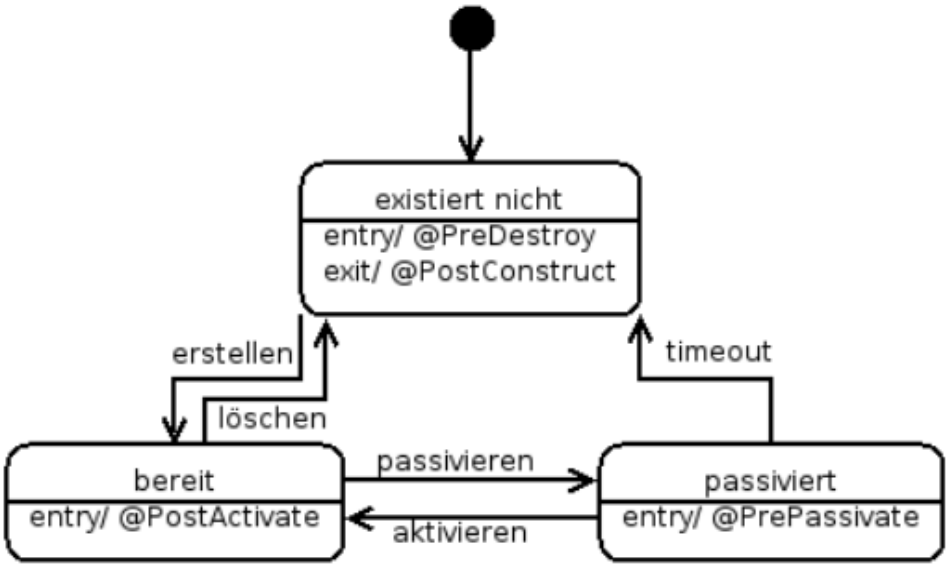

Abbildung 5.2: Lebenszyklus einer Stateful Session Bean, mit Annotationen

len, Vorhalten und Löschen der einzelnen Bean-Instanzen vollautomatisch durch den EJB-Container, trotzdem besteht manchmal die Notwendigkeit den Lebenszyklus einer Bean zu beeinflussen. Hierfür mussten in früheren EJB-Versionen sogenannte "Lifecycle-Callbacks" implementiert werden. In EJB 3.0 sind diese Callbacks überflüssig, und werden durch spezielle Annotationen ersetzt, die zu bestimmten Zeitpunkten im Lebenszyklus aufzurufende Methoden vermerken. Abbildung 5.2 stellt den Lebenszyklus einer Stateful Session Bean zusammen mit den jeweiligen Annotationen dar. Die meisten dieser Annotationen sind sowohl für Stateful Session Beans, als auch für Stateless Session Beans zulässig, im folgenden werden die wichtigsten aufgeführt:

@PostConstruct Die annotierte Methode wird direkt nachdem die Instanz erstellt wurde vom Container aufgerufen, @PostConstruct ist sowohl für Stateful Session Beans als auch für Stateless Session Beans zulässig.

@PreDestroy Die annotierte Methode wird aufgerufen bevor der Container eine unbenutzte oder abgelaufene Instanz aus seinem Objektpool löscht, @PreDestroy ist sowohl für Stateful als auch für Stateless Session Beans zulässig.

@PrePassivate Wenn eine Stateful Session Beans Instanz zu lange nicht aufgerufen werden kann der Container diese passivieren, und ihren Zustand in seinem Cache zwischenspeichern. Eine mit @PrePassivate annotierte Methode wird direkt vor dem Passivieren aufgerufen.

@PostActivate Wenn eine Anfrage an eine passivierte Bean gestellt wird wird eine neue Instanz der Bean-Klasse erzeugt und der gespeicherte Zustand wieder hergestellt. Nachdem dies geschehen ist, aber noch bevor der Aufruf schliesslich an die Bean-Instanz weitergeleitet wird, wird die mit @PostActivate annotierte Methode aufgerufen. Diese Annotation ist nur für Stateful Session Beans zulässig.

@Init Diese Annotation bezeichnet Initialisierungsmethoden für eine Stateful Session Bean. Der Unterschied zu @PostConstruct besteht darin, dass innerhalb einer Bean-Klasse mehrere Methoden mit @Init annotiert werden können, allerdings wird immer nur eine dieser Methoden tatsächlich vom Container aufgerufen. Welche das ist bestimmt sich aus der Art und Weise, wie die Bean erstellt wurde, näheres ist der EJB 3.0 Spezifikation [EJB07] zu entnehmen. Die @Init-Methode wird noch vor der @PostConstruct-Methode aufgerufen.

5.5.1 Interface

Um die zustandsvorhaltenden Eigenschaften der Stateful Session Bean zu testen enthält das Service-Interface eine Methode um den Namen der zu grüßenden Person zu setzen. Die Grußmethode selbst nimmt folglich keine Parameter entgegen, sondern nutzt den zuvor gesetzten Namen um die Grußbotschaft zu erstellen.

```
public interface SFHelloWorld {
    public void setName(String s) throws ScriptException;
    public String sayHello() throws ScriptException;
}
```

Listing 5.8: Stateful Hello World Interface

5.5.2 PHP-Implementierung

Im Gegensatz zum Service-Interface unterscheidet sich die Implementierung der Stateful Session Bean erheblich von der Stateless Session Bean. Da der Zustand der Bean nicht im Java-Teil, sondern vom PHP-Code vorgehalten werden sollte mussten Lebenszyklus-Methoden implementiert werden.

Zunächst wurde die mit `@PostConstruct` annotierte Methode `initialize()` geschrieben, in der der Quelltext übersetzt und eine Instanz der implementierenden PHP-Klasse erzeugt wird. Dieses Erzeugen geschieht mittels einer Art "friend-Methode", aber es sind auch beliebige andere Methoden denkbar. Sowohl das übersetzte Skript als auch das `PHPObject` der PHP-Klasse werden als Attribute der Bean

gespeichert. Allerdings konnten diese Attribute aus zwei Gründen nicht zur Speicherung des Zustandes genutzt werden: zum einen sollte im Hinblick auf eine mögliche Automatisierung sämtliche Funktionalität in PHP realisiert werden, und zum anderen kann innerhalb des Bean-Containers nicht davon ausgegangen werden, dass die in ByteBuffern gespeicherten Informationen - der Zend-Opcode des übersetzten Skriptes und vor allem der C-Pointer auf den `zval` des PHP-Objektes - beim nächsten Aufruf einer Bean-Methode noch valide sind. Folglich mussten der Bean zwei weitere Methoden hinzugefügt werden, die mit `@PrePassivate` annotierte Methode `sleep()`, und die mit `@PostActivate` annotierte Methode `wakeUp()`. Als Methode zur Persistierung des PHP-Objektes wurde der PHP-Mechanismus zur Serialisierung von Daten gewählt. Hierzu wurden der PHP-Klasse die Methoden `ser()` und `unser()` hinzugefügt, erstere gibt einen String zurück der eine serialisierte Fassung des Objektes enthält, letztere nimmt einen solchen String entgegen und stellt daraus den ursprünglichen Zustand wieder her. Damit gewährleistet ist dass die oben angesprochenen Attribute der Java-Klasse nicht versehentlich persistiert werden, werden die Referenzen der beiden Objekte auf `null` gesetzt. Die Aufrufe der eigentlichen Service-Methoden werden genau wie bei der Stateless Session Bean mittels des Invocable-Interfaces durchgeführt, ihre Implementierung in PHP gestaltete sich als trivial.

```
<?php
class SFHelloWorld {
  var $name = 'initial value';
  function setName($n) {
    $this->name = $n;
  }
  function sayHello() {
    return 'Stateful PHP says hello to '.$this->name;
  }
  function ser($obj) {
    return serialize($obj);
  }
  function unser($str) {
    $cpy = unserialize($str);
    $this->setName($cpy->name);
  }
}
```

Listing 5.9: PHP-Implementierung

5.6 Message Driven Beans

Eine Message Driven Bean implementiert kein Service-Interface im eigentlichen Sinn, ihre Methoden können somit nicht direkt aufgerufen werden. Vielmehr erwartet sie Nachrichten in einer JMS Message Queue und bearbeitet diese. JMS - der Java Messaging Service - ist ein Teil des Application Servers. Er erlaubt das Senden von Nachrichten (Messages) in Queues, an welche sich Dienste anhängen können um auf diese Nachrichten zu reagieren, auf diese Art und Weise wird die Realisierung asynchroner Abläufe möglich. In früheren EJB Versionen mussten diese Queues umständlich über XML-Dateien konfiguriert werden, mit EJB 3.0 werden sie dynamisch erzeugt sobald ein Zuhörer sich an eine Queue hängt, oder ein Sender eine Nachricht in eine Queue schickt. Das Entwickeln einer Message Driven Bean gestaltet sich sehr einfach, es muss lediglich das Interface `MessageListener` implementiert werden, welches als einzige Methode `onMessage()` definiert. Diese Methode wird vom Application Server aufgerufen wenn in der Queue, in der die Message Driven Bean lauscht, eine neue Nachricht vorliegt. Als einziger Parameter wird die anliegende Nachricht an die Bean übergeben.

5.6.1 Implementierung

Die Implementierung der Message Driven Bean bereitete keine großen Probleme, es musste lediglich darauf geachtet werden, dass alle Laufzeitfehler die in der `onMessage`-Methode auftreten abgefangen und behandelt werden, da sie aufgrund der Asynchronität des Aufrufs nicht an den Client weitergereicht werden können. Die Bean muss mit `@MessageDriven` annotiert werden. Diese Annotation kann weiterhin konfiguriert werden, zum einen mit dem Property `destinationType`, das den Typ der Queue angibt, auf der die Bean hören soll, zum anderen mit dem Property `destination`, das den Namen der Queue enthält.

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(
        propertyName="destination",
        propertyValue="queue/mdb")
    })
public class MDHelloWorld implements MessageListener {
    public void onMessage(Message msg) {
...
```

Listing 5.10: Konfiguration der Message Driven Bean

Der PHP-Teil der Implementierung unterscheidet sich kaum von dem der Stateless Session Bean, wieder wird die PHP-Klasse mittels einer Helper-Funktion instantiiert.

Weil es aber diesmal kein Service-Interface gibt wird die entsprechende Methode der PHP-Klasse von Java aus mit der `invokeMethod()`-Methode des `Invocable` Interfaces aufgerufen. Allerdings ist der Aufruf einer Message Driven Bean für den Client etwas kompliziertes als bei den vorangegangenen Beanarten, es muss eine JMS-Nachricht erstellt und an die richtige Queue verschickt werden. Hierzu muss zunächst die Queue im Kontext nachgeschlagen werden und eine QueueConnection erstellt werden, mit der dann eine QueueSession erzeugt werden kann. Mit Hilfe dieser Session wird ein Sender erstellt, der dann schließlich die Nachtticht - in diesem Fall eine einfache TextMessage - an die Queue schicken kann. Sobald die Session nicht mehr gebraucht wird sollte sie geschlossen werden, um sicherzustellen dass noch gecachte Nachrichten auch wirklich verschickt werden.

```
Queue queue = (Queue)ctx.lookup("queue/mdb");
QueueConnectionFactory factory =
    (QueueConnectionFactory)ctx.lookup("ConnectionFactory");
QueueConnection cnn = factory.createQueueConnection();
QueueSession sess =
    cnn.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
TextMessage msg = sess.createTextMessage("TestClient");
QueueSender sender = sess.createSender(queue);
sender.send(msg);
sess.close();
```

Listing 5.11: Senden einer JMS-Nachricht

5.7 Änderungen an Turpitude

Im Laufe der Entwicklung dieser Beispielanwendungen zeigten sich einige Schwachstellen in Turpitude, die aber behoben werden konnten. Die meisten aufgetretenen Probleme rührten daher, dass innerhalb eines J2EE Application Servers besondere Bedingungen vorherrschen. Dieses Kapitel erläutert diese Probleme und ihre Behebung.

5.7.1 finalize

Eine Besonderheit für eine Klasse wenn sie innerhalb eines Application Servers aufgerufen wird ist, dass keine Garantie über die Lebenszeit und -zyklen der Instanzen gegeben wird. Ein Objekt kann - immer entsprechend der J2EE-Spezifikation - mehrfach oder auch nur einfach benutzt werden, und es besteht keine Kontrolle über die Laufzeit der ausführenden JVM. Deswegen reichte der in der PHPScriptEngine eingebaut *VMShutdown-Hook* nicht mehr aus um den PHP-Interpreter aus dem Speicher zu entfernen und auf C-Ebene belegte Resourcen wieder freizugeben. Java kennt zwar keine Destruktoren, aber jede Klasse erbt von `Object` die Methode `finalize`, die jedesmal aufgerufen wird wenn das Objekt von der *Garbage Collection* aufgeräumt wird. Beim Überschreiben dieser Methode ist es angebracht einen `try-catch-finally`-Block um die ausgeführten Zeilen zu legen, und im finally-Teil `super.finalize()` aufzurufen. Jede in `finalize` auftretende Exception unterbricht zwar das Aufräumen, wird ansonsten aber ignoriert. Folglich wurde in der `PHPScriptEngine` die Methode `finalize` überschrieben, und innerhalb dieser wird `shutDown` aufgerufen.

5.7.2 Mehrfachinstanziierung

Wird innerhalb eines Java-Threads der PHP-Interpreter mehrfach instanziiert kann es zu Problemen kommen: Zum einen kann nicht mehr gewährleistet werden, dass die Ausführung des übersetzten PHP-Quelltextes fehlerfrei abläuft, da unter Umständen während der Ausführung die sogenannten *Executor Globals* (Skriptvariablen, Aufrufstacks und die Zend-Opcodes selbst) verändert werden, und zum anderen kann der mehrfache Aufruf der `startUp()`-Methode, in der der PHP-Interpreter initialisiert wird, zu schweren Fehlern bis hin zu Abstürzen der JVM führen. Um dies zu verhindern wurde die PHPScriptEngineFactory derart angepasst, dass sie eine abgeänderte Version des Singleton-Patterns implementiert um zu verhindern, dass mehrere `PHPScriptEngines` gleichzeitig existieren. Da der Konstruktor der PHPScriptEngine als `protected` deklariert wurde, kann angenommen werden dass die Methode `getScriptEngine()` der PHPScriptEngineFactory die einzige Stelle ist, an der ScriptEngine-Instanzen erzeugt werden. So wurde der Factory ein privates und statisches Attribut `MyEngine` hinzugefügt, welches die Singleton-Instanz der PHPScriptEngine vorhält. Dieses Attribut ist zu Begin `null`, und wird beim ersten Aufruf von `getScriptEngine()` gesetzt, jeder weitere Aufruf dieser Methode gibt nur noch

eine Referenz auf diese einzige ScriptEngine zurück. Das Erzeugen des Singletons wird zusätzlich durch einen `synchronized`-Block vor gleichzeitiger Mehrfachausführung geschützt.

```
public ScriptEngine getScriptEngine() {
    if (MyEngine == null) {
        synchronized (PHPScriptEngineFactory.class) {
            if (MyEngine == null)
                MyEngine = new PHPScriptEngine(this);
        }
    }
    return MyEngine;
}
```

Listing 5.12: Singleton-Erzeugung

5.7.3 Parameterlose Methodenaufrufe

Ein besonders hartnäckiger und schwer zu findender Fehler führte zu Abstürzen der JVM beim Aufrufen parameterloser Methoden über Java-Interfaces, die mittels `Invocable.getInterface` erzeugt wurden. Obwohl die InvocationHandler-Methode `invoke` genau wie die Methoden `invokeMethod` und `invokeFunction` die Methodenparameter mittels der Signatur `Object... args` übergibt, ist `args` für parameterlose Methoden bei `invoke` null, während bei `invokeMethod` und `invokeFunction` in diesem Fall ein leeres Array übergeben wird. Die vom nativen Code aufgerufene JNI-Funktion `GetArrayLength` führt zu einem Absturz der JVM wenn ihr ein Nullpointer übergeben wird. Um dies zu verhindern wurde in der `invoke()`-Methode des `PHPInvocationHandlers` ein Test eingeführt, ob das Parameterarray `null` ist. In diesem Fall wird einfach ein leeres Object-Array erzeugt, und an den nativen Code übergeben. So werden Abstürze der JVM vermieden.

5.7.4 Garbage Collection

Anders als bei den Testanwendungen aus Kapitel 4 in denen die JVM und der in ihr eingebettete PHP-Interpreter nur sehr kurz laufen, muss innerhalb eines Application Servers darauf geachtet werden, dass reservierter Speicher wieder freigegeben wird wenn er nicht mehr benötigt wird. Java-Anwendungen können hierbei auf die Java Garbage Collection vertrauen, allerdings besteht Turpitude zu einem Großteil aus nativem Code. Da bei der Entwicklung der Bibliothek schon auf eine saubere Speicherverwaltung geachtet wurde traten im Turpitude-Code selbst keine Speicherlecks auf, und es ist davon auszugehen, dass auch der PHP-Interpreter weitestgehend frei von solchen Fehlern ist. Allerdings verbrauchen auch die PHP-Skripte selbst Speicher. Um PHP-Programmierer von der Speicherverwaltung zu befreien enthält die Zend-Engine eine eigene Garbage Collection, und jeder *zval* hat ein Attribut `refcount`, welches die

Anzahl der auf ihn gehaltenen Referenzen angibt. In regelmäßigen Abständen wird für jeden `zval` überprüft, ob noch Referenzen auf ihn vorhanden sind. Ist dies nicht der Fall wird er zerstört und der von ihm belegte Speicher somit wieder freigegeben. Jeder von Turpitude als PHPObject an die Java-Applikation weitergegebene `zval`-Pointer zählt ebenfalls als Referenz. Wird das PHPObject von der Java Garbage Collection aufgeräumt muss dafür gesorgt werden, dass der PHP-Referenzzähler wieder um eins dekrementiert wird. Hierzu wurde dem PHPObject ebenfalls eine `finalize()`-Methode hinzugefügt. In dieser wird die ebenfalls neue, native Methode `destroy()` aufgerufen, in der schließlich der im ByteBuffer gespeicherte `zval`-Pointer ausgelesen, und dessen Referenzzähler herabgesetzt wird.

5.8 Fazit

In diesem Kapitel wurden zunächst die Unterschiede zwischen EJB 3.0 und früheren Versionen erörtert, diese Erkenntnise wurden dann genutzt um eine Beispielanwendung zu entwickeln. Hierbei zeigten sich die immensen Vorteile die EJB 3.0 gegenüber EJB 2.0 bietet, besonders die Tatsache dass wenig bis gar keine externe Konfiguration der Applikationen erfolgen muss, um zumindest eine funktionierende Anwendung zu erhalten. Dies spart nicht nur erheblich Entwicklungszeit, sondern entbindet den Entwickler auch von implementationsspezifischem Wissen über Struktur und Inhalt diverser Konfigurationsdateien. Die Entscheidung EJB 3.0 zu benutzen wurde in keinster Weise bereut.

Nachdem es gelungen war den JBoss, wie in 5.3.1 beschrieben, so zu konfigurieren, dass die PHP-Implementierung für alle in ihm deployten Dienste verfügbar war, war das eigentliche Entwickeln der Enterprise Java Beans unerwartet einfach, und beanspruchte weniger Zeit als zuvor gedacht. Diese Zeitersparnis wurde allerdings leider von den nötigen Änderungen an Turpitude relativiert. Andererseits war das Finden und Beheben solcher Fehler ausdrücklich Ziel dieses Kapitels.

Ein weiteres Problem das auftrat und dessen Analyse sehr viel Zeit benötigte war ein Fehler in der benutzten JBoss-Version, der auftritt wenn man diese in einer Java 6 VM ausführt: Um Daten zwischen Client und Server zu übertragen serialisiert JBoss die zu übertragenden Objekte. Diese werden dann deserialisiert, und auf der anderen Seite mittels der Methode `loadClass()` des Klassenladers wieder geladen. Entgegen der *Java Language Specification* funktionierte diese Methode bis einschliesslich Java 5 auch mit Objektarrays, dies wurde aber mit Version 6 behoben. Da aber JBoss 4.0.5 diese Methode im Zusammenhang mit Objektarrays noch benutzt musste während der Entwicklung auf den JBoss in der neuesten Version 5 umgestiegen werden, obwohl sich dieser noch in einem Betastatus befand.

Alles in allem hat sich allerdings gezeigt, dass Turpitude durchaus geeignet ist um es im geforderten Kontext einzusetzen. Es traten - abgesehen von den in 5.7 beschriebenen Fehlern - keine größeren Probleme auf, und es konnten die geforderten

Enterprise Java Beans mit ihrem vollen Funktionsumfang realisiert werden. Auf eine Implementierung von Entity Beans wurde absichtlich verzichtet, zum einen wegen der neuen *Java Persistance Api*, und zum anderen weil sinnlos erscheint, dass eine Java-Applikation die Persistierung von Daten in ein PHP-Skript auslagert, welches dann völlig an den Konventionen eines Application Servers vorbei Verbindungen zu Datenbankservern öffnet oder Dateien manipuliert. Auch auf den Aspekt einer möglichen Automatisierung des Deployments und des Erstellens des nötigen Java-Quelltextes wurde eingegangen, und es wurden mögliche Ansätze zur Erfüllung dieser Anforderungen aufgezeigt.

Kapitel 6

Fazit und Ausblick

Aufgabe dieser Diplomarbeit war es PHP in die Welt der Java Enterprise Edition Application Server einzuführen. Zum einen um eine einfachere Kommunikation zwischen Anwendungen der beiden Programmiersprachen zu ermöglichen, und zum anderen um dem PHP-Anwender zu erlauben, die Vorteile die ein solcher Application Server und Java an sich bietet in vollem Umfang auszunutzen. Hierzu wurde zunächst evaluiert welche Möglichkeiten überhaupt existieren eine Skriptsprache in Java einzubetten, und welche existierenden Technologien hierzu genutzt werden könnten. Insbesondere wurden zwei Technologien beleuchtet, das Bean Scripting Framework vom Apache Jakarta Projekt, und der Java Specification Request 223. Letzterer wurde schliesslich auch ausgewählt um die Aufgabe zu lösen.

In Kapitel 4 wurde erläutert wie eine Bibliothek namens Turpitude erstellt wurde, die den JSR 223 implementiert, und so die Ausführung von PHP-Skripten innerhalb einer Java-Anwendung, sowie den Zugriff auf Objekte, Funktionen und Methoden in diesen Skrioten erlaubt. Darüber hinaus ermöglicht diese Bibliothek dem PHP-Entwickler den Zugriff auf die komplette Funktionalität der Java-Umgebung, sowie den Datenaustausch zwischen den beiden Programmiersprachen. Objekte der jeweils anderen Sprache werden nicht als bloße Kopien der enthaltenen Daten, sondern als echte Referenzen vorgehalten, was einen transparenten Umgang mit ihnen in der Wirtsprogrammiersprache erlaubt. Diese Möglichkeiten machen Turpitude einzigartig, es existiert kein Softwarepaket das ähnliche Möglichkeiten eröffnet. Dieser Teil der Aufgabe nahm einen erheblichen Teil der zur Verfügung stehenden Zeit in Anspruch, weshalb entschieden wurde bei den verbleibenden Teilen an Umfang zu kürzen.

In Kapitel 5 wird schließlich ein mögliches Einsatzszenario für Turpitude erforscht: Das Schreiben von Enterprise Java Beans in PHP. Es wurden erfolgreich sowohl Stateful- als auch Stateless Session Beans und Message Driven Beans in PHP geschrieben und deployt. Die verbliebene Zeit reichte nicht aus einen Automatismus zu entwickeln, der dem PHP-Entwickler das Schreiben von Java-Quelltext komplett

abnimmt, allerdings wurden die Lösungen so gewählt, dass die Implementierung eines solchen weiterhin möglich bleibt. Es wurde somit gezeigt dass sich Turpitude durchaus eignet um in den Systemen die in der Abteilung *i::Dev* und der Firma 1&1 benötigt werden eingesetzt zu werden.

Zusammenfassend wird die Aufgabenstellung als vollständig gelöst gewertet. Zwar wurde sie an einigen Stellen gekürzt, aber nur um an anderen Stellen eine Vertiefung zu erlauben. Noch muss sich Turpitude zwar im echten Produktiveinsatz beweisen, der Autor hat diesbezüglich aber keinerlei Bedenken, da die bisher durchgeführten Tests nichts gegenteiliges vermuten lassen. An der Bibliothek selbst sind noch eine Vielzahl von Verbesserungen möglich, sowohl was die Qualität des Quelltextes als auch was den Umfang der gebotenen Funktionalität angeht. Beispielsweise könnte der Array-Zugriff in PHP auf alle Java-Objekte ausgedehnt werden, die das Interface Iteratable implementieren. Außerdem wäre eine weitere Vereinfachung des Zugriffs auf Methoden und Attribute wünschenswert, mit dem Ziel den Benutzer komplett von der Notwendigkeit die JNI-Kodierungen zu kennen zu befreien. Auch bessere Kontrolle über die Typisierung beim Methodenaufruf und beim Zugriff auf Attribute wäre wünschenswert.

Es besteht sowohl innerhalb als auch außerhalb von 1&1 ein großes Interesse an Turpitude, und es sind viele Bereiche abseits der Web- und Enterprise-Welt denkbar in denen die Bibliothek sinnvoll eingesetzt werden könnte. Win Beipiel hierfür wären ein Plugin für die Entwicklungsumgebung Eclipse das eine bessere Integration von PHP bietet und so ein komfortables Entwickeln von PHP-Skripten mit Eclipse erlauben würde. Dem kommt zugute, dass Turpitude unter einer Open Source Lizenz veröffentlicht werden wird, um eine möglichst weite Verbreitung der Bibliothek zu erlauben. Die Möglichkeiten, die Turpitude eröffnet erscheinen endlos, und der Autor freut sich auf Projekte die mit Hilfe der Bibliothek erstellt werden.

Literaturverzeichnis

[All06] Paul Allen. *Service Orientation, winning strategies and best practices*. Cambridge University Press, 2006. 13-978-0-521-84336-2. [zitiert auf S. 8]

[ANT06] *The Apache Ant Project*. Apache Foundation, 2006. http://ant.apache.org/. [zitiert auf S. 74]

[APA06] *The Apache HTTP Server Project*. The Apache Software Foundation, 1999-2006. http://httpd.apache.org/. [zitiert auf S. 10]

[BEA07] *BeanShell - Lightweight Scripting for Java*. 2007. http://www.beanshell.org/. [zitiert auf S. 20]

[Boy07] Norris Boyd. *Rhino: JavaScript for Java*. Mozilla Foundation, 2007. http://www.mozilla.org/rhino/. [zitiert auf S. 20]

[BRI06] *php/Java bridge*. php/Java bridge Project Team, 2006. http://php-java-bridge.sourceforge.net/pjb/. [zitiert auf S. 15, 68]

[BSD06] *The BSD License*. Open Source Initiative, 2006. http://www.opensource.org/licenses/bsd-license.php. [zitiert auf S. 11]

[BSF06] *Jakarta BSF - Bean Scripting Framework*. Apache Software Foundation, 2002-2006. http://jakarta.apache.org/bsf/. [zitiert auf S. 21]

[CAU06] *Quercus: PHP in Java*. Caucho Technology, 1998-2006. http://www.caucho.com/resin-3.0/quercus/. [zitiert auf S. 21]

[DEP07] *Dependency Injection*. Wikipedia, Stand 19. Januar 2007. http://de.wikipedia.org/wiki/Dependency_Injection. [zitiert auf S. 73]

[ea06a] Grogan et. al. *JSR 223: Scripting for the Java Platform*. Sun Microsystems, 1995-2006. http://jcp.org/en/jsr/detail?id=223. [zitiert auf S. 23, 101]

[ea06b] Grogan et. al. *JSR 223: Scripting for the Java Platform*. Sun Microsystems, 1995-2006. http://jcp.org/aboutJava/communityprocess/pr/jsr223/index.html. [zitiert auf S. 27]

[Eck03] Bruce Eckel. *Strong Typing vs. Strong Testing*. 2003. http://www.mindview.net/WebLog/log-0025. [zitiert auf S. 9]

[EJB07] *EJB Specifications*. Sun Microsystems Inc., 1994-2007. http://java.sun.com/products/ejb/docs.html. [zitiert auf S. 14, 72, 80]

[Erl04] Thomas Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, 2004. 0-13-142898-5. [zitiert auf S. 8]

[Gel05] Christian Gellweiler. *Implementierung einer Protokollbrucke zur Kommunikation mit J2EE Application Servern uber RMI mit PHP*. 2005. [zitiert auf S. 14, 15, 113]

[Gje05] Gleditsch; Gjermshus. *Cross-Referencing Linux*. 2005. http://lxr.linux.no/. [zitiert auf S. 68]

[Gro04] W3C Soap Working Group. *SOAP Version 1.2 specification*. World Wide Web Consortium, 2004. http://www.w3.org/TR/soap/. [zitiert auf S. 12]

[GRO07] *Groovy: An agile dynamic language for the Java Platform*. 2007. http://groovy.codehaus.org/. [zitiert auf S. 20]

[HIB06] *Hibernate - Relational Persistence for Java and .NET*. Red Hat Middleware LLC., 2006. http://www.hibernate.org/. [zitiert auf S. 73]

[JAC07] *The Tcl/Java Project*. 2007. http://tcljava.sourceforge.net/docs/website/index.html. [zitiert auf S. 20]

[JAR03] *JAR File Specification*. Sun Microsystems Inc, 2003. http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html. [zitiert auf S. 36]

[JAV04] *Annotations*. Sun Microsystems Inc., 2004. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html. [zitiert auf S. 73]

[JAV06a] *Java Technology*. Sun Microsystems, Inc., 1994-2006. http://java.sun.com. [zitiert auf S. 44, 101]

[JAV06b] *Javadoc Tool Homepage*. Sun Microsystems, Inc., 1994-2006. http://java.sun.com/j2se/javadoc/. [zitiert auf S. 11]

[JBO06] *JBoss Enterprise Middleware Suite*. Red Hat Middleware LLC, 2006. http://www.jboss.com/. [zitiert auf S. 73]

[JEE06] *Java EE Technologies at a Glance*. Sun Microsystems Inc., 1994-2006. http://java.sun.com/j2ee/5.0/index.jsp. [zitiert auf S. 13]

[JMS07] *Java Message Service*. Sun Microsystems Inc., 1994-2007. http://java.sun.com/products/jms. [zitiert auf S. 13]

[JNI06] *Java Native Interface 5.0 Specification*. Sun Microsystems Inc., 2004-2006. http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html. [zitiert auf S. 16, 17, 51, 97, 105, 113, 114]

[JYT07] *The Jython Project*. 2007. http://www.jython.org/Project/index.html. [zitiert auf S. 20]

[Lia99] Sheng Liang. *The Java Native Interface*. Addison-Wesley Longman, 1999. 978-0201325775. [zitiert auf S. 16]

[MyS07] *MySQL - The world's most popular open source database*. MySQL AB, 1995-2007. http://mysql.com/. [zitiert auf S. 10]

[OMG06] *The Object Management Group*. 2006. http://www.omg.org/. [zitiert auf S. 7]

[PHP06a] *PHP Handbuch*. PHP-Dokumentationsgruppe, 1997-2006. http://de3.php.net/manual/. [zitiert auf S. 9, 40]

[PHP06b] *PHP Homepage*. The PHP Group, 1997-2006. http://php.net/. [zitiert auf S. 44, 68, 101]

[PHP07] *PHP Cross Reference*. Zend Technologies Ltd., 2005-2007. http://lxr.php.net/. [zitiert auf S. 68]

[SYB07] *Sybase Adaptive Server Enterprise*. Sybase Inc., 2007. http://www.sybase.com/. [zitiert auf S. 10]

[Wee01] Christensen; Curbera; Meredith; Weerawarana. *Web Services Description Language*. World Wide Web Consortium, 2001. http://www.w3.org/TR/wsdl/. [zitiert auf S. 11, 12]

[WSI06] *Web Services Interopability Group*. 2006. http://www.ws-i.org/. [zitiert auf S. 12]

[XDO05] *XDoclet: Attribute-Oriented Programming*. XDoclet Team, 2000-2005. http://xdoclet.sourceforge.net/. [zitiert auf S. 73]

[XPH06] *XP-Framework Homepage*. The XP team, 2001-2006. http://xp-framework.info/. [zitiert auf S. 11]

[ZEN06] *Zend Engine version 2.0: Feature Overview and Design*. Zend Technologies Ltd., 2006. http://www.zend.com/zend/zend-engine-summary.php. [zitiert auf S. 9]

Appendices

Anhang A

Turpitude - API-Dokumentation der PHP-Klassen

Turpitude definiert einige neue PHP-Klassen, an dieser Stelle sollen deren Methoden und Eigenschaften kurz erläutert werden. Typenbezeichner, Klassennamen und Methoden- beziehungsweise Feldsignaturen müssen, wenn sie als String übergeben werden, immer speziell kodiert sein. Siehe hierzu [JNI06].

A.1 TurpitudeEnvironment

Die Klasse `TurpitudeEnvironment` ist Basis aller Java-Zugriffe aus PHP heraus. Sie kann entweder jederzeit wie jede normale PHP-Klasse per `new` erzeugt werden, eine Instanz wird beim Start des Interpreters in das Superglobal `$_SERVER` injiziert. Sie bietet folgende Methoden:

- `TurpitudeJavaClass findClass(string classname)`
  Diese Methode erstellt aus einem String, der einen Klassennamen enthält, ein Klassenobjekt. Der Klassenname muss JNI-kodiert sein. Siehe hierzu [JNI06].
- `void throwNew(string classname, string message)`
  Wirft eine Java-Exception des übergebenen Typs mit der übergebenen Nachricht.
- `void throw(TurpitudeJavaObject<java.lang.Throwable> ex)`
  Wirft die übergebene Exception. `ex` muss vom Typ `TurpitudeJavaObject` sein, und ein Java-Objekt des Typs `java.lang.Throwable` enthalten.
- 1. `bool instanceOf(TurpitudeJavaObject a, TurpitudeJavaClass b)`
  2. `bool instanceOf(TurpitudeJavaObject a, String b)`

Gibt `true` zurück, wenn `a` ein Java-Objekt ist, und bei 1. eine Instanz der in `b` repräsentierten Java-Klasse b, bei 2. eine Instanz der im String `b` beschriebenen Klasse ist.

- `TurpitudeJavaObject exceptionOccurred()`
  Gibt die anliegende Java-Exception zurück. Liegt keine Exception vor, gibt die Methode `null` zurück.

- `void exceptionClear()`
  Löscht alle anliegenden Java-Exceptions.

- `TurpitudeJavaArray newArray(string type, int len)`
  Erzeugt ein Java-Array des gewünschten Typs, mit der gewünschten Länge.

- `TurpitudeJavaObject<ScriptContext> getScriptContext()`
  Gibt den ScriptContext der ScriptEngine zurück.

A.2 TurpitudeJavaClass

`TurpitudeJavaClass` kapselt eine Java-Klasse.

- `TurpitudeJavaMethod findMethod(string name, string sig)`
  gibt eine Java-Methode mit dem übergebenen Namen und Signatur zurück

- `TurpitudeJavaMethod findStaticMethod(string name, string sig)`
  gibt eine statische Java-Methode mit dem übergebenen Namen und Signatur zurück

- `TurpitudeJavaMethod findConstructor(string sig)`
  gibt einen Konstruktor mit der übergebenen Signatur zurück

- `mixed invokeStatic(TurpitudeJavaMethod m, args...)`
  ruft eine statische Methode auf

- `mixed getStatic(string name, string sig)`
  gibt ein statisches Feld der Klasse zurück

- `void setStatic(string name, string sig, mixed val)`
  setzt ein statisches Feld der Klasse

- `TurpitudeJavaObject create(TurpitudeJavaMethod constr, args...)`
  erzeugt eine Instanz der Klasse

- `bool isCastable(TurpitudeJavaObject a)`
  gibt `true` zurück, wenn `a` auf diese Klasse castbar ist.

A.3 TurpitudeJavaMethod

`TurpitudeJavaMethod` kapselt eine Java-Methode. Sie besitzt keine Methoden, und wird nur als Parameter in Methoden anderer Objekte verwendet.

A.4 TurpitudeJavaObject

`TurpitudeJavaObject` kapselt ein Java-Objekt. Neben den unten genannten Methoden erlaubt es auch den direkten Aufruf von Methoden sowie den direkten Zugriff auf Attribute in gewohnter Form.

- `mixed javaInvoke(TurpitudeJavaMethod m, args...)`
  ruft eine Methode des gekapselten Java-Objektes auf
- `mixed javaGet(string name, string sig)`
  gibt den Wert eines Felds des gekapselten Java-Objekts zurück
- `void javaSet(string name, string sig, mixed val`
  setzt den Wert eines Feldes des gekapselten Java-Objekts

A.5 TurpitudeJavaArray

`TurpitudeJavaArray` kapselt ein Java-Array. Neben den unten genannten Methoden erlaubt es auch den direkten Zugriff auf Elemente in gewohnter Form.

- `int getLength()`
  gibt die Länge des Arrays zurück
- `mixed get(int index)`
  gibt das Element an der Stelle `index` zurück
- `void set(int index, mixed val)`
  setzt das Element an der Stelle `index`

Anhang B

Turpitude - Programmierbeispiele

Turpitude ist eine JSR223-Implementation [ea06a], die es einem Anwender erlaubt PHP-Skripte aus Java heraus auszuführen, Funktionen und Objektmethoden innerhalb solcher Skripte aufzurufen und Rückgabewerte dieser Skripte, Funktions- und Methodenaufrufe zurückzuerhalten. Ausserdem erweitert Turpitude den PHP-Interpreter um die Möglichkeit aus PHP heraus Java-Objekte zu erzeugen und Java-Methoden auf diesen Objekten aufzurufen. An dieser Stelle soll eine kurze Einführung in die notwendigen Vorbereitungen, sowie in die API sowohl in Java als auch in PHP erfolgen. Diese Dokumentation richtet sich vor allem an Entwickler die planen mit Turpitude zu arbeiten.

B.1 Installation

Die Installationsanweisungen in diesem Kapitel beziehen sich im Allgemeinen auf ein unixoides System, sind aber leicht für Windows anpassbar.

B.1.1 Java

Zunächst muss ein Java-SDK der Version 6.0 oder höher vorhanden sein, zu finden auf der Java-Homepage [JAV06a]. Das Installationsverzeichnis des SDK wird im weiteren Verlauf als *JAVA_HOME* bezeichnet.

B.1.2 PHP

Weiterhin muss PHP aus den Quellen übersetzt werden. Hierzu muss zunächst der PHP-Sourcecode in einer Version 5.2.0 oder höher von der PHP-Homepage [PHP06b] oder direkt aus dem PHP CVS-Repository heruntergeladen werden. Das Verzeichnis der PHP-Quellen wird im weiteren Verlauf als *PHP_HOME* bezeichnet. PHP kann

nun mittels des Befehls *configure* konfiguriert werden, der Anwender kann hier jeden gewohnten Parameter benutzen, wichtig ist nur dass der Paramter *enable-embed* auf *shared* gesetzt wird. Nach dieser Konfiguration kann PHP mittels des Befehls *make* übersetzt werden:

```
~ # cd $PHP_HOME
php-5.2.0 # ./configure --enable-embed=shared
php-5.2.0 # ./make
```

Listing B.1: Konfigurieren und Übersetzen von PHP

Im Verzeichnis *PHP_HOME/libs* sollte sich nun eine Bibliothek names *libphp5.so* befinden.

B.1.3 Turpitude

Wechseln Sie nun in das Turpitude-Verzeichnis und öffnen Sie die Datei *Makefile* mit einem Texteditor ihrer Wahl. Ändern sie die gekennzeichneten Zeilen ihrem System entsprechend:

```
# vim Makefile
JAVA_HOME = /home/nsn/jdk1.6.0
PHP_HOME = /home/nsn/devel/php/php-5.2.0
```

Listing B.2: Anpassen des Makefiles

Ein Aufruf des Befehls *make* erzeugt nun zwei Dateien: *turpitude.jar*, welches die benötigten Java-Klassen enthält, sowie ein *libturpitude.so*.

B.2 Ausführen

Um nun die beigefügten Beispiele auszuführen, oder um eigene Applikationen entwickeln zu können muss die Java-Laufzeitumgebung entsprechend angepasst werden: stellen Sie zunächst sicher dass sich die in B.1.2 erzeugte libphp5.so in einem Verzeichnis befindet, aus dem das Betriebssystem Programmbibliotheken laden kann, meist *$LD_LIBRARY_PATH*. Dies geschieht entweder durch ein installieren der libphp5.so in den Systembibliothekspfad, oder in dem die entsprechende Variable umgesetzt wird:

```
# export LD_LIBRARY_PATH=.
```

Listing B.3: Anpassen des Makefiles

Weiterhin muss die in B.1.3 erzeugte libturpitude.so in dem Verzeichnis liegen, aus dem Java Programmbibliotheken lädt. Dieses Verzeichnis steht im Systemproperty *java.library.path*. Auch hier kann entweder die Bibliothek in das entsprechende Verzeichnis kopiert werden, oder der JVM kann beim Start ein alternativer Pfad übergeben werden:

```
# java -cp ./turpitude.jar \
  -Djava.library.path=. \
  net.xp_framework.turpitude.samples.EngineList
```

Listing B.4: Ausführen eines Turpitude-Programmes

Dieser Befehl sollte nun alle verfügbaren ScriptEngines auflisten, Turpitude sollte hier erscheinen.

B.3 Beispiele

B.3.1 Hello World

Als erstes soll nun ein einfaches Programm entstehen, das mittels PHP die Zeichenfolge "Hello World!" auf dem Bildschirm ausgibt. Dazu muss zunächst einmal eine Instanz der PHP-Scriptengine erzeugt werden:

```
ScriptEngineManager mgr = new ScriptEngineManager();
ScriptEngine eng = mgr.getEngineByName("turpitude");
```

Listing B.5: Erzeugen der PHP-ScriptEngine

Der ScriptEngineManager stellt Methoden zum Auffinden von ScriptEngines zur Verfügung, die Methode *getEngineByName()* gibt uns die gewünschte Instanz zurück. Nun kann das Script ausgeführt werden:

```
try {
    eng.eval("echo(\"Hello World!\n\");");
} catch(ScriptException e) {
    System.out.println("ScriptException caught:");
    e.printStackTrace();
}
```

Listing B.6: Hello World Skript

Die Methode *eval()* der ScriptEngine führt ein in einem String gespeichertes Skript aus. Da es sich hierbei um Java-Quelltext handelt ist es nötig etwaige Sonderzeichen mit einem "$backslash$" zu maskieren.

Der Quelltext dieses Programmes befindet sich in der Datei *HelloWorld.java*, und kann mittels des Befehls *make hello* ausgeführt werden.

B.3.2 Skriptdateien ausführen

Wie sich in B.3.1 gezeigt hat erweist es sich als umständlich PHP-Quelltext in einem Java-Programm einzubetten. Alternativ kann das Skript sich auch in einer Datei befinden, was den zusätzlichen Vorteil bringt nicht jedes Mal die Java-Klasse neu übersetzen zu müssen wenn sich das Skript äendert.

Hierzu muss ein *java.io.Reader* erzeugt werden, aus welchem das Skript geladen werden kann:

```
FileReader r = new FileReader(filename);
```

Listing B.7: Laden eines Skriptes aus einer Datei

Diesen Reader kann man nun der ScriptEngine übergeben:

```
Object retval = null;
try {
    retval = eng.eval(r);
} catch(PHPCompileException e) {
    System.out.println("Compile Error:");
    e.printStackTrace();
} catch(PHPEvalException e) {
    System.out.println("Eval Error:");
    e.printStackTrace();
} catch(ScriptException e) {
    System.out.println("ScriptException caught:");
    e.printStackTrace();
}
```

Listing B.8: Übergabe des Readers

Hier zeigen sich gleich zwei Besonderheiten die es bei der Benutzung von Turpitude zu beachten gilt: Neben der vom JSR223-Standard beschriebenen *ScriptException* wirft Turpitude noch *PHPCompileException*s und *PHPEvalException*s, was es dem Anwender erlaubt zwischen Übersetzungs- und Laufzeitfehlern zu unterscheiden. Ausserdem können PHP-Skripte einfach mittels "return" Werte zurückgeben, welche dann in Java entweder als skalare Typen, oder aber als Instanzen der Klasse *PHPObject* repräsentiert werden. Ein PHPObject enthält eine *java.util.HashMap* die alle Attribute des PHP-Objektes unter dem jeweiligen Namen gespeichert hat.

Der Quelltext dieses Programmes befindet sich in der Datei *ScriptExec.java*, und kann mittels des Befehls *make exec* ausgeführt werden.

B.3.3 Skripte Übersetzen

Anstatt Skripte ständig zu laden und auszuführen bietet Turpitude auch die Möglichkeit ein Skript einmal zu übersetzen und dann mehrfach auszuführen. Hierzu implementiert die ScriptEngine das JSR223-Interface *javax.script.Compilable*. Um ein Skript zu übersetzen muss statt *eval()* die Methode *compile()* aufgerufen werden:

```
Compilable comp = (Compilable)eng;
CompiledScript script = comp.compile(src);
```

Listing B.9: Übersetzen eine Skriptes

Ein derart übersetztes Skript läßt sich beliebig oft ausführen:

```
for (int i=0; i<5; i++) {
    System.out.println("executing " + i);
    script.eval();
}
```

Listing B.10: Ausführen des übersetzten Skriptes

Auch bei übersetzten Skripten gelten die Regeln Besonderheiten aus den vorhergehenden Beispielen.

Der Quelltext dieses Programmes befindet sich in der Datei *CompileSample.java*, und kann mittels des Befehls *make compiledscript* ausgeführt werden.

B.3.4 Java-Objekte in PHP

Turpitude erlaubt es dem Anwender aus PHP heraus Java-Objekte zu erzeugen und Methoden auf diesen aufzurufen, allerdings gilt es dabei einige Besonderheiten zu beachten.

Zunächst muss eine Instanz der Klasse *TurpitudeEnvironment* gefunden werden. Dies kann entweder mittels eines einfach "new" geschehen, allerdings findet sich eine solche auch im PHP-Superglobal *$_SERVER*:

```
$turpenv = $_SERVER["TURP_ENV"];
```

Listing B.11: TurpitudeEnvironment-Instanz

Der Name, unter dem sich die Instanz finden läßt, ist in der PHPScriptEngine konfigurierbar, der Default lautet "TURP_ENV". Nachdem dieser Schritt getan ist kann eine Instanz der Klasse *TurpitudeJavaClass*, welche eine Java-Klasse repräsentiert, erzeugt werden:

```
$class = $turpenv->findClass("net/xp_framework/turpitude/samples/ExampleClass");
```

Listing B.12: Java-Klassen finden

Wichtig ist hierbei das Format des Klassennamens - er entspricht der JNI-Syntax [JNI06]. Mit diesem Objekt können nun bereits statische Methoden aufgerufen werden, allerdings nicht ohne zuvor eine Instanz der Klasse *TurpitudeJavaMethod* erzeugt zu haben, diese Klasse beschreibt eine Java-Methode. Um statisch aufrufbare Methoden zu finden muss die Funktion *findStaticMethod()* genutzt werden, welche zwei Parameter erwartet: den Methodennamen und die Methodensignatur, wieder in JNI-Syntax kodiert:

```
$method = $class->findStaticMethod('staticMethod', '(I)Ljava/lang/String;');
```

Listing B.13: statische Methoden finden

Jetzt kann die statische Methode aufgerufen werden:

```
$retval = $class->invokeStatic($method, 17);
```

Listing B.14: statischer Methodenaufruf

Der erste Parameter der Methode *invokeStatic()* ist die aufzurufende Methode, alle weiteren werden der Java Virtual Machine als Methodenparameter übergeben, es obliegt also dem Anwender sicherzustellen dass die richtige Anzahl Parameter, als auch die richtigen Typen übergeben werden. Um nun eine Instanz der Java-Klasse erzeugen zu können muss zunächst der Konstruktor gefunden werden, un zwar mittels der Methode *findMethod()*, die wieder die Signatur des gewünschten Konstruktors als Parameter erwartet:

```
$constructor = $class->findConstructor('(ILjava/lang/String;)V');
```

Listing B.15: Konstruktoren finden

Der so gefunden Konstruktor kann nun als erster Parameter für die Methode *create()* verwandt werden, jeder weitere Parameter wird wieder an die JVM weitergeleitet. Auf diese Weise erhält man ein Objekt der Klasse *TurpitudeJavaObject*:

```
$instance = $class->create($constructor, 1337, 'eleet');
```

Listing B.16: Konstruktoren finden

Um nun Attribute dieser Instanz auslesen und setzen zu können stehen die beiden Methoden *javaGet()* und *javaSet()* zur Verfügung. Erstere erwartet zwei Parameter, den Namen des zu setzenden Attributes und dessen Signatur, wiederum JNI-kodiert. Letztere erwartet einen dritten Parameter, den neuen Wert:

```
$int = $instance->javaGet('intval', 'I');
$instance->javaSet('intval', 'I', 666);
```

Listing B.17: Attribute lesen und schreiben

Man beachte allerdings: auf diese Weise lassen sich nicht nur öffentliche Attribute lesen und schreiben, sondern auch private. Auch hier obliegt dem Anwender eine gewisse Sorgfaltspflicht. Will der Anwender nur auf öffentliche Attribute zugreiffen kann er dies wie gewohnt bei ganz normalen PHP-Objekten tun:

```
$int = $instance->intval;
$instance->intval = 666;
```

Listing B.18: Attribute lesen und schreiben

Weiterhin lassen sich natürlich auch Methoden des Java-Objektes aufrufen, allerdings nicht ohne vorher wieder ein TurpitudeJavaMethod-Objekt zu erzeugen, und zwar unter Verwendung der Methode *findMethod()*, welche analog zu *findStaticMethod()* funktioniert. Die derart erzeugte Methode wird - ähnlich wie bei statischen Methoden - als erster Parameter für die Methode *javaInvoke()* verwandt:

```
$method = $class->findMethod('setValues', '(ILjava/lang/String;)V');
$instance->javaInvoke($method, 1338, 'eleeter');
```

Listing B.19: Methoden aufrufen

Allerdings können Java-Methoden auch mit etwas weniger Aufwand aufgerufen werden: jeglicher Methodenaufruf auf einem Objekt der Klasse *TurpitudeJavaObject* wird an die JVM weitergeleitet, wenn es sich nicht um einen Aufruf mit dem Namen *javaGet*, *javaSet* oder *javaInvoke* handelt.

```
$result = $instance->getDate();
```

Listing B.20: direkter Methodenaufruf

Das *TurpitudeEnvironment* bietet noch eine weitere nützliche Methode an: *instanceOf*. Sie funktioniert ähnlich wie der Java-Operator *instanceof*, allerdings nur auf Objekten der Klasse *TurpitudeJavaObject*. Diese Methode erwartet als ersten Parameter das zu überprüfende Objekt, und als zweiten Parameter entweder einen JNI-kodierten Klassennamen, oder eine Instanz der Klasse *TurpitudeJavaClass*:

```
$turpenv->instanceOf($instance, 'java/util/Date');
$turpenv->instanceOf($instance, $class);
```

Listing B.21: instanceOf

Schlussendlich soll noch erwähnt werden, dass PHP-Objekte der Klassen *TurpitudeJavaClass* und *TurpitudeJavaObject* nicht als PHPObject, sondern als "echte" Java-Objekte an die JVM zurückgegeben werden.

Der Quelltext dieses Beispieles befindet sich in der Datei *ObjectSample.java*, und kann mittels des Befehls *make objects* ausgeführt werden. Der Quelltext der verwendeten Beispielklasse befindet sich in der Datei *ExampleClass.java*.

B.3.5 Java-Arrays in PHP

Einen Sonderfall stellen Java-Arrays dar. Sie werden in PHP nicht durch ein *TurpitudeJavaObject*, sondern durch eine weitere Klasse, das *TurpitudeJavaArray* repräsentiert. Ein solches Objekt bietet hauptsächlich drei Methoden: *getLength()* gibt die Länge des Java-Arrays zurück, mit *get()* lässt sich ein Element des Arrays auslesen, und *set()* ermöglicht das Setzen eines Elementes:

```
$method = $class->findMethod(
    'getStringArray',
    '()[Ljava/lang/String;');
$array = $instance->javaInvoke($method);
$length = $array->getLength();
$val = $array->get(0);
$array->set(0, 'test');
```

Listing B.22: Normaler Zugriff auf ein TurpitudeJavaArray

Zusätzlich implementiert das *TurpitudeJavaArray* das PHP-Interface *ArrayAccess*, und erlaubt dem Anwender somit den direkten Zugriff auf Elemente mittels des []-Operators:

```
$val = $array[0];
$array[0] = 'test';
```

Listing B.23: Klammern-Operator

Weiterhin implementiert die Klasse das Interface, *IteratorAggregate*, und bietet folglich die Methode *getIterator()* an, die eine Instanz der Klasse *TurpitudeJavaArrayIterator* zurückgibt, welches wiederum das Interface *Iterator* implementiert. Somit ist es möglich auf gewohnte Weise über Java-Arrays zu iterieren:

```
$iterator = $array->getIterator();
while ($iterator->valid()) {
    $row = $iterator->current();
    $key = $iterator->key();
    var_dump($row);
    var_dump($key);
    $iterator->next();
}
```

Listing B.24: Iterator

Die Methoden des *TurpitudeJavaArrayIterator* können allerdings nicht nur direkt aufgerufen werden, wie Klassen die das Interface *Iterator* implementieren kann auch er in einer *foreach*-Schleife verwendet werden:

```
foreach($iterator as $key => $row) {
  var_dump($key);
  var_dump($row);
}
```

Listing B.25: Iterator in einer foreach-Schleife

Weiterhin ist es möglich Java-Arrays direkt zu erzeugen, dazu wird die Methode *newArray()* beim *TurpitudeEnvironment* aufgerufen, die zwei Argumente erwartet, den JNI-kodierten Typen und die gewünschte Größe des Arrays:

```
$arr = $turpenv->newArray('I', 5);
$arr2 = $turpenv->newArray('Ljava/lang/Object;');
```

Listing B.26: Erzeugen von Arrays

Der Quelltext dieses Beispieles befindet sich in der Datei *ArraySample.java*, und kann mittels des Befehls *make arrays* ausgeführt werden. Der Quelltext der verwendeten Beispielklasse befindet sich in der Datei *ExampleClass.java*.

B.3.6 Java-Exceptions in PHP

Ein weiteres wichtiges Merkmal der Programmiersprache Java sind Exceptions. Turpitude bietet eine Reihe von Methoden die das Werfen und Fangen von Java-Exceptions in PHP erlauben.

Zunächst bringt das *TurpitudeEnvironment* zwei Methoden mit die dem Anwender das Werfen von Java-Exceptions erlauben: *throw()* und *throwNew()*. *throw()* erwartet als einzigen Parameter eine Instanz einer Java-Exception und wirft diese direkt, während *throwNew()* zwei Parameter erwartet - den JNI-kodierten Klassennamen der zu werfenden Exception, sowie die Nachricht (message) die diese Exception enthalten soll.

```
$class = $turpenv->findClass('java/lang/Exception');
$constructor = $class->findConstructor('(Ljava/lang/String;)V');
$instance = $class->create($constructor, 'Test');
$turpenv->throw($instance);
...
$turpenv->throwNew('java/lang/IllegalArgumentException', 'Test');
```

Listing B.27: Werfen von Exceptions

Um zu überprüfen, ob eine Exception aufgetreten ist kann die Methode *exceptionOccurred()* genutzt werden, welche die gerade aktuelle Exception zurückgibt:

```
if ($exc = $turpenv->exceptionOccurred()) {
    ...
}
```

Listing B.28: Exceptions aufgetreten?

Hat der Anwender die Fehlerbehandlung abgeschlossen muss die Methode *exceptionClear()* aufgerufen werden, um der JVM mitzuteilen dass sie die Exception als gefangen betrachten soll. Tut der Anwender dies nicht gilt die Exception als nicht behandelt, und liegt weiterhin an:

```
$turpenv->throwNew('java/lang/IllegalArgumentException', 'Test');
if ($exc = $turpenv->exceptionOccurred()) {
    printf(\"Msg: %s\\n\", $exc->toString('()Ljava/lang/String;'));
    $turpenv->exceptionClear();
}
```

Listing B.29: Exceptions behandeln

Der Quelltext dieses Beispieles befindet sich in der Datei *ExceptionSample.java*, und kann mittels des Befehls *make exceptions* ausgeführt werden.

B.3.7 Der ScriptContext

Die JSR223-Spezifikation beschreibt den bevorzugten Weg Daten an das ausgeführte Skript zu übergeben, und Daten aus dem Skript an die Java-Applikation zurück-

zuübermitteln: Den ScriptContext. Dieser nimmt Objekte auf und ist im Skript verfügbar. Bei Turpitude sind diese Objekte allerdings nicht nur Kopien der Java-Objekte, sondern Referenzen. Änderungen an diesen Objekten haben direkte Auswirkungen auf die Java-Applikation.

Zunächst muss dem ScriptContext allerdings ein Objekt zugeführt werden, dies geschieht - nachdem die Referenz des ScriptContext von der ScriptEngine geholt wurde - mittels der Methode *setAttribute()*. Der dritte Parameter beschreibt den sogenannten *Scope*, den Gültigkeitsbereich, des Wertes.

```
ScriptContext ctx = eng.getContext();
StringBuffer sb = new StringBuffer();
sb.append("before Script\n");
ctx.setAttribute("buffer", sb, ScriptContext.ENGINE_SCOPE);
```

Listing B.30: Kontext befüllen

Weitere Methoden Objekte im Kontext abzulegen sowie weitere Details zum Scope entnehmen Sie bitte der Java API-Dokumentation der Klasse *javax.script.ScriptContext*. Nun kann das ausgeführte PHP-Skript auf den StringBuffer im ScriptContext zugreiffen:

```
$ctx = $turpenv->getScriptContext();
$buffer = $ctx->getAttribute(
                    '(Ljava/lang/String;)Ljava/lang/Object;',
                    'buffer'
                );
$buffer->append(
            '(Ljava/lang/String;)Ljava/lang/StringBuffer;',
            'Script Value'
        );
```

Listing B.31: Kontext in PHP

Liest man nun nachdem das Skript beendet ist den StringBuffer aus, enthält er nicht nur die Zeile "before Script", sondern zusätzlich die Zeile "Script Value".

Der Quelltext dieses Beispieles befindet sich in der Datei *ContextSample.java*, und kann mittels des Befehls *make context* ausgeführt werden.

B.3.8 Aufrufen von PHP-Methoden

Bisher wurde hauptsächlich besprochen wie ein PHP-Skript auf Java-Klassen, deren Attribute und Methoden zugreiffen kann. In diesem Abschnitt soll der umgekehrte Weg gegangen, und aus Java heraus Methoden und Top-Level Funktionen eines PHP-Skriptes aufgerufen werden. Hierzu wird zunchst ein solches Skript geschrieben, in dem eine Klasse definiert wird, und das eine Funktion enthält die eine Instanz dieser Klasse zurückgibt:

```
function useless($i) {
    return new foo($i);
}

class foo {
  var $val = '';
  function __construct($i) {
    $this->val = $i;
  }
  function bar($i) {
    return 'foo::bar::'.$i.' ('.$this->val.')';
  }
}
```

Listing B.32: PHP-Skript

Um nun solche Skript-Funktionen aufzurufen definiert der JSR223 ein weiteres Interface, *Invocable*. Laut Spezifikation soll eigentlich lediglich die jeweilige *ScriptEngine* dieses Interface implementieren, allerdings wäre dann nicht klar auf welchem Skript die Funktion aufgerufen werden soll. Deswegen implementiert das *PHPCompiledScript* dieses Interface ebenfalls, und die *PHPScriptEngine* leitet die Interface-Aufrufe an das zuletzt übersetzte Skript weiter. Zunächst muss das Skript also übersetzt werden, siehe hierzu auch Kapitel B.3.8.

```
ScriptEngine eng = mgr.getEngineByName("turpitude");
Compilable comp = (Compilable)eng;
CompiledScript script = comp.compile(/*Source*/);
```

Listing B.33: Übersetzen

Nachdem das Skript übersetzt ist kann die globale Funktion aufgerufen werden:

```
Invocable inv = (Invocable)script;
Object phpobj = inv.invokeFunction("useless", "Function Value");
```

Listing B.34: Funktionsaufruf

Das phpobj hält nun eine Referenz auf die Instanz der PHP-Klasse *foo*. Jetzt kann bei dieser die Methode *bar()* aufgerufen werden:

```
Object retval = inv.invokeMethod(phpobj, "bar", "Method Value");
```

Listing B.35: Methodenaufruf

Die Methoden *invokeFunction()* und *invokeMethod()* haben sehr ähnliche Signaturen, beide erwarten den Namen der Funktion/Methode als String und die zu übergebenden Argumente als weitere Parameter, *invokeMethod()* erwartet zusätzlich ein PHPObjekt auf dem die Methode aufgerufen werden soll.

Ein weiteres Feature des *Invocable*-Interfaces ist die Methode *getInterface()*, die in zwei Ausführungen existiert. Diese Methode gibt eine in der jeweiligen Skriptsprache implementierte Instanz eines übergebenen Interfaces zurück, auf die dann in Java wie auf ein "echtes" Java-Objekt zugegriffen werden kann. Zur Demonstation ein simples Java-Interface:

```
public interface ExampleInterface {
    public String bar(String s);
}
```

Listing B.36: Java-Interface

Zufälligerweise implementiert die PHP-Klasse *foo* bereits dieses Interface, und da wir dies wissen, können wir das phpobj *getInterface()* übergeben, und diese Methode gibt uns ein entsprechendes Java-Objekt zurück, auf dem wir ganz normal arbeiten können:

```
ExampleInterface exint;
exint = inv.getInterface(phpobj, ExampleInterface.class);
exint.bar("interface call");
```

Listing B.37: getInterface() mit übergebenem Objekt

Falls kein passendes Objekt zur Verfügung steht kann der Anwender die Methode *getInterface()* auch ohne den ersten Parameter aufrufen, dann versucht Turpitude anhand des simplen Klassennamens (bei *java.lang.String* wäre dieser *String*) eine passende Klasse zu finden, und mittels des Default-Konstruktors eine Instanz dieser Klasse zu erzeugen. Erweitern wir als das obige PHP-Skript um folgende Zeilen:

```
class ExampleInterface {
  function bar($i) {
    return 'ExampleInterface::bar::'.$i;
  }
}
```

Listing B.38: Interface Implementation in PHP

Können wir einfach folgenden Java-Code benutzen um die Methode *bar()* aufzurufen, diesmal bei der PHP-Klasse *ExampleInterface*:

```
exint = inv.getInterface(ExampleInterface.class);
exint.bar("interface call");
```

Listing B.39: getInterface() mit übergebenes Objekt

Der Quelltext dieses Beispieles befindet sich in der Datei *InvocableSample.java*, und kann mittels des Befehls *make objects* ausgeführt werden. Der Quelltext des verwendeten Beispielinterfaces befindet sich in der Datei *ExampleInterface.java*.

Abbildungsverzeichnis

Tabellenverzeichnis