CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

Individuelles Projekt

Alternatives Backend für Jassda

Johannes Rieken
1. Juni 2005

Gutachter: Prof. Dr. E.-R. Olderog
Betreuer: Dipl.-Inf. M. Möller

Carl von Ossietzky Universität Oldenburg
Department für Informatik
Abteilung für die Entwicklung korrekter Systeme

Hiermit versichere ich, Johannes Rieken, dieses individuelle Projekt eigenständig und nur mit Hilfe der angegebenen Quellen und Hilfsmittel bearbeitet zu haben.

Oldenburg, der 1. Juni 2005 ______________________________

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Zielsetzung . . . 2

1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **3**

2.1 Die Java Virtual Machine . . . 3

2.2 Die Java Platform Debugging Architecture . . . 5

2.3 Jassda . . . 8

2.3.1 Architektur . . . 9

2.3.2 Funktionsweise Jassdas . . . 10

2.3.3 Beispiel: Setzen von Breakpoints . . . 11

2.3.4 Jassda und Exceptions . . . 13

2.4 Aufbau von Class-Dateien . . . 14

2.5 Entwurfsmuster . . . 19

3 Entwurfsentscheidungen **23**

3.1 Integration des Backends . . . 23

3.1.1 Direkte Integration . . . 23

3.1.2 Implementierung des JDIs . . . 24

3.1.3 Implementierung des JDWPs . . . 25

3.2 Vorgehensweisen zur Ereigniserzeugung . . . 26

3.3 Wahl eines Bytecode-Toolkits . . . 29

3.3.1 BCEL . . . 29

3.3.2 Javassist . . . 30

3.3.3 ASM . . . 31

4 Das alternative Backend **33**

4.1 Architektur . . . 34

4.2 Arbeitsweise . . . 37

4.3 Einschränkungen durch die Sun Binary License . . . 39

5 Ergebnisse **41**

5.1 Fazit . . . 41

5.2 Ausblick . . . 42

Kapitel 1

Einleitung

1.1 Motivation

Computerprogramme und Softwaresysteme werden immer größer und komplexer. Dies führt dazu, dass sie nur schwer nachvollziehbar und kaum auf Korrektheit prüfbar sind. Diese Problematik und speziell die Verwendung von Softwaresystemen in sicherheitskritischen Anwendungsfeldern eröffnet ein Spannungsfeld. Daraus lässt sich der Bedarf nach Werkzeugen, die Entwickler nicht nur beim Programmieren, sondern auch beim Verstehen von Softwaresystemen unterstützen, ableiten. Jassda ist ein Werkzeug das dies leistet, indem es Programmabläufe durch eine Sequenz von Ereignissen beschreibt. Das Betreten oder Verlassen einer Methode ist bspw. ein solches Ereignis. Sequenzen von Ereignissen können genutzt werden, um Aussagen über das zeitliche Verhalten von Programmen zu machen oder um Programmabläufe anhand formaler Spezifikationen zu verifizieren.

Es gibt zahlreiche Werkzeuge, die in der Zielsetzung mit Jassda übereinstimmen. Jassda unterscheidet sich jedoch von diesen in der Vorgehensweise. Weit verbreitet ist die Manipulation des Programmcodes. Dabei werden im Quell- oder Maschinencode Anweisungen hinzugefügt, so dass zur Laufzeit entsprechende Ereignisse erzeugt werden. Dieses Vorgehen ist statisch, da bei jeder Änderung der Anforderungen (es sind andere Ereignisse gewünscht) eine erneute Manipulation des Programmcodes erforderlich ist. Jassda nutzt anstelle der Code-Manipulation die Debugging-Mechanismen der Java Platform Debugging Architecture (kurz: JPDA). Diese ermöglichen es, zur Laufzeit eines Programms Ereignisse zu definieren und auf ihre Auslösung zu reagieren. So kann Jassda sehr flexibel wechselnde Anforderungen umsetzen. Bei der Anbindung der zu untersuchenden Programme mittels JPDA an Jassda wird von *Backend* gesprochen. Dieses individuelle Projekt beschäftigt sich mit der Entwicklung eines alternativen Backends, das Ereignisse durch Code-Manipulation erzeugt aber die Flexibilität Jassdas erhält.

1.2 Zielsetzung

Für die Entwicklung des alternativen Backends gelten zwei zentrale Anforderungen. Es soll das klassische Vorgehen der Ereigniserzeugung, also die Manipulation von Code, genutzt werden und das alternative Backend muss sich nahtlos in Jassda integrieren lassen. Letzteres umfasst sowohl die software-technische Anbindung an Jassda als auch die funktionale Gleichheit zum bisherigen Backend. Darüber hinaus soll beim alternativen Backend auf hohe Ausführungsgeschwindigkeit, welche bei Verwendung der JPDA Anlass zur Kritik bietet, geachtet werden. Aus diesen Anforderungen lassen sich Einzelaufgaben ableiten.

1. **Erläuterung technischer Grundlagen.** In dieser Arbeit soll das Werkzeug Jassda erweitert werden. Dazu muss Jassda und seine Technologie bekannt sein. Zudem muss eine technische Grundlage für die Code-Manipulation und die Integration in Jassda geschaffen werden.

2. **Erarbeitung eines Konzepts.** Unterschiedliche Integrationsweisen des alternativen Backends und Techniken zur Code-Manipulation müssen vorgestellt und evaluiert werden. Daraus muss ein Konzept für das Backend abgeleitet werden.

3. **Implementierung des Konzepts.** Das Konzept muss in eine Softwarearchitektur umgewandelt und implementiert werden. Tests mit vorhandenen Konfigurationen müssen die erfolgreiche Anbindung an Jassda und die funktionale Gleichheit beider Backends zeigen.

1.3 Aufbau der Arbeit

Diese Ausarbeitung gliedert sich in fünf Kapiteln. Kapitel 2 behandelt technische Grundlagen mit Schwerpunkt auf die Java™ Technologie. So werden Konzepte der Programmiersprache Java, die Java Platform Debugging Architecture und der Aufbau von Class-Dateien beschrieben. Zudem wird Jassda und seine Arbeitsweise vorgestellt. Abgeschlossen wird das Grundlagenkapitel mit einer kurzen Einführung in Entwurfsmuster. In Kapitel 3 werden die Anforderungen weiter konkretisiert und Entwurfsentscheidungen diskutiert. Daraus ergibt sich ein Konzept für das alternative Backend. Kapitel 4 stellt vor, wie das erarbeitete Konzept als Programm umgesetzt wurde. Dies beinhaltet die Architektur und Arbeitsweise des alternativen Backends. Abgeschlossen wird die Arbeit durch ein Fazit und einen Ausblick in Kapitel 5.

Auf der Internetseite http://jassda.sourceforge.net wird Jassda sowie das alternative Backend veröffentlicht. Durch den Zugriff auf das CVS-Repository kann dort auch der Quelltext bezogen werden.

Kapitel 2

Grundlagen

Für die Bearbeitung des individuellen Projekts sind einige tiefergehende Kenntnisse rund um die Java^{TM} Technologie und Jassda notwendig. Ziel dieses Kapitels soll es sein, diese Kenntnisse zu vermitteln.

In den Abschnitten 2.1 und 2.2 wird die *Java Virtual Machine* vorgestellt und erläutert wie mit Hilfe der *Java Platform Debugging Architecture* ein Programm in der Virtual Machine auf Fehler untersucht wird. Darauf aufbauend zeigt Abschnitt 2.3 die Funktionsweise Jassdas und wie dort die Debugging-Architektur verwendet wird. Zum Verständnis der Bytecode-Manipulation wird in Abschnitt 2.4 der Aufbau von Class-Dateien beschrieben. Den Abschluss dieses Kapitels bilden Entwurfsmuster, ein Thema aus dem Software-Engineering.

2.1 Die Java Virtual Machine

Die Programmiersprache Java ermöglicht Entwicklern plattformunabhängige Programme zu schreiben. Das bedeutet, dass ein einmal übersetztes Programm auf unterschiedlichen Rechnerarchitekturen und Betriebssystemen ausgeführt werden kann.[1] Um diese Plattformunabhängigkeit zu erreichen, wird Java Programmcode nicht direkt in Maschinecode übersetzt, sondern in Java Bytecode. Java Bytecode wird von einer plattform-spezifischen Virtual Machine ausgeführt. Da jede Java Virtual Machine Bytecodeanweisungen gleich interpretiert, sind Java-Programme plattformunabhängig ausführbar. [HC99a].

Mit dem Wort „Java“ wird oft nur die gleichnamige Programmiersprache gemeint. Zentraler Bestandteil der Java^{TM} Technologie ist aber auch die Java Virtual Machine (kurz: JVM). Dabei handelt es sich um einen abstrakten Computer, der Bytecode Befehle ausführen kann. Dieser virtuelle Computer soll nun in einigen Aspekten näher gebracht werden.

An dieser Stelle wird die Repräsentation eines Java Programms in der JVM behandelt. Daher werden die unterschiedlichen Speicher wie Stack, Heap und Method Area in den Vordergrund gestellt. Daneben wird das Konzept des Klassenladers

[1] Java wurde anfangs der 90er Jahre mit dem Slogan *Write once, run everywhere* vermarktet.

erläutert. Weitere Konzepte wie das Sandbox-Prinzip der JVM oder Threads und deren Synchronisation werden an dieser Stelle nicht behandelt. Dazu ist das Buch „The Java Virtual Machine Spezification“ [LY97] zu empfehlen.

Java Stack

Jedem Thread in der JVM ist ein Java Stack zugeordnet. Die Nutzung des Stacks unterscheidet sich hierbei nicht von der in anderen Programmiersprachen. Auf dem Stack werden lokale Variablen, Methodenparameter, Ergebnisse von Zwischenberechnungen sowie Methodenrückgabewerte gespeichert. Auf Stacks wird über die Standardmethoden *Push* und *Pop* zugegriffen. Java Stacks werden in Stack Frames, die alle Werte eines Methodenaufrufs kapseln, organisiert. Der Ablauf, gezeigt in Abbildung 2.1, ist folgender: Wenn eine Methode aufgerufen wird, legt diese einen neuen Stack Frame an. Im Stack Frame werden oben genannte Werte gespeichert. Bei Beendigung entfernt die Methode den Stack Frame wieder und legt dort ggf. einen Rückgabewert ab.

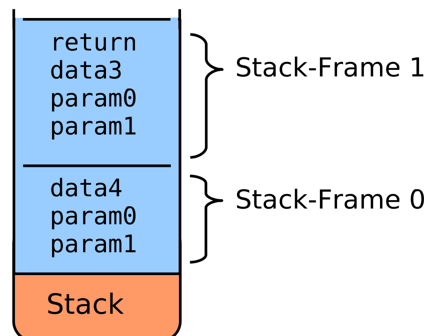


Abbildung 2.1: Gliederung des Java-Stacks in unterschiedliche Stack Frames, welche jeweils einem Methodenaufruf entsprechen.

Tritt bei Ausführung einer Methode eine Exception *(deutsch: Ausnahme)* auf und wird diese nicht abgefangen, wird ihr Frame vom Stack entfernt und die Exception an die aufrufende Methode, dies ist der nächste Frame auf dem Stack, weitergereicht. Fängt diese Methode die Exception auch nicht ab, wird auch ihr Frame vom Stack entfernt. Dies wiederholt sich bis die Exception abgefangen wird oder die letzte Methode, die Main-Methode, verlassen und somit das Programm beendet wird.

PC Register

Zu jedem Java Stack gibt es einen Program-Counter (*deutsch: Programmzähler*). Der Wert des Progammzählers (PC) wird in einem Register, dem PC-Register, abgespeichert. Dessen Wert gibt an, welche Instruktion momentan ausgeführt wird. Nach Ausführung einer Instruktion wird der Programmzähler um eins erhöht. Die Instruktionen befinden sich in der Method Area, die als nächstes beschrieben wird.

Method Area

In der Method Area wird ausführbarer Code gespeichert. Für objektorientierte Programme gilt zu beachten, dass sich unterschiedliche Instanzen einer Klasse nur in den Belegungen ihrer Variablen unterscheiden. Der Bytecode ist für alle Instanzen der selbe und kann daher zwischen ihnen geteilt werden. Beim Bytecode handelt es sich um eine Abfolge von Opcodes. Ihre Abarbeitung erfolgt seriell und kann an beliebiger Stelle unterbrochen werden. Dies ist nötig wenn die JVM angehalten wird oder Nebenläufigkeitsbedingungen unterschiedlicher Threads beachtet werden müssen.

Heap

Der Heap *(deutsch: Haufen)* ist ein gemeinsamer Speicher. Anders als Java Stacks wird er von allen Threads geteilt. Er enthält Klasseninstanzen, globale Variablen sowie Arrays. Der Heap-Speicher wird von einer externen Garbage Collection verwaltet, so dass der Java-Programmierer nicht die Möglichkeit hat Speicherbereiche explizit freizugegeben. Objekte, auf die keine Referenz mehr zeigt, werden von der Garbage Collection aus dem Speicher gelöscht. Auch die Method Area befindet sich im Heap, sie wird jedoch von der Garbage Collection ausgeschlossen.

Native Method Stack

Für den Aufruf nativer Methoden steht ein besonderer Stack bereit. Dieser wird von allen Threads geteilt. Der Programmzähler (PC-Register) eines Java-Stacks ist während der Ausführung von nativen Methoden undefiniert.

Classloader

Der Classloader *(deutsch: Klassenlader)* lädt Klassendefinitionen aus den Class-Dateien. Dies schließt eine Überprüfung und Validierung des Bytecodes ein, auf die im Rahmen dieser Arbeit nicht weiter eingegangen wird.

Klassenlader können hierarchisch angeordnet werden. Die Hierarchie der Klassenlader beginnt immer mit dem Bootstrap-Classloader und dem System-Classloader. Sie sind von JVM vorgegeben und implementieren die angesprochenden Bytecodeprüfungen. In die bestehende Hierarchie können aber auch eigene Klassenlader eingefügt werden. Dies wird häufig genutzt, um Klassen über Netzwerkverbindungen oder aus Archiven zu laden. Eigene Klassenlader ermöglichen aber auch die Manipulation des Bytecodes zum Zeitpunkt des Ladens. Diese Eigenschaft wird sich für das alternative Backend als äußerst nützlich erweisen.

2.2 Die Java Platform Debugging Architecture

Computerprogramme enthalten Fehler. Damit diese vom Entwickler gefunden und behoben werden können, gibt es zahlreiche Strategien. Eine von ihnen ist das Debugging. Beim Debugging können zwei Akteure identifiziert werden. Der *Debuggee*

ist ein Programm, das auf Fehler untersucht wird, und der *Debugger* ist ein Programm, das diese Fehlersuche durchführt. Seit der Version 1.2 unterstützt Java mit der Java Platform Debugging Architecture (kurz: JPDA) diese Form der Fehlersuche. Wie das Debugging in Java funktioniert soll nun vorgestellt und erläutert werden. Abbildung[2] 2.2 zeigt, dass es sich bei der JPDA um eine 3-Schichtenarchitekur handelt. Sie besteht aus *Frontend*, *Communication* und *Backend*. Auf diese Schichten soll nun näher eingegangen werden.

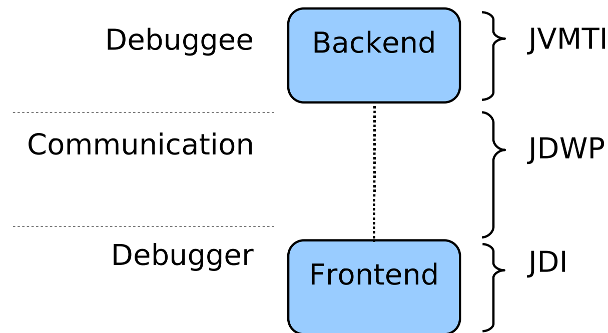


Abbildung 2.2: Die Schichten der Java Platform Debugging Architecture.

Backend

Das Backend ist die unterste Schicht im Architekturmodell der JPDA und muss den Debuggee ausführen sowie Debugging-Befehle befolgen. Üblicherweise handelt es sich bei dem Backend um die Java Virtual Machine. Jedes andere Programm, das Debugging-Befehle korrekt befolgen kann, gilt aber auch als Backend. Befehle werden immer in Form von Paketen des Java Debug Wire Protocols (kurz: JDWP) erteilt.

Frontend

Das Frontend ist die Anschlussstelle zum eigentlichen Debugger. Dieser kann über eine Programmschnittstelle Debugging-Befehle absetzen und den Debuggee steuern. Diese Schnittstelle ist das Java Debug Interface (kurz: JDI). Ein Debugger kann diese dieses selbst implementieren oder eine Standard-Implementierung, die im Java Runtime Environment enthalten ist, nutzen.

Communication

Die Architekturschicht *Communication* spezifiziert das Java Debug Wire Protocol, welches ein Kommunikationsprotokoll ist, das zwischen Backend und Frontend verwendet wird. Bei diesem Protokoll handelt es sich um ein zustandsloses paketorientiertes Protokoll. Die Pakete des JDWPs lassen sich hinsichtlich Aufbau und

[2]Die Abbildung wurde von Sun Microsystems übernommen.

Bedeutung klassifizieren. Bzgl. des Aufbaus kann zwischen Anfrage- und Antwortpaket unterschieden werden. Anfragen erfordern immer eine Antwort. Der genaue Aufbau dieser Pakete ist in Abbildung 2.3 gezeigt. Dort ist zu sehen, dass jedes Paket aus Kopf- und Datenteil besteht. Der Kopfteil ist immer 11 Byte lang und setzt sich wie folgt zusammen. Die vordersten vier Byte codieren die Gesamtlänge des Pakets, ihnen folgt eine Identifikationsnummer, die ebenfalls vier Byte lang ist. Diese Identifikation bezieht sich immer auf ein Anfrage-Antwort Paar. Das neunte Byte im Kopfteil gibt den Typ des Pakets an, also ob es sich um ein Anfrage- oder Antwort-Paket handelt. Die letzten zwei Byte im Kopfteil hängen vom Typ des Pakets ab. Anfrage-Pakete enthalten dort einen Befehlscode, Antwort-Pakete können dort einen Fehler codieren.

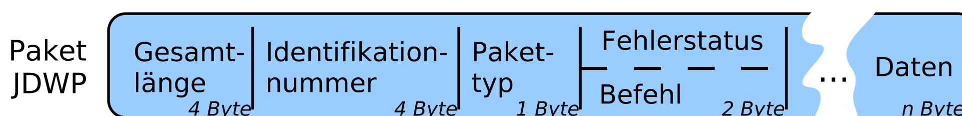


Abbildung 2.3: Aufbau eines JDWP-Pakets.

Neben der Aufteilung in Anfrage- und Antwortbefehle gibt es eine weitere Klassifizierung. Wir haben gesehen, dass Anfrage-Pakete einen konkreten Befehlscode enthalten. Insgesamt gibt es ca. 100 Befehle im JDWP. Der Übersicht wegen werden diese in Befehlsgruppen gegliedert. Dabei handelt es sich immer um logische Einheiten. So gibt es eine Befehlsgruppe mit dem Namen *VirtualMachine*, welche Befehle mit direktem Zusammenhang zur JVM wie bspw. das Beenden selbiger enthält. Eine besondere Befehlsgruppe ist die Gruppe *EventRequest*. Befehle dieser Gruppe erzeugen Wächter in der Virtual Machine. Nach seiner Erzeugung wartet ein Wächter auf das Auftreten eines bestimmten Ereignisses und informiert den Debugger darüber. Die Virtual Machine legt eine Reihe von Ereignissen fest, zu denen ein entsprechender Wächtertyp existieren kann. Diese Ereignisse sind bspw. das Laden von Klassen, das Erreichen eines Breakpoints oder das Auftreten einer Exception. Die Zeitspanne zwischen dem Erzeugen eines Wächters und dem Auftreten des Ereignisses ist nicht vorhersagbar. Daher wird jedem Wächter eine Identifikationsnummer zugeordnet. Diese wird dem Debugger unmittelbar nach Erzeugen des Wächters mitgeteilt. Sobald das entsprechende Ereignis ausgelöst wurde, meldet sich der Wächter mit seiner Identifikationsnummer und den Ereignisdaten beim Debugger. Anhand der Identifikationsnummer kann der Zusammenhang zwischen Ereignisdaten und Wächter hergestellt werden. Wir werden später sehen, dass Jassda und auch das alternative Backend Wächter nutzen, um Programmabläufe zu beobachten.

Die Kommunikation zwischen Backend und Debugger soll noch etwas ausführlicher besprochen werden. Das JDWP definiert nur, wie ausgetauschte Daten zu interpretieren sind. Es wird aber nicht festgelegt, wie der Datenaustausch stattfindet.

Dazu gibt das Konzept der Konnektoren. Die Debugging-Architektur enthält unterschiedliche Konnektoren [Sun01], welche nun vorgestellt werden.

- *Command Line Launching Connector*
  Dieser Konnektor wird genutzt, um die JVM des Debuggee zu starten und sich mit ihr zu verbinden. Der Datenaustausch erfolgt dann über eine Socketverbindung. Der Launching Connector lässt sich in zwei weitere Konnektoren aufteilen. Der *Sun Command Line Launching Connector* startet explizit die Sun Java Virtual Machine, wobei *Raw Command Line Launching Connector* benutzt wird, um ein beliebiges Programm zu starten. Hierbei müssen besondere Pfadangaben und Parameter beachtet werden. Wird dieser Konnektor-Typ genutzt, werden Debugger und Debuggee auf dem selben Rechner ausgeführt.

- *Socket Connector*
  Ermöglicht das Remote-Debugging, da Debugger und Debuggee auf zwei unterschiedlichen Rechnern ausgeführt werden können. Der Datenaustausch erfolgt auch hier über eine Socketverbindung. Es gibt zwei unterschiedliche Typen dieses Konnektors. Der *Socket Attaching Connector* kann genutzt werden, um eine Verbindung zu einer bereits laufenden Applikation aufzubauen. Dies ist nützlich, um Anwendungen im produktiven Einsatz zu untersuchen. Der *Socket Listing Connector* hingegen veranlasst den Debugger zu warten bis eine Verbindung von der Debuggee-VM initiiert wird.

- *Shared Memory Connector*
  Analog zu dem vorangehenden Konnektorenpaar lässt sich dieser Konnektor in *Shared Memory Listing Connector* und *Shared Memory Attaching Connector* aufteilen. Diese Konnektoren kommunizieren nicht über eine Socketverbindung, sondern über gemeinsame Speicherbereiche mit dem Debugger. Sie stehen nur unter Microsoft Windows™ zur Verfügung.

Seit der Java Version 1.5 ist es zudem möglich, eigene Konnektoren bereitzustellen. So können Transportwege erschlossen werden, die bisher nicht abgedeckt wurden. Eigene Konnektoren könnten dann genutzt werden, um Debuggee und Debugger eine verschlüsselte Verbindung nutzen zu lassen oder über Infarot-Verbindungen zwischen zwei Endgeräten zu kommunizieren. Wie Konnektoren und Transportmechanismen implementiert und bereitgestellt werden, wird auf der Internetseite [Sun04d] beschrieben.

2.3 Jassda

Die *Java with assertions debugging architecture* (kurz: Jassda) wurde von Mark Brökens im Rahmen seiner Diplomarbeit entwickelt. Bei Jassda handelt es sich um eine Architektur, mit der Programmabläufe beobachtet werden können. Zur Interpretation dieser Programmabläufe stellt Jassda unterschiedliche Module bereit und bietet die Möglichkeit durch neue Module erweitert zu werden. Ein Programmablauf wird immer durch eine Sequenz von Ereignissen beschrieben. Jassda

unterstützt derzeit drei unterschiedliche Ereignistypen, die sich jeweils auf Methodenaufrufe beziehen.

- *Methodenanfang:* Bevor der Code einer Methode ausgeführt wird, tritt dieses Ereignis auf.
- *Methodenende:* Nach der letzten Codeanweisung in einer Methode wird dieses Ereignis ausgelöst.
- *Ausnahme:* Bei Ausführung der Methode ist eine Exception aufgetreten, die dazu führt, dass die Methode beendet wird. Exceptions beenden die Ausführung einer Methode, sind aber nicht dem regulären Beenden einer Methode zu verwechseln.

Zur Ermittlung dieser Ereignisse nutzt Jassda die *Java Platform Debugging Architecture* (siehe Abschnitt 2.2). Dies unterscheidet Jassda von Programmen mit ähnlicher Zielsetzung. Sie nutzen meist die Manipulation von Quell- oder Bytecode, um Ereignisse im Programmablauf zu erzeugen. Die Verwendung der JPDA ermöglicht das Beobachten von Programmabläufen auf höherer Ebene. Jassda ist weder auf Quell- noch auf Bytecode angewiesen, da durch die JPDA eine direkte Verbindung zum ablaufenden Programm besteht. Daraus ergibt sich, dass Jassda sehr flexibel auf wechselnde Anforderungen reagieren kann. Soll jeder Programmablauf auf unterschiedliche Ereignisse untersucht werden, kann dies in Jassda direkt umgesetzt werden. Bei der klassischen Methode der Code-Manipulation wäre jedesmal eine Änderung und Neuübersetzung notwendig.

Die folgenden Abschnitte sollen klären, wie Jassda funktioniert. Dabei wird der Schwerpunkt auf die Architektur und die generelle Funktionsweise gelegt. Auf das Tracechecker-Modul und seiner theoretischen Grundlage kann nicht eingegangen werden. Dies ist der Diplomarbeit [Brö02] direkt zu entnehmen.

Bemerkung: Im weiteren Verlauf dieser Arbeit werden die Begriffe „Jassda“ und „Debugger“ gleichermaßen verwendet. Korrekterweise müsste von einer JDI-Implementierung gesprochen werden, da es sich bei Jassda nicht um einen Debugger im eigentlichen Sinne handelt.

2.3.1 Architektur

Jassda ist gekennzeichnet durch einen modularen Aufbau und der Eigenschaft durch neue Module erweiterbar zu sein. Dies spiegelt sich in der Architektur wider. Sie besteht aus den drei Schichten *Debuggee*, *Core* und *Module*. Diese sollen nun erläutert werden.

- *Debuggee* - Die Debuggee-Schicht ist die unterste Schicht. Sie kapselt die zu untersuchenden Programme (Debuggees) und bietet eine einheitliche Schnittstelle für weitere Programmkomponenten. Die Verbindung zum Debuggee und der JPDA wird mittels Konnektoren hergestellt (siehe 2.2).

- *Core* - Diese Komponente kapselt die Programmlogik, da sie die Schichten *Debuggee* und *Module* verbindet. Ereignisse, die vom Debuggee stammen, werden hier aufbereitet und an die Module weitergereit. Zudem enthält diese Schicht weitere Komponenten zur Datenhaltung und zur Benutzerinteraktion.

- *Module* - Module implementieren die eigentliche Funktionalität Jassdas. Dazu werden sie von der *Core*-Schicht über das Auftreten von Ereignissen informiert. Jedes Modul entscheidet, wie es die eingehenden Ereignisse behandelt. Für Jassda gibt es momentan zwei Module. Ein Logger-Modul und einen CSP-Tracechecker. Das Logger-Modul protokolliert lediglich den Ablauf eines Programms in eine Datei. Der CSP-Tracechecker validiert Programme anhand CSP_{jassda}-Spezifikationen. Ein Modul muss wissen, an welchen Ereignissen es interessiert ist. Dies wird i.d.R. vom Benutzer über eine Spezifikation festgelegt.

2.3.2 Funktionsweise Jassdas

Jassda kennt derzeit drei Ereignistypen: Methodenanfang, Methodenende und das Auftreten einer Ausnahme. Es soll erläutert werden, wie Jassda die JPDA nutzt, um das Auftreten dieser Ereignisse zu ermitteln. In Abschnitt 2.2 wurde das Konzept der Wächter vorgestellt. Diese können den Debugger über das Auftreten unterschiedlicher Ereignisse informieren. Jassda macht sich dieses Konzept zu Nutze und verwendet einige dieser Wächter. Methodenanfänge und -enden werden jeweils durch einen Anhaltepunkte *(engl. Breakpoint)*, welcher durch den Wächter *BreakpointRequest* erzeugt wird, gekennzeichnet. Exceptions können mit Hilfe des Wächters *ExceptionRequest* gemeldet werden.

Entgegen der Breakpoints kann der *ExceptionRequest* global für alle Klassen des Debuggees installiert werden. Um einen Breakpoint zu setzen, muss zuerst geklärt werden, an welcher Stelle, also in welcher Klasse und Methode, dies geschehen soll. Dazu werden die Module herangezogen. Sie geben an, an welche Ereignisse sie interessiert sind und welchen Typs diese Ereignisse sind. Eine Komponente der Core-Schicht, der Broker, übernimmt das Installieren der Wächter. Dabei geht er nach folgendem Algorithmus vor:

1. Der Broker wird über das Laden einer Klasse informiert. Ist ein Modul an dieser Klasse interessiert, werden alle Methoden dieser Klasse ermittelt.

2. Für jede Methode werden zwei *Dummy*-Ereignisse erzeugt. Diese täuschen den Anfang bzw. das Ende der Methode vor. Diese Ereignisse werden an die Module weitergeleitet.

3. Falls eins der Module an den Ereignissen interessiert ist, erzeugt der Broker einen Breakpoint. In einer Datenbank wird gespeichert, ob der Breakpoint einem Methodenanfang oder -ende zugeordnet werden muss. Das eigentliche Setzen des Breakpoints wird dann durch das JDI an das Debugging-Backend delegiert.

Der Algorithmus setzt voraus, dass der Broker über das Laden einer Klasse informiert wird (siehe Punkt 1). Auch dies geschieht durch einen entsprechenden Wächter, welcher *ClassPrepareRequest* heißt und von Jassda vor Ausführung des Debuggees installiert wird. Um unnötige Ereignisse zu vermeiden, werden die Klassen anhand ihrer Namen gefiltert. Diese Filterung findet bereits im Backend statt, so dass das Datenaufkommen reduziert wird.

Bemerkung: Bereits in seiner Diplomarbeit diskutiert Mark Brökens einige Unzulänglichkeiten der JPDA. Dies gilt besonders für den Wächter *MethodExitRequest*. Obwohl dieser Wächter das Beenden einer Methode signalisiert und somit für Jassda geeignet erscheint, gibt es keine Möglichkeit den Rückgabewert der Methode zu ermitteln oder ihn auf bestimmte Methoden zu beschränken. Dies führte zur Entscheidung, Breakpoints für das Anzeigen eines Methodenanfangs bzw. -endes zu nutzen und Rückgabewerte von Methoden manuell zu ermitteln. Die Diskussion um das *MethodExitEvent* wird von zahlreichen JDPA-Anwendern geführt und wurde schon 1998 als Bug bei SUN gemeldet [Sun98]. Für die Version 1.5, die im Winter 2004 veröffentlicht wurde, war die Behebung dieses Bugs vorgesehen, musste aber aus Zeitgründen verschoben werden. Unter dem Arbeitstitel „Mustang“ wird momentan die Version 1.6 von Java entwickelt. Diese wird voraussichtlich im ersten Halbjahr 2006 erscheinen. In der aktuellen Beta-Version, Mustang(b19), ist der Bug bereits behoben. Es ist also zu erwarten, dass die nächste offizielle Version der JPDA die Möglichkeit bietet Methodenrückgabewerte zu ermitteln.

2.3.3 Beispiel: Setzen von Breakpoints

Es soll nun exemplarisch gezeigt werden, wie Jassda Breakpoints setzt. Das Beispiel beginnt damit, dass der Broker über das Laden einer Klasse informiert wird und mittels der Modul-Spezifikationen erkennt, dass die Klasse von Interesse ist und weiter behandelt werden muss. Um einen Überblick zu schaffen, zeigt Abbildung 2.4 den Ablauf in einem Sequenzdiagramm. Bei dem gezeigten Diagramm stehen die JDWP-Befehle und ihre Abfolge im Vorderung. Eine ausführliche Dokumentation und Übersicht aller JDWP-Befehle enthält die Webseite [Sun04c].

1. Das Backend bereitet eine Klasse für das Laden vor. Da ein entsprechender Wächter installiert ist, wird dies dem Debugger gemeldet.

2. Da der Debugger an der Klasse interessiert ist, werden die Methodeninformationen abgefragt. Der Debuggee sendet Informationen über alle Methoden, die in der Klassen deklariert wurden. Dies umfasst Name und Signatur der Methode, eine Bitmaske der Zugriffsrechte sowie eine Identifikationsnummer.

   Geerbte Methoden werden bei Schritt 2 nicht erfasst. Daher wird dieser für alle Oberklassen in der Vererbungshierarchie und alle implementierten Schnittstellen wiederholt.

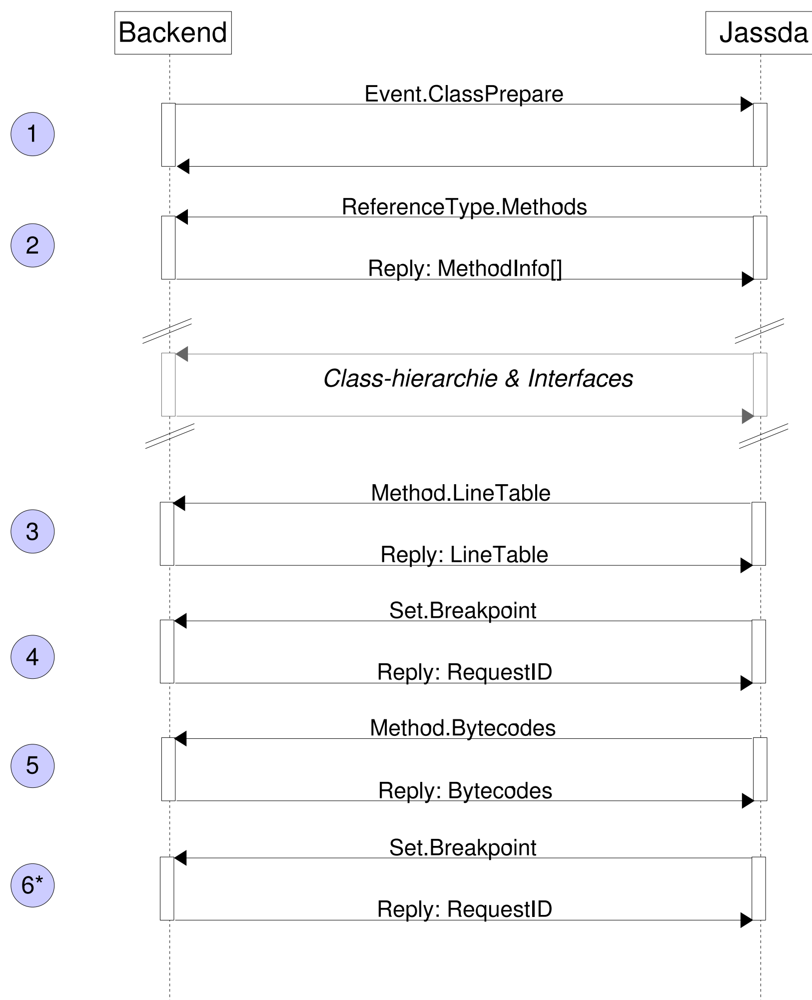


Abbildung 2.4: Setzen der Breakpoints.

Anhand der Modul-Spezifikationen legt Jassda eine Reihe von Methoden fest, die weitergehend betrachten werden müssen. Für jede dieser Methoden werden die Schritte 3-6 ausgeführt:

3. Breakpoints werden direkt im Quelltext für bestimmte Zeilen gesetzt. Aus einer Zeile Quelltext können durch den Compiler mehrere Opcodes erzeugt werden. Der jeweils erste Opcode zu einer Zeile wird durch die *LineNumberTable* beschrieben. Damit Jassda die Breakpoints nicht in die Mitte einer Zeile setzt, wird die LineNumberTable benötigt.

4. Jassda markiert das Betreten einer Methode mit einem Breakpoint in „Zeile 1“. Der erste Breakpoint ist gesetzt.

5. Damit auch das Verlassen einer Methode mit einem Breakpoint versehen werden kann, muss Jassda die Return-Anweisungen lokalisieren. Dazu wird der Bytecode der Methode abgefragt und analysiert.

6. Für jede gefundene *Return*-Anweisung wird ein Breakpoint gesetzt. Sind alle Breakpoints gesetzt, wird mit der nächsten Methode fortgefahren (siehe Punkt 3). Wenn alle Methoden abgearbeitet wurden, ist dieser Prozess beendet.

2.3.4 Jassda und Exceptions

An dieser Stelle soll etwas genauer behandelt werden, wie Jassda mit Exceptions umgeht.[3] Jassda ist daran interessiert, ob eine bestimmte Methode durch das Auftreten einer Exception vorzeitig beendet wurde. Es spielt keine Rolle, ob die Exception direkt in der Methode entstanden ist oder durch eine Kaskade von Exceptions aus anderen Methoden (ins. Methoden anderer Klassen) weitergereicht wurde. Die JPDA verfolgt einen anderen Ansatz, denn sie kennt nur den Entstehungsort einer Exception und ggf. die Codestelle, an der sie abgefangen wurde. Die daraus resultierende Problematik soll anhand Abbildung 2.5 etwas genauer erläutert werden.

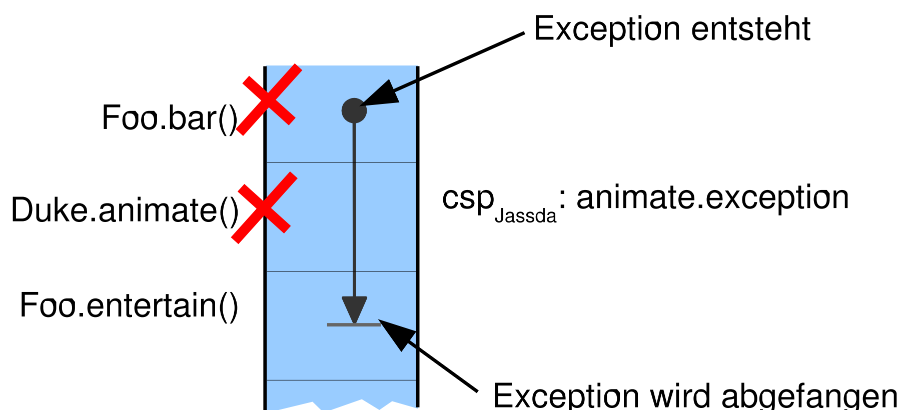


Abbildung 2.5: Ausschnitt des Stack mit mehreren Frames. Jeder Frame entspricht einem Methodenaufruf. Die oberen zwei Frames werden aufgrund einer Exception vom Stack entfernt.

Es wird ein Ausschnitt aus dem Stack gezeigt. In der aktuellen Methode *Foo.bar()* tritt eine Exception auf. Da sie weder hier noch in der Methode *Duke.animate()* abgefangen wird, werden die zwei Methoden beendet und ihre Frames vom Stack entfernt. Die Methode *Foo.entertain()* fängt die aufgetretene Exception ab. Jassda will wissen, ob die Methode *Duke.animate()* aufgrund einer Exception beendet wurde. Dies ist offensichtlich der Fall. Das angedeutete Problem ist folgendes:

[3] Exceptions in Jassda werden an dieser Stelle ausführlicher besprochen, da vor der Entwicklung des alternativen Backends das Exception-Ereignis nicht vollständig implementiert war. Diese Funktion ist nun also neu und hier zum ersten Mal beschrieben.

Das Exception-Ereignis, das Jassda erhält, wird die Methode *Foo.bar()* als Entstehungsort und *Foo.entertain()* als Abfangort nennen. *Duke.animate()*, woran Jassda eigentlich interessiert ist, taucht nicht auf. Ein simples Exception-Ereignis reicht also nicht aus. Neben der Entstehung- und Abfang-Methode müssen auch alle dazwischen liegenden Methoden bekannt sein. Diese Sammlung von Methoden nennt man *Stacktrace* - also die Spur, die sich durch den Stack zieht. In dem obigen Beispiel enthielte der Stacktrace die besagte Methode. Das Exception-Ereignis muss also in zwei Schritten generiert werden. Zunächst wird Jassda über das Auftreten jeder Exception informiert. Dies geschieht durch das Backend und einen entsprechenden Wächter. Daraufhin muss Jassda den Stacktrace ermitteln und genauer untersuchen. Nur wenn dort eine Methode auftaucht, in der Jassda eine Exception erwartet hat, wird das Exception-Ereignis erzeugt. Das eigentliche Exception-Ereignis wird also von Jassda selbst und nicht vom Backend erzeugt bzw. Jassda filtert die Menge der eingehenden Exception-Ereignisse. Wir werden in Abschnitt 3.2 sehen, dass sich die Erzeugung von Exception-Ereignissen durch das alternative Backend geringfügig hiervon unterscheidet.

Begriffsklärung: Backend. Nachdem Jassda und die JPDA eingeführt wurden, kann der Begriff des *Backends* konkretisiert werden. Es wird unter dem Titel *„Alternatives Backend für Jassda“* gearbeitet, aber worum handelt es sich bei einem Backend? Bisher tauchte der Begriff nur in der Architekturbeschreibung der JPDA (siehe Abschnitt 2.2) auf. Dort ist das Backend die Entität, die Debugging-Befehle verstehen und ausführen kann. In Jassda taucht der Begriff des Backends nicht auf. Vergleichbar damit ist aber die unterste Schicht in Jassdas Architektur: die Debuggee-Schicht. Diese Komponente kapselt das Debugging; also die komplette JPDA mit den 3 Schichten Frontend, Communication und Backend. Es gilt also zu unterscheiden zwischen Backend im Sinne der JPDA und Backend als Oberbegriff für die Anbindung des Debuggees. Das alternative Backend soll als ein Programm, das sich in die Debuggee-Schicht eingliedert und Ereignisse durch Bytecode-Manipulation erzeugt, definiert werden. Abbildung 2.6 zeigt einen Überblick und die Unschärfe dieser Defintion. Im weiteren Verlauf der Arbeit wird genauer spezifiert wie das alternative Backend funktioniert und sich in Jassda integriert.

2.4 Aufbau von Class-Dateien

Da sich diese Arbeit mit der Manipulation des Bytecodes beschäftigt, ist ein gewisses Grundverständnis notwendig. Dieser Abschnitt gibt eine kurze Einführung, ausführlichere Informationen bietet die Spezifikation der Java Virtual Machine und des Bytecodes in [LY97].

Java Bytecode wird durch einen Java Compiler erzeugt und in einer Class-Datei gespeichert. Der schematisierte Aufbau einer Class-Datei ist in Abbildung 2.7 gezeigt.

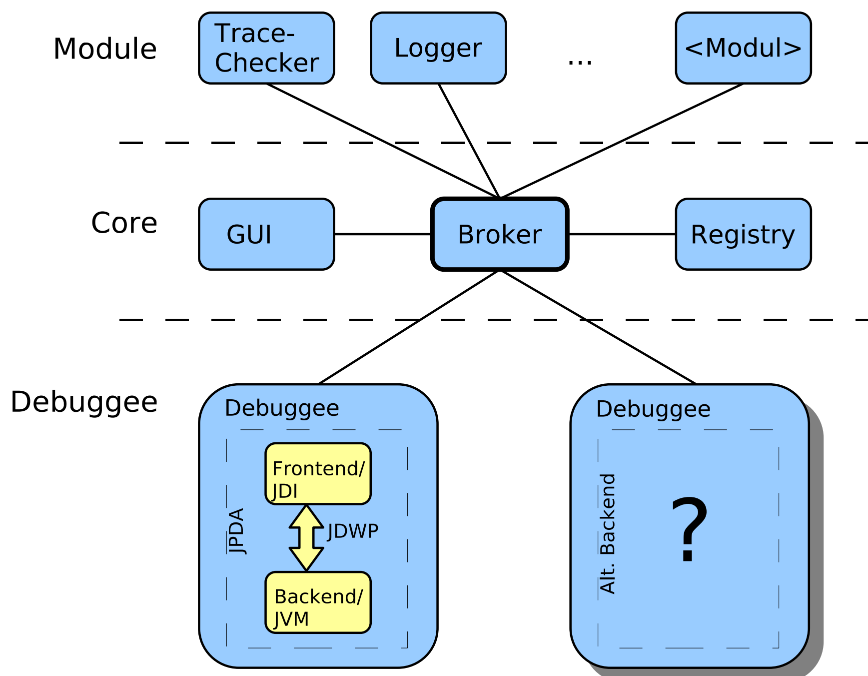


Abbildung 2.6: Die JPDA ist in der Debuggee-Schicht Jassdas gekapselt und kann als Backend betrachtet werden. Das alternative Backend wird sich in die Debugge-Schicht eingliederen, muss aber noch konkretisiert werden.

- *Header*
  Jede Class-Datei beginnt mit dem Header, welcher Versionsinformationen und die „magische“ Number `0xCAFEBABE` zur Identifizierung als Class-Datei enthält.

- *Constant Pool*
  Bezogen auf die Datenmenge, ist der Constant Pool der größte Teil einer Class-Datei. Dabei handelt es sich um eine Tabelle, die unterschiedliche Strukturen speichert. Dies sind beispielsweise Referenzen zu anderen Klassen, Variablen- oder Methodendeskriptoren, Zahlenkonstanten oder Zeichenketten. Jedes Element des Constant Pools wird über einen Index referenziert.

- *Access rights*
  Die Bitmaske *Access rights* legt Art und die Zugriffsmöglichkeit der Klasse fest. So gibt es jeweils ein Element, das angibt, ob es sich bei der Klasse um ein Interface, eine abstrakte oder eine gewöhnliche Klasse handelt. Das gleiche gilt für die Schlüsselwörter `public`, `protected`, `private` und `static`. Auch sie werden durch Elemente der Bitmaske codiert.

- *Class hierachy*
  Hier sind Indizes einer eventuellen Oberklasse sowie implementierter Inter-

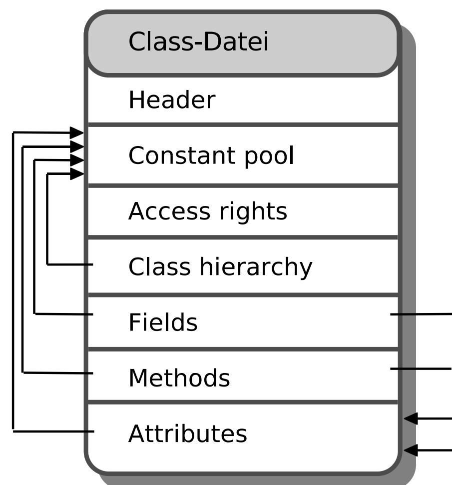


Abbildung 2.7: Schematische Darstellung einer Class-Datei.

faces enthalten. Damit wird die Vererbungshierarchie festgelegt. Die Indizes verweisen in den *Constant Pool*.

- *Fields*
  Beschreibt alle globalen Variablen, die in dieser Klasse definiert wurden. Auch hier wird wieder in den Constant Pool referenziert. Von Oberklassen oder implementierten Interfaces geerbte Variablen werden hier nicht aufgelistet.

- *Methods*
  Dieser Abschnitt beschreibt die Methoden einer Klasse. Dazu gehören neben den deklarierten Methoden auch Konstruktoren und der statische Initialisierer. Analog zu den globalen Variablen werden nur deklarierte und keine geerbten Methoden und Konstruktoren verwaltet. Eine Methode wird durch eine spezielle Struktur beschrieben. Sie besteht aus einer Bitmaske, einem Index für den Methodennamen und der Signatur sowie einer Tabelle mit Attributen.

  Es fällt auf, dass bis zu diesem Punkt noch überhaupt kein Bytecode in der Class-Datei auftaucht. Dies soll sich mit den Attributen ändern. Zunächst wird aber die Methodensignatur näher erläutert. Aus Java-Programmierer- bzw. Java-Quellcode-Sicht besteht die Methodensignatur aus Methodenname, Rückgabewert und Methodenparametern. Dies gilt auch für den Java-Bytecode, doch die Darstellung ist eine andere. Tabelle 2.1 zeigt eine Übersetzungstabelle für die unterschiedlichen Datentypen.

  Anhand dieser Tabelle läßt sich leicht nachvollziehen wie im Bytecode eine Methode dargestellt wird. Beispielsweise wird die Methode

```
String createString(char[] characters, int offset){/*...*/}
```

Boolean	Z
Byte	B
Short	S
Char	C
Int	I
Long	J
Float	F
Double	D
Array	[<*Datentyp*>
Object	L<*Klassenname*>;
Void	V

Tabelle 2.1: Übersetzungtabelle für Java-Datentypen.

mittels der Zeichenkette

([CI)Ljava/lang/String;

im Java-Bytecode beschrieben. Dies ist zu lesen als eine Methode „(...)“, deren erster Parameter ein Array „[“ vom Typ char „C“ ist, deren zweiter Parameter den Typ int „I“ hat und als Ergebnistyp ein Object „L“ liefert. Die Signatur setzt sich also aus einer geklammerten Liste der Parameter gefolgt vom Rückgabewert zusammen. Diese Schreibweise wird auch vom Java Native Interface (JNI) genutzt, so dass Programme, die in einer anderen Programmiersprache geschrieben wurde, Methoden eines Java-Programms aufrufen können [Ste, HC99b].

- *Class attributes*
  Klassenattribute bilden den Abschluss einer Class-Datei. Mittels dieser Attribute kann eine Class-Datei um weitere Informationen angereichert bzw. erweitert werden[4]. Ein Reihe von Attributen sind jedoch verpflichtend. Dazu gehören die Attribute *Code* und *Exception*, die für jede Methode definiert werden müssen. Neben Metainformationen enthält das Code-Attribut die Bytecodeanweisungen (Opcodes) die vom Compiler erzeugt wurden. Das Exception-Attribut ist eng mit dem Code-Attribute verzahnt und kann als Äquivalent zum Try-Catch-Ausdruck verstanden werden, da es für bestimmte Codebereiche Sprungadressen bereitstellt. Tritt eine Exception auf, werden die Exception-Tabelle und der Wert des Programmzählers verglichen. Liegt der Programmzähler innerhalb eines durch Try-Catch abgedeckten Code-Blocks, wird dieser auf den Wert der Sprundadresse gesetzt. Dazu ein kurzes Beispiel:

[4]Durch Hinzufügen neuer Attribute ist es möglich die Programmiersprache Java zu erweitern, ohne die Binärkompatibilität zu verletzen. Davon wurde bei der Erweiterung des Java-Sprachumfangs (z.B. Annotation) Gebrauch gemacht.

Listing 2.1: Java-Programmcode

```
public void dangerous(long n) throws Exception{
        if(n % 2 == 0){
                throw new Exception("that's odd");
        }
}

public String playTheGame(){
        try{
                long now = System.currentTimeMillis();
                dangerous(now);
                return "lucky winner";
        }catch(Exception e){
                return "lucky looser";
        }
}
```

Das Listing 2.1 zeigt, dass eine eventuell auftretende Exception in der Methode *playTheGame* durch einen Try-Catch-Konstrukt aufgefangen wird. In Listing 2.2 ist der Bytecode dieser Methode zu sehen[5]. Man beachte, dass es keine Opcodes für das Try-Catch-Konstrukt gibt sondern nur die Exception-Tabelle.

Listing 2.2: Java-Bytecode

```
public java.lang.String playTheGame();

 Code:
 0:   invokestatic     #7; //System.currentTimeMillis:()J
 3:   lstore_1
 4:   aload_0
 5:   lload_1
 6:   invokevirtual    #8; //Method dangerous:(J)V
 9:   ldc              #9; //String lucky winner
11:   areturn
12:   astore_1
13:   ldc              #10; //String lucky looser
15:   areturn

 Exception table:
 from   to   target  type
   0    11     12    Class java/lang/Exception
```

[5]Bytecode-Listings dieser Art lassen sich mit den Programm javap, das zum JDK gehört, erzeugen.

2.5 Entwurfsmuster

Da diese Arbeit große praktische Anteile enthält, sollen auch Entwurfsmuster (*engl.: design patterns*) vorgestellt werden. Entwurfsmuster stammen ursprünglich aus der Architektur, wurden aber recht früh von der Softwareentwicklung adaptiert [Wik02]. Dabei handelt es sich um generische und bereits verwendete Lösungen für wiederkehrende Probleme. Sie helfen Probleme auf bewährte Art und Weise zu lösen und dienen durch ihre Standardisierung als Kommunikationsmittel. Vor allem in der objektorientierten Entwicklung gehört die Verwendung von Entwurfsmustern „*zum guten Ton*“.

Bei der Implementierung des alternativen Backends wurden einige Entwurfsmuster angewendet. Zudem werden im weiteren Text wiederholt Entwurfsmuster auftauchen. Die klassische Literatur ist hierzu das Buch „Design patterns“ [GHJV94]. Dort werden eine Reihe von unterschiedlichen Entwurfsmustern besprochen und ausführlich erklärt. Im Rahmen dieser Ausarbeitung werden nur ein paar konkrete Entwurfsmuster gezeigt. Zuvor soll aber eine Klassifizierung gegeben werden. Die meisten Entwurfsmuster lassen sich in eine der drei folgenden Klassen einteilen:

- *Erzeugermuster* - Das Entwurfsmuster steuert die Erzeugung eines neuen Objekts.

- *Strukturmuster* - Verdeutlichen die Beziehungen unterschiedlicher Objekte zueinander wie zum Beispiel eine Liste von Objekten.

- *Verhaltensmuster* - Beschreibt die Interaktion von Objekten untereinander. Dies beinhaltet die Kommunikation sowie die Steuerung.

Besucher *(engl. Visitor-Pattern)*

Das Besucher-Entwurfsmuster ermöglicht die funktionale Erweiterung einer Klasse ohne die Klasse selbst zu ändern. Dies setzt mindestens zwei Akteure voraus: Ein Besucher, der eine entsprechende Funktion implementiert, sowie die eigentliche Klasse auf der die Funktion arbeitet. Das Besucher-Entwurfsmuster findet Anwendung, wenn eine Trennung von Objektstruktur und Funktionalität gewünscht oder erfordert ist.

Ein Fallbeispiel soll verdeutlichen, wie das Besucher-Entwurfsmuster angewendet werden kann. Um die Nähe zum Thema dieser Arbeit beizubehalten, wird vorgestellt wie eine Bibliothek zur Bytecodemanipulation und Analyse aufgebaut werden kann. Ziel soll es sein, eine Objektrepräsentation einer Class-Datei zu haben, die die unterschiedlichsten Operationen wie Bytecodemanipulation, Code-Optimierungen oder Programmablauf-Analyse zulässt. Die Objektrepräsentation der Class-Datei soll in diesem Beispiel *JavaClass* heißen.

Ein erster Implementierungsansatz dieser Klasse würde neben der eigentlichen Objekt-Repräsentation wahrscheinlich auch die unterschiedlichen Operationen enthalten. Dieser Ansatz hat einige Kritikpunkte. Durch die Ansammlung

unterschiedlichster Operationen (Analyse, Manipulation, usw.) in nur einer Klasse wird das Programmier-Prinzip der Kohäsion verletzt. Also die Unterbringung logisch verschiedener Konzepte in einer Klasse. Durch die Einführung von Schnittstellen und die Trennung von Objektrepräsentation und Operationen behebt das Besucher-Entwurfsmuster dieses Problem.

Zentraler Bestandteil ist eine Schnittstelle, die üblicherweise Besucher (engl. Visitor) genannt wird und von jeder gewünschten Operation implementiert werden muss. Diese Schnittstelle bietet für Elemente der eigentlichen Objektrepräsentation Methoden an. Bspw. wäre dies eine Methode *besucheVariable(Name, Datentyp)*. Die Operationen bestimmen durch ihre Implementierungen, was in den Methoden geschehen soll. Im Falle der Manipulation könnte der Name verändert werden, ein Optimierer prüft evtl. ob die Variable überhaupt gelesen wird, oä. Damit die Operationen tatsächlich ausgeführt werden, gibt es in *JavaClass* eine Methode *akzeptiere(Besucher)*. Diese ruft nacheinander alle Methoden des übergebenen Besuchers auf und übergibt sich selbst als Parameter. Zum besseren Verständnis ist in Abbildung 2.8 ein UML-Diagramm des Beispiels gezeigt.

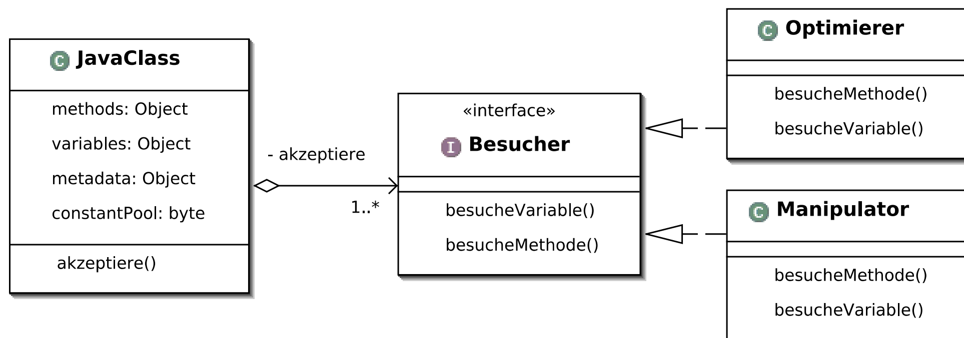


Abbildung 2.8: Das Besucher-Entwurfsmuster im UML-Diagramm. Die Klasse *JavaClass* enthält die Objektrepräsentation, wobei die Implementierungen des Besuchers unterschiedliche Funktionen darstellen.

Beobachter *(engl. Listener/Observer-Pattern)*

Ein Objekt möchte auf ein bestimmtes Ereignis reagieren, aber nicht aktiv auf dessen Auftreten warten. Wenn diese Problemstellung gegeben ist, kann das Beobachter-Entwurfsmuster verwendet werden. Auch dort gibt es zwei Akteure. Ein *Beobachter* wartet auf das Auftreten eines Ereignisses und ein *Erzeuger* generiert diese Ereignisse. Damit der Erzeuger die Beobachter über das Auftreten eines Ereignisses informieren kann, müssen sie eine spezielle Schnittstelle implementieren und sich beim Erzeuger registrieren.

Singleton *(engl. Singleton-Pattern)*

Das Entwurfsmuster „Singleton“ gehört zu den Erzeugermustern. Es wird genutzt, um sicherzustellen, dass es immer nur eine Instanz einer Klasse gibt. Um dies

zu erreichen werden die Konstruktoren als nicht-öffentlich deklariert. Stattdessen liefert eine spezielle Methode die Singleton-Instanz. Ein Code-Beispiel in Listing 2.3 soll dies verdeutlichen.

Listing 2.3: Instanzmethode für Singleton-Pattern

```
private static MyObject instance = null;

private MyObject(){
  //Konstruktor - Objekt erzeugen...
}

public static MyObject getInstance(){
  if(instance == null){
    instance = new MyObject();
  }
  return instance;
}
```

Kapitel 3

Entwurfsentscheidungen

In der Einleitung wurden die Ziele dieser Arbeit skizziert, welche nun konkretisiert werden. Im Mittelpunkt steht, die Forderungen nach einer nahtlosen Integration des alternativen Backends und die Verwendung von Bytecode-Manipulationen anstelle der JPDA zu erfüllen. Damit soll die klassische Vorgehensweise, die der Code-Manipulation, auch in Jassda möglich sein. Es handelt sich dabei aber keineswegs um einen Umbau, sondern um eine Erweiterung Jassdas. Die Nutzung des bisherigen und alternativen Backends soll parallel oder zumindest alternativ möglich sein.

In den folgenden Abschnitten sollen daher zunächst unterschiedliche Integrationsweisen diskutiert werden. Daran knüpft die Vorstellung und Bewertung unterschiedlicher Vorgehensweisen bei der Bytecode-Manipulation an. Dies umfasst den Zeitpunkt der Manipulation und die konkrete Vorgehensweise bzgl. bekannter Ereignisse. Abschließend werden unterschiedliche Bytecode-Toolkits vorgestellt. Ihr Ziel ist eine vereinfachte Manipulation des Bytecodes. Neben der nahtlosen Integration soll das alternative Backend hohe Geschwindigkeitsforderungen erfüllen. Die Verwendung der JPDA bietet bzgl. der Ausführungsgeschwindigkeit Anlass zur Kritik. Die Entwicklung des alternativen Backends soll dies beachten.

3.1 Integration des Backends

Es sollen nun drei unterschiedliche Integrationsweisen des alternativen Backends in Jassda vorgestellt werden. Dabei handelt es sich in zwei Fällen um Ansätze, die eine in Jassda definierte Schnittstelle nutzen. Trotz der offensichtlichen Vorteile dieser Ansätze wird zunächst eine direkte Integration des Backends in Jassda vorgeschlagen.

3.1.1 Direkte Integration

Die direkte Integration des alternativen Backends kann auf unterschiedlichen Wegen erreicht werden. Ihre Implementierungen gleichen sich darin, dass keine Programmschnittstellen, die als solche gedacht waren, genutzt werden, sondern Komponenten Jassdas direkt erweitert werden. Dies ist nur möglich, weil Jassda im

Quelltext mit Entwurfsdokumenten vorliegt. Der Ansatz der direkten Integration hat entscheidene Nachteile.

Wird Jassda erweitert ohne auf eine definierte Schnittstelle zurückzugreifen, ist das sehr genaue Verständnis aller Programmkomponenten und ihrer Zusammenhänge unabdinglich. Eine solche Integration würde viele Komponenten Jassdas als Schnittstelle brauchen und damit eine elegante Integration erschweren. Dies gilt insbesondere, wenn zwischen den unterschiedlichen Backends gewechselt werden soll, da das alternative Backend direkt in Jassda „hinein-programmiert“ wäre. Zudem ist aus Abschnitt 2.3 bekannt, dass ein Backend über einen Konnektor mit Jassda verbunden wird. Es wäre also nicht sinnvoll, wenn ein alternatives Backend diese Flexibilität aufgibt.

3.1.2 Implementierung des JDIs

Das *Java Debug Interface* ist die oberste Schicht in der JPDA und stellt eine Schnittstelle zu den Debuggees dar. Bei dem JDI handelt es sich lediglich um eine Schnittstellenspezifikation. Die Java-Umgebung wird mit einer Standardimplementierung des JDIs geliefert, welche von Jassda genutzt wird. Die Grundidee dieser Integrationweise ist die Schaffung einer alternativen Implementierung des JDIs.

Abschnitt 2.2 erläutert das Prinzip des Konnektors. Stellt ein Konnektor eine Verbindung zur Virtual Machine des Debuggees her, liefert dieser ein Objekt vom Typ *VirtualMachine*. Dabei handelt es sich um ein Spiegelbild der Debuggee-VM. Es erlaubt die VM zu inspizieren und zu steuern. Das Interface *VirtualMachine* ist im JDI spezifiziert und dadurch prädestiniert zentrales Objekt einer eigenen JDI-Implementierung zu werden. Diese kann inkrementell vom Objekt *VirtualMachine* begonnen werden.

Da Jassda das JDI als Schnittstelle zum Debugger nutzt, lässt sich so auch ein alternatives Backend anbinden. Die elegante Integration wäre also sichergestellt. Es bleibt aber ungeklärt, wie die eigene JDI-Implementierung mit dem Debuggee kommuniziert. Aus Abschnitt 2.2 wissen wir, dass in der Java Platform Debugging Architecture das JDWP (Java Debug Wire Protocol) die Kommunikation zwischen JDI-Implementierung (dem Debugger) und Debuggee definiert. Da das alternative Backend die Funktionalität nicht einschränken darf, muss dieses einen ähnlichen Kommunikationsmechanismus bieten. Es muss also ein Protokoll spezifiziert werden, das den Debuggee mit der eigenen JDI-Implementierung verbindet. Dies führt zu einer gewissen Problematik.

Da das JDWP und JDI mit einer Java Virtual Machine verbunden werden sollen, wurde dessen Architektur beim Entwurf berücksichtigt. Dies bedeutet, dass die Befehle des JDWPs im JDI und in der JVM ihre Entsprechung finden. So stimmen fast alle Methoden der Klasse *com.sun.jdi.VirtualMachine* mit den Befehlen aus der Befehlsgruppe *VirtualMachine* überein. Auch das Java Virtual Machine Tool Interface (kurz: JVMTI) bietet eine ähnliche Funktion. Entwirft man ein proprietäres Protokoll, ist eine Wiederholung der JDWP-Befehle durch die Definition des JDIs quasi vorgegeben. Zumindest die Befehle des JDWP, die durch Jassda genutzt werden, müssten in gleicher Weise oder sehr ähnlich „neu“ entworfen wer-

den.

3.1.3 Implementierung des JDWPs

Der vorangehende Abschnitt diskutiert die JDI-Implementierung und stellt die Redundanz heraus, die sich durch die Schaffung eines proprietären Protokolls ergeben würde. Es liegt also nahe, anstatt das JDI das JDWP zu implementieren. Wenn eine JVM im Debug-Modus ausgeführt wird, ist sie mittels JDWP steuerbar. Dieses Verhalten soll dem alternativen Backend Vorbild sein. Es würde sich aus der Sicht Jassdas wie eine gewöhnliche JVM verhalten, indem es über das JDWP kommuniziert und den Debuggee überwacht. Integrationsprobleme tauchen bei diesem Ansatz nicht auf. Es müssen lediglich alle JDWP-Befehle implementiert werden, die von Jassda benötigt werden.

Nachteile bei Implementierung des JDWPs ergeben sich dadurch, dass das JDWP und JDI die Architektur und Arbeitsweise der JVM berücksichtigen müssen. Aus Abschnitt 2.1 wissen wir, dass die JVM einen stackorientierten Rechner simuliert. Daraus ergeben sich unterschiedliche Repräsentationen eines Programms zu Lauf-/Debug-Zeit und zur Zeit des Programmierens. Während der Programmierer ein Programm schreibt hat er bspw. vollen Zugriff auf lokale Variablen. Zur Laufzeit ist dies anders, da nur über den Stack auf Variablenwerte zugegriffen werden kann. Die JPDA dient als Vermittler zwischen diesen beiden Repräsentationen und versucht dadurch das Debugging einfach zu gestalten. Nichtsdestotrotz sind die Befehle des JDWPs auf eine JVM zugeschnitten und das alternatives Backend müsste sich diesen Umständen anpassen und ggf. das Vorhandensein eines Stacks sowie weiterer Komponenten der JVM emulieren.

Bewertung der Integrationsvorschläge

Die direkte Integration in Jassda ist nicht zu empfehlen. Ein nahtlose Integration ist hierbei ausgeschlossen und würde zu einer wenig eleganten Lösung führen. Die Implementierung des *Java Debug Interface*s entspricht den Grundsätzen einer guten Integration. Eine bereits verwendete Schnittstelle, das JDI, wird neu implementiert und stellt so das gewünschte Verhalten bereit. Es wird aber nicht beschrieben, wie die Kommunikation zwischen Debuggee und dem JDI ablaufen soll. Die entstehende Redundanz eines proprietären Protokolls kann durch die Implementierung des *Java Debug Wire Protocol*s vermieden werden. Dabei entsteht ein Backend das direkt mit dem Debugger verbunden ist und JDWP-Befehle interpretieren kann. Trotz der teilweisen Emulation einer Virtual Machine bietet dieses Vorgehen die meisten Vorzüge. Diese sind ein geringerer Implementierungsaufwand, die Nutzung einer fest definierten Schnittstelle und somit eine nahtlose Integration in Jassda. Zur Verdeutlichung stellt Abbildung 3.1 die unterschiedlichen Integrationsweisen gegenüber.

Begriffsklärung: Backend (zweiter Teil). Es soll noch einmal der Begriff des Backends behandelt werden. Bereits das Ende von Abschnitt 2.3 konkretisiert

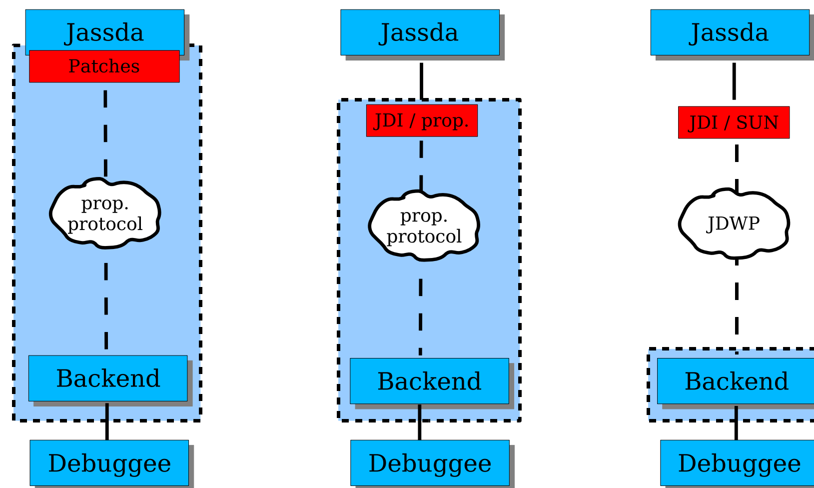


Abbildung 3.1: Drei mögliche Integrationsweisen des alternativen Backends. Von links sind dies: *Direkte Integration, Implementierung JDI*, *Implementierung JDWP*. Zu implementierende Komponenten sind durch ein Rechteck umschlossen.

diesen Begriff und verdeutlicht die Unterschiede zwischen dem Backend aus der Debugging-Architektur und der Idee des Backends in Jassda. Die Bewertung der Integrationsweisen hat ergeben, dass das alternative Backend für Jassda auch ein Backend in der Java Debugging-Architektur ist. Dadurch werden bereits vorhandene Strukturen genutzt und eine optimale Integration erreicht. Die bereits bekannte Abbildung kann vervollständigt werden, so dass sich die in Abbildung 3.2 gezeigte Gesamtarchitektur ergibt.

3.2 Vorgehensweisen zur Ereigniserzeugung

In diesem Abschnitt soll diskutiert werden, wie sich die gewünschte Ereignisse durch die Manipulation des Bytecodes erzeugen lässt. Dazu werden rückblickend die Ereignisse aufgelistet, an denen Jassda potentielles Interesse besitzt.

- Methodenanfang: Die zu untersuchende Methode wurde betreten/gestartet.
- Methodenende: Die zu untersuchende Methode wurde normal verlassen/beendet.
- Exception: Die zu untersuchende Methode wurde aufgrund einer Ausnahme beendet.

Bevor diskutiert wird, wie sich diese Ereignisse durch Änderungen des Bytecodes erzeugen lassen, soll geklärt werden, zu welchem Zeitpunkt die Code-Manipulation durchgeführt werden kann. Es muss zwischen der statischen und dynamischen Bytecode-Manipulation entschieden werden. Erstere liest den Bytecode ein, manipuliert diesen und speichert das Ereignis in eine Class-Datei. Die dynamische

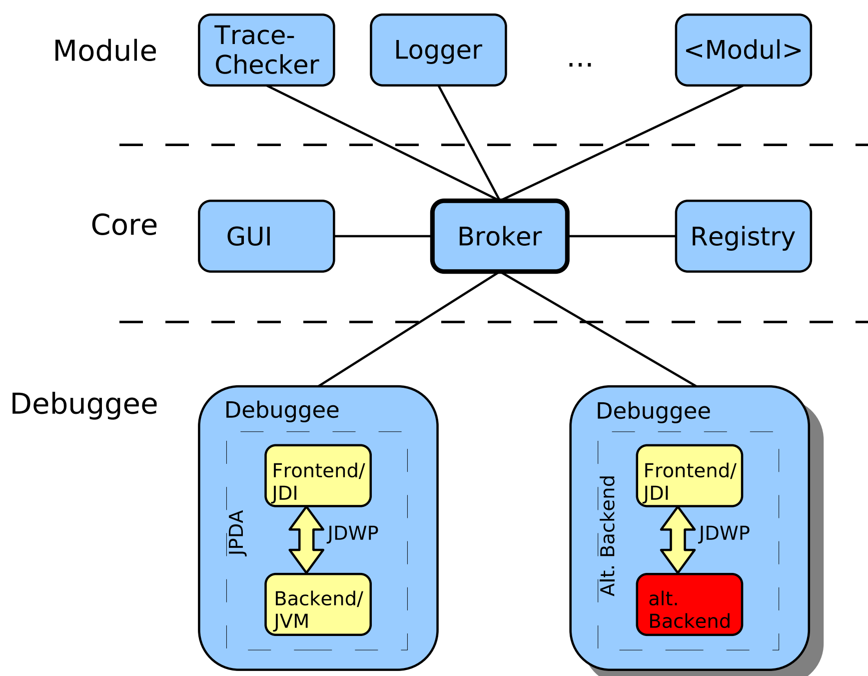


Abbildung 3.2: Die Integration des alternativen Backends ist eine Ersetzung des Backends aus der JPDA und damit sowohl Backend in der Java Debugging-Architektur als auch für Jassda.

Manipulation nutzt einen modifizierten Klassenlader (siehe Abschnitt 2.1), der Code-Manipulationen ausführt während die Klasse geladen wird. Auch wenn sich das Laden der Klassen dadurch etwas verzögert, überwiegt der Gewinn an Flexibilität und die Nähe zur Arbeitsweise Jassdas. Darüber hinaus können die Vorteile der hierarchisch angeordnenten Klassenlader ausgenutzt werden. Der Bytecode-Manipulation kann bspw. ein Klassenlader vorangestellt werden, der Bytecodes erst über eine Netzwerkverbindung lädt oder aus einer Datei entschlüsselt. Dies ist bei der statischen Code-Manipulation nicht möglicht und führt neben der höheren Flexibilität zur Entscheidung die dynamische Code-Manipulation für dieses Projekt vorzuziehen.

Es soll nun im Einzelnen geklärt werden, wie konkrete Ereignisse erzeugt werden können.

Methodenanfang

Um den Anfang einer Methode zu erkennen, gibt es zwei unterschiedliche, recht simple Verfahren. Einmal wird dem Code einer Methode eine weitere Anweisung hinzugefügt. Diese meldet den Anfang der Methodenausführung und muss selbstverständlich die erste Anweisung im Code der Methode sein.

Eine andere Vorgehensweise benutzt eine Wrapper-Methode. Bei diesem Vorgehen wird die zu untersuchende Methode umbenannt und eine neue Methode mit dem bisherigen Namen und der bisherigen Signatur erzeugt. Die Aufgabe der

Wrapper-Methode ist es, das Betreten der eigentlich zu untersuchenden Methode zu melden und diese dann aufzurufen. Das wiederholte Auslösen eines Ereignisses durch Rekursion der Methode kann vermieden werden, indem die Umbennung der Methode auch im Programmcode durchgeführt wird. Dies ist nützlich, wenn die gesamte Ausführungszeit einer Methode gemessen werden soll.

Beide Verfahren sind recht ähnlich und es soll nun eines für das alternative Backend bestimmt werden. In Abschnitt 2.1 haben wir gesehen, dass für jeden Methodenaufruf ein neuer Stack-Frame angelegt wird. Die Nutzung einer Wrapper-Methode verbraucht also mehr Speicher und erhöht den Verwaltungsaufwand für die JVM. Für sie spricht die bessere Lesbarkeit und die Möglichkeit rekursives Melden von Ereignissen zu ignorieren. Da letzteres nicht gewollt ist und auch die Lesbarkeit des Bytecodes eine untergeordnete Rolle spielt, wird das direkte Einfügen einer Anweisung in die Methode bevorzugt.

Methodenende

Das Methodenende kann analog zur Diskussion über den Methodenanfang betrachtet werden. Auch hier wird das direkte Einfügen einer Meldeanweisung gewählt.

Exception

Exceptions lassen sich in zwei Klassen unterteilen. Solche, die durch einen Catch-Block abgefangen werden, und Exceptions, die nicht abgefangen. Letztere zu erkennen, ist recht einfach, da das entsprechende Programm in diesem Fall abbricht. Auf diese Klasse von Exceptions wird nicht weiter eingegangen. Wie wird aber mit Exceptions verfahren, die zur Laufzeit abgefangen werden? Es bieten sich mehrere Ansätze an, aber nicht alle sind zufriedenstellend.

Es drängt sich auf, alle Class-Dateien nach der Anweisungen zu durchsuchen, die Exceptions auslösen. Der *ATHROW*-Opcode macht genau dies. Es erwartet ein Exception-Objekt als oberstes Element auf dem Stack und wirft es. (Der genaue Mechanismus wird in den Abschnitten 2.1 und 2.4 beschrieben.) Ist der *ATRHOW*-Opcode gefunden, kann ihm eine zusätzliche Anweisung vorangestellt werden, die das Auftreten der Exception, noch vor dem eigentlichen Werfen, melden. Leider können auf diese Weise nicht alle Exceptions erkannt werden. Die Spezifikation der Java Virtual Machine sieht vor, dass Exceptions auch direkt in der JVM erzeugt werden können - also nirgendwo im Bytecode definiert sind. Solche Exceptions treten bspw. auf, wenn der Virtual Machine nicht ausreichend Speicher zur Verfügung steht.

Eine alternative Vorgehensweise besteht darin, die Klasse *java.lang.Throwable* zu manipulieren. Wenn eine Exception erzeugt wird, handelt es sich immer um eine Instanz dieser Klasse. Wird der Konstruktor dieser Klasse manipuliert, kann jede neue Instanz dieser Klasse festgestellt werden. Leider ist dieses Vorgehen nicht uneingeschränkt möglich, da das Werfen einer Exception nicht voraussetzt, dass eine neue Instanz erstellt wird. Es ist also möglich eine Exception-Instanz mehrfach zu

benutzen. Dies lässt sich evtl. noch durch einen konsequenten Programmierstil verhindern. Weitaus problematischer ist die Tatsache, dass durch die Manipulation der Klasse *java.lang.Throwable* die Sun Microsystems Binary Code License [Sun04a] verletzt wird. Anhang C (Java Technology Restrictions) verbietet ausdrücklich jede Manipulation der Klassen aus den Standard-Bibliotheken und schließt damit diese Vorgehensweise aus.

Bisher konnte keine Vorgehensweise gefunden werden, die allen Ansprüchen genügt. Es soll nun eine weitere Alternative, die sich auf die Untersuchung von Catch-Blöcken beschränkt, vorgestellt werden. Ein Catch-Block wird immer dann betreten, wenn eine Exception auftritt und diese von einem Try-Catch-Konstrukt umschlossen ist. Zu jeder abgefangenen Exception gibt es also einen Catch-Block, der genau dieses Abfangen übernimmt. Die Idee besteht darin, in jeden Catch-Block zusätzliche Anweisungen einzufügen. Diese analysieren die Exception und senden ein Exception-Ereignis an Jassda. Im Gegensatz zum JVM-Backend kann durch die Bytecodemanipulation nicht jeder Catch-Block untersucht werden. Dies ist durch die Restriktionen der Sun Binary License [Sun04a] bedingt. Diese Einschränkung gilt jedoch allgemein für das alternative Backend. Die Konsequenzen, die sich daraus ergeben, werden in 4.3 diskutiert. Auch wenn nicht alle Klassen auf Exceptions untersucht werden können, kann jede Exception, unabhängig von ihrem Entstehungsort, erkannt werden. Voraussetzung hierfür ist, dass sie im Debuggee abgefangen wird.

3.3 Wahl eines Bytecode-Toolkits

Abschnitt 2.4 beschreibt den Aufbau von Class-Dateien und stellt komplexe strukturelle Zusammenhänge im Bytecode heraus. Bytecode-Toolkits vereinfachen die Arbeit mit Class-Dateien. Sie bieten Programmschnittstellen zur Analyse und Manipulation. Ein solches Toolkit soll auch vom alternativen Backend genutzt werden. Daher werden an dieser Stelle drei unterschiedliche freie Bibliotheken zur Bytecodemanipulation vorgestellt. Sie wurden ausgewählt, weil sie Vertreter unterschiedlicher Ansätze sind. Auf der Internetseite [Jav] kann eine Auflistung weiterer Bytecode-Toolkits gefunden werden. Zum Abschluss dieses Abschnitts wird ein Bytecode-Toolkit benannt, das für das alternative Backend am besten geeignet ist.

3.3.1 BCEL

Die Bytecode Engineering Library (kurz: BCEL) ist wohl die bekannteste und verbreiteste Bibliothek um Java Bytecode zu manipulieren. BCEL wird seit der Java-Version 1.1 gepflegt und gehört mittlerweile zur Apache Software Foundation [Dah01]. BCEL arbeitet in drei Phasen, wobei in der ersten Phase eine Class-Datei geparst und als Objekt repräsentiert wird. Dieses Objekt kann in der zweiten Phase manipuliert werden um danach, in der dritten Phase, wieder als Class-Datei gespei-

chert zu werden. Da die Manipulation im Speicher an der Objektrepräsentation und nicht am Bytecode direkt durchgeführt wird, ergeben sich einige Komfort-Methoden. Bspw. kann dafür Sorge getragen werden, dass bei Umbenennung einer Methode auch der Constant-Pool referenzierender Klassen aktualisiert wird.

BCEL wurde bereits in vielen namhaften Projekten eingesetzt. Trotz ihrer weiten Verbreitung ist seit April 2003 keine Entwickleraktivität im BCEL-Projekt zu beobachten. Nötige Erweiterungen, die durch Änderungen im Zuge der neuen Java Version 1.5 entstanden sind, wurden nicht vorgenommen. Es bleibt abzuwarten, wie sich die BCEL entwickelt.

3.3.2 Javassist

Die relativ junge Bibliothek Javassist (*Java* Programming *Assist*ent) wurde am Tokio Institute of Technology entwickelt und vereinfacht durch eine zusätzliche Abstraktionsebene die Bytecodemanipulation [Chi98, CN03]. Neben der direkten Manipulation des Bytecodes gibt es die Möglichkeit, Ergänzungen und Änderungen als Java-Quellcode anzugeben. Diese indirekte Manipulation des Bytecodes stellt einen wesentlich einfacheren Einstieg in die Bytecodemanipulation dar. Zudem entsteht Code der im Allgeimeinen leichter zu warten und schlichtweg kürzer ist. Es ist jedoch nicht zu empfehlen, Bytecodemanipulation ohne Kenntnis der Class-Dateien oder der Opcodes durchzuführen. Zudem entsteht durch den Komfort der indirekten Bytecodemanipulation ein gewisses Risiko ungültigen Bytecode zu erzeugen. Beispielsweise gewährleistet der Javassist-Compiler nicht die Typsicherheit. Es ist also durchaus möglich Bytecode zu erzeugen, der einen String in eine Integervariable speichert. Solche Fehler werden erst zur Laufzeit bemerkbar und sind im Allgemeinen schwerwiegend, so dass die Ausführung des Programms abgebrochen werden muss.

Dennoch soll in den folgenden Code-Listings gezeigt werden, wie mit Javassist im Vergleich zu BCEL eine Klasse manipuliert wird. Bei der Änderung handelt es sich lediglich um das Einfügen einer Textausgabe zu Beginn einer Methode.

Listing 3.1: Javassist

```
ClassPool classPool = ClassPool.getDefault();
CtClass clazz = pool.get("example.Geom.Point");
CtMethod method = clazz.getDeclaredMethod("move");
method.insertBefore("System.out.println(\"[moving point]\");");
clazz.writeFile();
```

Listing 3.2: BCEL

```
ClassGen classGen = new ClassGen(
   new ClassParser("example.Geom.Point").parse());
ConstantPoolGen constPool = classGen.getConstantPool();

//method 'move' is first method in the class
Method method = classGen.getMethodAt(1);
MethodGen methodGen = new MethodGen(method,
```

```
    "example.Geom.Point", constPool);

InstructionFactory ifactory = new InstructionFactory(classGen,
    constPool);
InstructionList il = new InstructionList();

il.append(ifactory.createFieldAccess("java.lang.System", "out",
    new ObjectType("java.io.PrintStream"), Constants.GETSTATIC));
il.append(new PUSH(constPool, "[moving point]"));
il.append(ifactory.createInvoke("java.io.PrintStream",
    "println", Type.VOID, new Type[] { Type.STRING },
    Constants.INVOKEVIRTUAL));

//append the old instruction list to the new on
il.append(methodGen.getInstructionList());

methodGen.setInstructionList(il);
classGen.getJavaClass().dump("example/Geom/Point.class");
```

Ohne genauer auf den Code einzugehen, sollte aus den Listings 3.1 und 3.2 deutlich werden, dass Javassist die elegantere Methode zur Bytecodemanipulation bietet. Zudem ist durch die Aufnahme von Javassist in das Opensource Projekt JBoss und der Verwendung im Projekt AspectJ[1] eine Weiterentwicklung und Verbesserung zu erwarten.

3.3.3 ASM

ASM[2] wurde mit Hinblick auf Kompaktheit und Geschwindigkeit entwickelt. Um dieses Ziel zu erreichen, wurde auf eine Objektrepräsentation des Bytecodes verzichtet [BLC02]. Eine Objektrepräsentation setzt voraus, dass jede Bytecodeanweisung als Instanz einer entsprechenden Klasse existiert. Dies führt zu speicherintensiven Anwendungen und nicht zuletzt auch zu großen Bibliotheken, da für jeden Opcode eine entsprechende Klasse vorhanden sein muss.

ASM nutzt das Besucher-Entwurfsmuster (siehe Abschnitt 2.5), um dem Programmierer Zugriff auf die einzelnen Komponenten einer Class-Datei zu ermöglichen. Die ASM Bibliothek enthält das Interface *ClassVistor*, welches für alle Elemente einer Class-Datei eine entsprechende Funktion definiert. Die Klasse *ClassReader* liest eine Class-Datei ein und ruft dabei die Methoden eines *ClassVisitor*s entsprechend der Elemente der Class-Datei auf. Wichtig hierbei ist, dass die Inhalte der Class-Datei nicht im Speicher gehalten werden, sondern direkt an den *ClassVisitor* übergeben werden. Die jeweilige Implementierung des *ClassVisitor*s legt fest, was mit den Inhalten der Class-Datei geschehen soll.
Der Abschnitt 3.3.2 zeigt die Einfachheit auf, die Javassist durch die „indirekte“

[1] AspectJ ist eine aspekt-orientierte Programmiersprache basierend auf Java.

[2] Der Name ASM wurde vom Schlüsselwort __asm__ der Programmiersprache c abgeleitet. Dieses leitet einen Abschnitt mit Assemblercode ein.

Bytecodemanipulation bietet. Mit dem Tool *ASMifier* bietet die ASM Bibliothek einen anderen Mechanismus, der die Einarbeitungzeit verkürzen und das Erlernen des Bytecodes erleichtern kann. Ein typischer Arbeitsablauf mit dem *ASMifier* sieht so aus:

1. Eine gewünschte Funktion wird wie gewohnt in Java programmiert und zu einer Class-Datei compiliert.

2. Der ASMifier wird mit der Class-Datei als Argument aufgerufen und erzeugt Java-Quellcode.

3. Bei dem erzeugten Quellcode handelt es sich um Funktionsaufrufe aus der ASM Bibliothek. Genau diese Funktionsaufrufe erzeugen Bytecode, der äquivalent zur anfangs geschriebenen Funktion ist.

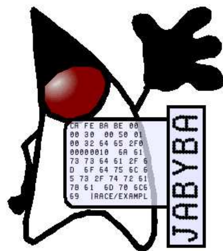
Ein ähnliches Tool existiert auch für die BCEL und heißt *BCELifier*. Beide Werkzeuge haben den Vorteil, dass der Komfort, den sie bieten, die Laufzeit nicht beeinflusst.

Zusammenfassung

Abschließend bleibt die Auswahl einer der Bibliotheken. Für BCEL spricht nicht nur die weite Verbreitung und Ausgereiftheit sondern auch die Tatsache, dass sie bereits in Jassda verwendet wird. Nachteilig ist jedoch die schlechte Performanz und der anscheinende Entwicklungsstopp. Javassist ist eine viel versprechende Bibliothek, die vor allem eine relativ einfache Entwicklung verspricht. Leider ist die indirekte Bytecodemanipulation nicht so mächtig wie gewünscht und bringt erhebliche Laufzeiteinbußen mit sich. Für diese Arbeit scheint ASM die am besten geeignete Bibliothek zu sein. Sie besticht durch hohe Geschwindigkeit und gute Unterstützung für den Entwickler. Teile in Jassda, die die BCEL nutzen, werden geändert, so dass nur eine Bibliothek zur Bytecode-Manipulation, nämlich ASM, verwendet wird.

Kapitel 4

Das alternative Backend



Im vorherigen Kapitel wurden Designentscheidungen diskutiert und Konzepte für das alternative Backend erarbeitet. Dies umfasste die Wahl der Integrationsweise in Jassda, eine Betrachtung unterschiedlicher Bibliotheken und Strategien zur Bytecode-Manipulation. In diesen Kapitel wird das alternative Backend, das den Arbeitstitel *JaByBa* trägt, vorgestellt. Es werden die Architektur, die sich aus der Analyse der Anforderungen ergeben hat, sowie die wichtigsten Programmkomponenten vorgestellt. Ausführliche Informationen enthalten die Entwurfsdokumente und der kommentierte Quelltext. Bei der Kommentierung und Formatierung des Quelltextes wurden die *JavaDoc*-Richtlinien und die Sun Code Conventions umgesetzt [Sun00, KND$^+$99].

Im Abschnitt 3.1.3 wird festgestellt, dass die Implementierung des *Java Debug Wire Protocol* (JDWP) für dieses Projekt die beste Variante ist. Dadurch ergibt sich, dass das alternative Backend für Jassda wie eine Virtual Machine erscheint. Es ist per JDWP steuerbar und kontrolliert die Ausführung des Debuggees. Abbildung 4.1 stellt das Debugging mit dem alternativen Backend dem bisherigen gegenüber.

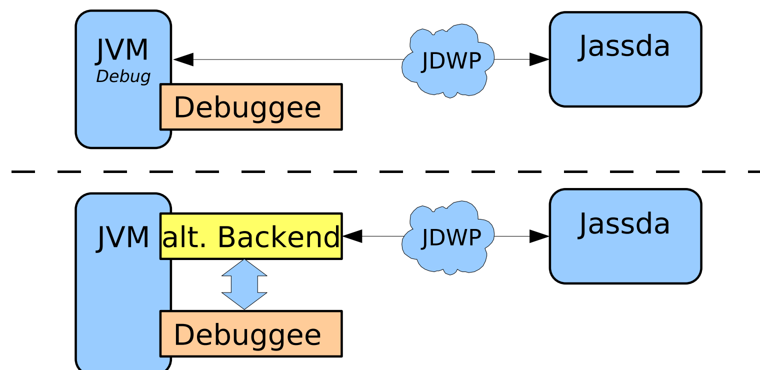


Abbildung 4.1: Debugging mit dem bisherigen und dem alternativen Backend.

In der Abbildung ist zu erkennen, dass bei Nutzung des bisherigen Backends Jassda direkt mit der JVM des Debuggees kommuniziert. Wie ein gewöhnliches Java-Programm, wird der Debuggee von ihr ausgeführt. Wird das alternative Backend genutzt, kommuniziert dieses und nicht die Virtual Machine mit Jassda. Der Debuggee und auch das Backend werden als ein Prozess von der JVM ausgeführt. Dabei besitzt das alternative Backend Einfluss auf den Debuggee. Die Tätigkeiten des Backend hinsichtlicht des Debuggees lassen sich in drei Bereiche gliedern.

1. Starten und Beenden des Debuggees.
2. Status-Informationen über den Debuggee.
3. Manipulation des Bytecodes.

Bevor die Architektur gezeigt wird, sollen Richtlinien für gute Software vorgestellt werden. Anhand dieser, der Entwurfentscheidungen und den Aufgaben 1-3 wurde das alternative Backend entwickelt.

Kriterien für gute Software

Das Entwerfen einer Softwarearchitektur ist eine kritische Aufgabe. Entscheidungen in dieser Phase haben die stärksten Auswirkungen auf das Endprodukt. Daher gibt es zahlreiche Richtlinien, die den Entwickler anleiten sollen. Einige Kriterien sind immer zu beachten. Programmierprinzipien der Kohäsion und der Koppelung sind da zu nennen. Sie beschreiben die Eigenschaft, dass logisch zusammenhängende Funktionen in entsprechenden Programmteilen stehen, und wie unterschiedliche Programmteile miteinander verwoben sind. Es wird immer eine hohe Kohäsion und geringe Koppelung angestrebt. Dies führt zur Modellierung von funktionalen Einheiten als Komponenten. Eine Komponente kapselt immer ein Konzept wie bspw. das der Datenhaltung oder der Kommunikation. Komponenten arbeiten nach dem Blackbox-Prinzip. Das heißt, um sie nutzen zu können, muss über ihre eigentliche Arbeitsweise nichts bekannt sein, aber eine Schnittstelle definiert sein, an der sie ihre Funktionen bereitstellt. Komponenten mit gleichen Schnittstellen können somit ausgetauscht werden, was die Wiederverwendbarkeit und Erweiterbarkeit von Software sichert. Diese Richtlinien und ihre konsequente Anwendung führen i.d.R. zu guter Software.

4.1 Architektur

Unter Beachtung der Richtlinien für gute Software hat die Umsetzung der Entwurfsentscheidungen aus Kapitel 3 zu fünf Basis-Komponenten geführt. Sie bilden die Architektur des alternativen Backends, welche in Abbildung 4.2 zu sehen ist.

Die nachfolgende Auflistung beschreibt die Komponenten im Einzelnen. Die Beschreibungen befinden sich trotz der Nähe zur Implementierung auf relativ hohem Abstraktionsniveau. Detaillierte Informationen können dem ausführlich dokumentieren Programmcode direkt entnommen werden.

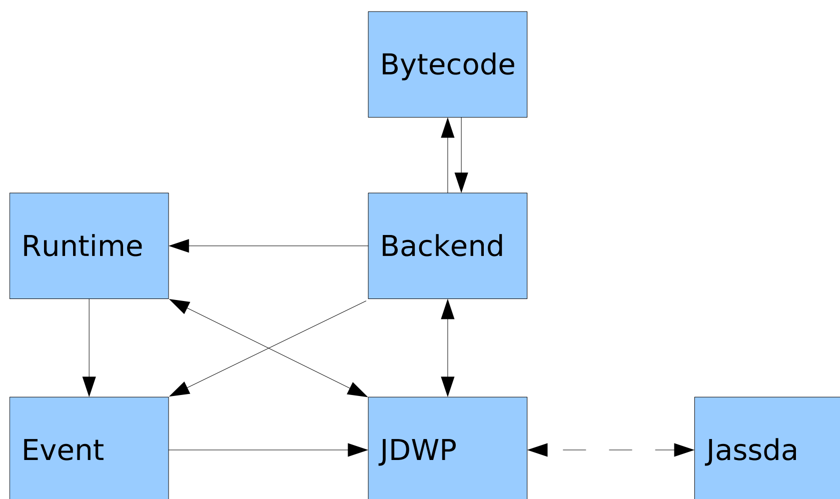


Abbildung 4.2: Die Architektur des alternativen Backends.

- **JDWP** - *de.jassda.jabyba.jdwp*
Die Kommunikation zwischen Backend und Debugger erfolgt über das Java Debug Wire Protocol (JDWP). Der Paketaustausch im JDWP findet über unterschiedlich lange Bytefolgen statt (siehe Abschnitt 2.2). Gilt die Verwendung von Bytefolgen auf Protokollebene als typisch, wird in einem objektorientierten Programm ein Paket als Instanz einer Klasse modelliert. Dies sowie Sende- und Empfangsmechanismen sind in der Komponente *JDWP* gekapselt.

  Wir wissen, dass Pakete des JDWPs in unterschiedliche logische Gruppen gegliedert sind. In allen Befehlsgruppen lassen sich ca. 100 unterschiedliche Befehle zählen. Es gilt diese Befehle auf effiziente und elegante Weise aufzubereiten und an die anderen Komponenten zu delegieren. Dazu stellt die Komponente *JDWP* einen Dispatcher und eine Reihe von Beobachtern (siehe Beobachter-Entwurfsmuster in 2.5) bereit. Für jede Befehlsgruppe im JDWP gibt es einen Beobachter, der für jeden in dieser Gruppe enthaltenen Befehl einen Methodenrumpf definiert. Durch die Implementierung der Beobachter in den übrigen Komponenten und dem Dispatcher, der eingehende Befehle Methoden der Beobachter zuordnet, ist das Empfangen und Delegieren von JDWP-Befehlen an die entsprechenden Komponenten garantiert. Zudem ergibt sich durch die Beobachter eine klare Schnittstelle, was die Komponente auch ausserhalb dieses Projekts einsetztbar macht.

- **Event** - *de.jassda.jabyba.backend.event*
Jassda beschreibt Programmabläufe anhand von Ereignissen. Diese müssen vom Backend erzeugt werden. Dafür gibt es die Komponente *Event*, welche das Konzept der Wächter (siehe Abschnitt „Communication“ in 2.2) umsetzt. Dazu sind zwei Schritte notwendig. Zuerst muss der Wächter installiert und dann, zur Laufzeit des Debuggees, ausgelöst werden. Ersteres übernimmt das

Backend, indem es Klassen zur Verfügung stellt, die die unterschiedlichen Wächtertypen modellieren. Jeder Anforderung nach einem neuen Wächter wird mit einer neuen Instanz dieser Klassen entsprochen. Der zweite Schritt wird vom Debuggee durchgeführt und benötigt die Bytecode-Manipulation. Das Auslösen eines Wächters entspricht einem Methodenaufruf in oben genannten Klassen. Dieser Methodenaufruf muss an entsprechender Stelle in den Code des Debuggee eingefügt werden.

- **Runtime** - *de.jassda.jabyba.backend.runtime*
  Aus Sicht der JVM sind das Backend und der Debuggee ein Programm, welches aus mehreren Threads besteht. Der Debuggee-Thread wird vom Backend gestartet und überwacht. Dazu ist es nötig einige Komponenten der JVM zu emuliert. Um zu erfahren, welche Klassen der Debuggee bereits geladen hat, ist ein Äquivalent zum Heap-Speicher nötig. Emulierte Stack-Frames ermöglichen den Zugriff auf Rückgabewerte von Methoden und ein Modell der Nebenläufigkeit soll es ermöglichen die Ausführung des Debuggees zu unterbrechen. Dies sind alles Eigenschafen, die von der JVM (siehe 2.1) bekannt sind und im alternativen Backend von der Komponenete *Runtime* modelliert werden.

  Besonderes Augenmerk soll auf die Nebenläufigkeit von Backend und Debuggee gelegt werden. Obwohl beide eigenständig ablaufen, kommt es beim Erreichen einer manipulierten Codestelle zur Interaktion. Der Debuggee ruft dann eine Methode des Backends, eine Wächter-Methode, auf und übergibt somit die Kontrolle über den Programmablauf an das alternative Backend. Damit kann die Fähigkeit der JVM, die Ausführung des Debuggee zu unterbrechen, zumindest teilweise nachgebildet werden. Gibt der Debuggee die Kontrolle an das alternative Backend ab, kann dieses entscheiden, wann die Ausführung des Debuggees fortgesetzt werden soll.

- **Bytecode** - *de.jassda.jabyba.backend.bytecode*
  Es wurde bereits herausgestellt, dass der Debuggee Methoden des alternativen Backends aufrufen muss. Dadurch können zur Laufzeit entsprechende Ereignisse erzeugt werden. Die Bytecode-Manipulation wird von der Komponente *Bytecode* übernommen.

  Im Mittelpunkt dieser Komponente steht ein modifizierter Klassenlader (siehe 2.1). Dieser prüft vor dem Laden einer Klassen, ob eine Manipulation gewünscht ist und führt diese ggf. durch. Die Manipulation erfolgt dabei immer zur Ladezeit und wird nicht in der Class-Datei gespeichert. Dies erhält die Flexibilität, die die Verwendung der JPDA bringt, und erzeugt aufgrund der performanten ASM-Bibliothek nur einen geringen Mehraufwand. Manipulations-Wünsche können von den Wächtern exklusiv für einzelne Klassen an diese Komponente gestellt werden. Dazu gibt es Bytecode-Manipulatoren, die den Wächtertypen entsprechen. Bisher sind dies der *BreakpointPatcher* und der *ExceptionPatcher*.

Da das alternative Backend auch ein Backend in der Java Debugging Architektur ist, können weitere Ereignistypen der JPDA implementiert werden. Jeder Ereignistyp erfordert einen entsprechenden Bytecode-Patcher, welcher durch die Implementierung einer Schnittstelle problemlos zur Komponente *Bytecode* hinzugefügt werden kann.

- **Backend** - *de.jassda.jabyba.backend*
  Diese Komponente ist für die Ausführung des Backends selbst und die des Debuggees verantwortlich. Zunächst wird das Backend gestartet und eine Verbindung zu Jassda hergestellt. Sobald Backend und Jassda verbunden sind, wird der Debuggee gestartet.

  Das Backend kennt zwei Betriebsmodi, die vom verwendeten Konnektor (2.2) abhängig sind. Wird das Backend auf dem selbem Rechner wie Jassda ausgeführt und der *Command Line Launching Connector* verwendet, verhält sich das alternative Backend wie ein Netzwerk-Client und versucht eine Verbindung zu Jassda aufzubauen. Der *Socket Connector* erfordert, dass das Backend explizit gestartet wird und Jassda die Verbindung initiiert. Das alternative Backend verhält sich also wie ein Netzwerk-Server und kann auf einem anderem Rechner ausgeführt werden.

  Neben diesen Aufgaben startet die Komponente *Backend* den Disptacher der Kommunikations-Komponente, verwaltet die Wächter und initialisiert die Laufzeitumgebung des Debuggees.

4.2 Arbeitsweise

In diesem Abschnitt soll kurz skizziert werden, wie das alternative Backend arbeitet. Exemplarisch soll betrachtet werden, wie ein Breakpoint gesetzt und somit in Jassda der Beginn bzw. das Ende einer Methode markiert wird. Bereits in Abschnitt 2.3.3 auf Seite 11 wurde aus Sicht Jassdas das Setzen eines Breakpoints gezeigt. Diesmal werden die Komponenten der oben gezeigten Architektur und ihre Zusammenarbeit im Vordergrund stehen.

In der Ausgangssituation ist der Debuggee bereits gestartet und im Begriff eine Klasse zu laden. Der folgende Prozess soll nun schrittweise dargelegt werden.

1. Das Laden der Klasse wird von einem modifizierten Klassenlader übernommen und zunächst an Jassda gemeldet. Dadurch wird die Ausführung des Debuggee und somit auch das Laden der Klasse unterbrochen. Bevor die Ausführung des Debuggees fortgesetzt wird, werden die folgenden Schritte vom Backend ausgeführt:

2. Jassda muss wissen, welche Methoden die zu ladende Klasse anbietet. Der entsprechende JDWP-Befehl wird von der Kommunikations-Komponente empfangen und an die Komponente *Runtime* delegiert. Diese hat Zugriff auf die Definition aller Klassen, ermittelt die gewünschten Methoden und sendet ein entsprechendes Antwortpaket an Jassda.

Jassda wiederholt diesen Prozess für alle Oberklassen und implementierten Schnittstellen der zu untersuchenden Klasse. Wird jedoch eine Oberklasse erreicht, die aus dem JRE stammt, meldet das alternative Backend, dass das Ende der Vererbungshierarchie erreicht ist. Dieses Verhalten wird von der Sun Binary License erzwungen (siehe dazu 4.3).

Im nächsten Schritt wählt Jassda einige der empfangen Methoden aus, um dort einen Breakpoint zu setzen. Dazu sind weitere Schritte nötig.

3. Die *LineNumberTable* einer Methode sagt aus, welche Opcodes zu welcher Zeile im Quellcode korrespondieren. Jassda braucht diese Informationen, damit ein Breakpoint nicht in die Mitte einer Zeile gesetzt wird. Der Ablauf ist analog zu Schritt 1. Die Kommunikations-Komponente empfängt ein Befehlspaket und reicht dieses an die *Runtime*-Komponente weiter. Diese sendet ein Paket, das die *LineNumberTable* enthält, an Jassda.

   Hierbei ist zu beachten, dass Jassda Breakpoints immer nur an Methodenanfängen und potentiellen Methodenenden setzt. Dieses Wissen macht sich das alternative Backend zu nutzen und sendet immer die selbe *LineNumberTable* an Jassda. Diese besteht nur aus zwei Zeilen, die Methodenanfang und -ende beschrieben. Das selbe gilt für die Opcodes, die von Jassda danach angefordert werden. Anstelle der tatsächlichen Opcodes wird immer der selbe Code geschickt, welcher nicht dem der Methode entspricht es aber erlaubt zwischen Methodenanfang und -ende zu unterscheiden. Dadurch ergeben sich zwei Vorteile. Es müssen die *LineNumberTable* und die Opcodes gar nicht ermittelt werden und die Antwort-Pakte sind i.d.R. kleiner und somit schneller zu übertragen.

   Jassda kennt nun die *LineNumberTable* und die Opcodes der Methoden. Damit sind alle Voraussetzungen erfüllt, um Breakpoints zu setzen. Es gilt zu beachten, dass die Ausführung des Debuggees immer noch unterbrochen ist.

4. Der Komponente *Backend* wird ein JDWP-Befehl zugeordent, der das Installieren eines Wächters fordert. Dazu wird im ersten Schritt ein *BreakpointEvent* instantiiert und gespeichert. Zur Auslösung des Wächters ist im nächsten Schritt eine Bytecode-Manipulation erforderlich. Daher wird eine Instanz des *BreakpointPatcher*s erzeugt und beim Klassenlader für die Klasse registriert. Dieser wird einen Methodenaufruf des entsprechenden *BreakpointEvent*s in den Code des Debuggee einfügen.

   Damit ist für Jassda das Setzen des Breakpoints beendet und die Ausführung des Debuggees kann fortgesetzt werden. Da nun für die zu ladene Klasse ein *BreakpointPatcher* registriert ist, wird die Bytecode-Manipulation direkt nach dem Fortsetzen des Debuggee durchgeführt. Danach ist auch aus der Perspektive des alternativen Backends das Setzen des Breakpoints abgeschlossen.

4.3 Einschränkungen durch die Sun Binary License

In den vorangehenden Kapiteln wurde bereits mehrfach die Sun Binary License [Sun04a] erwähnt. Anlage C des Dokuments verfügt, dass kein Programm Änderungen, gleich welcher Art, an Klassen in den Paketen *java.**, *javax.**, *sun.** sowie weiteren „SUN-üblichen Namensgebungen" vornimmt. Mit JaByBa können bzw. dürfen keine Klassen untersucht werden, die aus den genannten Paketen stammen.

Selbst wenn man versuchen wollte gegen die Lizenzbestimmungen zu verstoßen, ist dies nicht ganz leicht. Eine Möglichkeit wäre, die entsprechenden Klassen zu manipulieren und in das Archiv `rt.jar`, aus welchem die JVM die Standardklassen liest, zu schreiben. Dies ist aber nicht mit der dynamischen Manipulation, die von JaByBa verwendet wird, vereinbar. Der Versuch die Klassen durch einen modifizierten Klassenlader zu manipulieren, wird durch einen Sicherheitsmechanismus der JVM vereitelt. Das Laden der Klasse wird in diesem Fall durch eine SecurityException abgebrochen. Die einzig verbleibende Methode ist durch das *Java Virtual Machine Tool Interface* (JVMTI) gegeben. Seit der Java Version 1.5 ermöglicht diese Schnittstelle einen direkten Zugriff auf die JVM. Unter anderem ist ein `ClassFileLoadHook` definiert. Dieser ist zur Bytecode-Manipulation vorgesehen. Das JVMTI ist in der Programmiersprache `c` geschrieben, so dass eine Instrumentalisierung und Anbindung an das alternative Backend recht aufwendig ist. Daher soll betrachtet werden, wie gravierend die Einschränkungen der Binary License wirklich sind.

Jassda wurde entworfen, um Programmabläufe zu überwachen bzw. zu untersuchen. Durch die Möglichkeit unterschiedliche Module einzubinden, können die Anwendungsbereiche Jassdas zahlreich und unterschiedlich geartet sein. In der Regel haben sie aber alle gemein, dass ein entwickeltes Programm untersucht werden soll. Alle Java-Programme verwenden Klassen aus den Paketen *java.**, *javax.**, *sun.**, aber trotzdem sollte selten die Notwendigkeit gegeben, sein diese Klassen zu untersuchen. Gerade durch die Wiederverwendung bestehender Komponenten soll ein erneutes Testen/Untersuchen gespart werden. Dieser Annahme folgend kann gesagt werden, dass die Verfügungen der Sun Binary License in der praktischen Anwendung nur sehr geringe Auswirkungen haben.

Kapitel 5

Ergebnisse

5.1 Fazit

Im Rahmen dieser Arbeit wurde ein alternatives Backend für Jassda konzipiert, welches Programmabläufe durch die Manipulation von Bytecode beschreibt, ohne Jassda in Funktionalität oder Flexibilität einzuschränken. Die Implementierung hat dies anhand bestehender Tests bestätigt. Auf die Nutzung der Java Debugging-Architektur wurde bei der Ereigniserzeugung, wie gefordert, verzichtet und die Bytecode-Manipulation herangezogen. Diese erfolgt zur Laufzeit des zu untersuchenden Programms und stellt damit sicher, dass beide Backends in der Arbeitsweise gleich sind. Bei der Integration wurde auf die Java Debugging-Architektur zurückgegriffen, da Jassda diese nutzt, um ein zu untersuchendes Programm anzubinden. Damit ist die funktionale Gleichheit beider Backends und eine nahtlose Integration in Jassda erreicht worden.

Da das alternative Backend Jassdas auch ein Backend in der Java Debugging-Architektur ist, können weitere Funktionen dieser Architektur, durch Bytecode-Manipulation realisiert, implementiert werden. Bisher unterstützt das alternative Backend das Setzen von Breakpoints und das Erkennen von Exceptions, weitere Ereignistypen wie z.B. das Ändern einer globalen Variable können leicht hinzugefügt werden. Damit ist sichergestellt, dass das alternative Backend bei einer Erweiterung Jassdas nutzbar bleibt.

Neben der Entwicklung des alternativen Backends hat diese Arbeit auch einige kleinere Verbesserungen an Jassda hervorgebracht. So werden Ereignisse nun anhand von Klassennamen auf Seite des Backend vorgefiltert. Diese Einstellung verringert die Menge der zwischen Backend und Jassda ausgetauschten Daten. Zudem wurde ein Vorschlag erarbeitet, um Jassda in Bezug der JPDA-gestützten Ereigniserzeugung zu verbessern. Dieser ist durch eine Analyse der Arbeitsweise entstanden und kann auf der Internetseite des Projekts gefunden werden [Rie05].

5.2 Ausblick

Umfangreiche Tests des alternativen Backends mit vielen unterschiedlichen Klassen und Threads sowie beiden Backends zur gleichen Zeit wurden nicht durchgeführt. Diese Testfälle gilt es noch zu entwickeln. Bisher wurden vorhandene Jassda-Konfigurationen erfolgreich mit dem alternativen Backend getestet, was erwarten lässt, dass auch zukünftige Tests absolviert werden.

Desweiteren steht ein Vergleichstest aus, welcher Performanzunterschiede zwischen beiden Backends aufzeigen soll. Dabei ist eine Untersuchung hinsichtlich zweier Kriterien angeraten. Zum einem sollte untersucht werden, wie groß die zu übertragene Datenmenge zwischen Backend und Jassda ist und zum anderen wie hoch die tatsächliche Ausführungsgeschwindigkeit der Backends ist.

Neben den fehlenden Tests wurde nicht untersucht, ob das alternative Backend außerhalb des Jassda-Kontextes nutzbar ist. Das Backend wurde so entworfen, dass es JDWP-Befehle verstehen kann und damit ein möglicher Ersatz für das Backend in der Java Debugging-Architektur sein kann. Eine interessante Fragestellung wäre, ob sich durch das alternative Backend Debugging im eigentlichen Sinn realisieren lässt. Dadurch könnten Java Umgebungen, die keine Debugging-Fähigkeiten haben (wie auf Mobiltelefonen), die Bytecode-Manipulation nutzen, um rudimentäres Debugging zu ermöglichen.

Glossar

API

API steht für *Application Programming Interface* und bezeichnet eine Schnittstelle zu einem Softwaresystem. APIs werden genutzt um Softwarekomponenten in neuem Kontext einzusetzen oder zu erweitern.

Breakpoint

In einem Computerprogramm ein Anhalte-Punkt. An einem Breakpoint wird die Ausführung eines Programms unterbrochen. Dies wird von Debuggern genutzt um Momentaufnahmen von Programmen zu machen.

Exception

Exceptions *(deutsch: Ausnahme)* sind ein Sprachkonstrukt, das in unterschiedlichen Programmiersprachen, enthalten ist. Exceptions werden genutzt, um auf Fehler darzustellen, die zur Laufzeit auftreten. Man spricht beim Umgang mit Exceptions vom „Werfen" und „Fangen".

JRE - *Java Runtime Environment*

Da Java eine plattformunabhängige Sprache ist und in Bytecode übersetzt wird, ist eine Umgebung nötig, die Java-Bytecode ausführen kann. Bei dem JRE handelt es sich um diese Laufzeitumgebung. Sie enthält alle Programme die gebraucht werden, um Java Programme oder Java Applets auszuführen, sowie die Standard Java-Bibliothek.

Java SDK - *Java Software Development Kit*

Ein Programmpaket, das neben dem *JRE* auch die Programme umfasst, die für die Entwicklung von Java-Programmen benötigt werden. Das Java SDK ist auch als JDK (Java Development Kit) bekannt.

Kohäsion

Kohäsion beschreibt die Eigenschaft, dass in Softwaresystemen logisch verwandte Funktionen in einem Modul untergebracht sind. In der Softwareentwicklung wird immer eine hohe Kohäsion angestrebt.

Kupplung

Kupplung drückt die Verwobenheit von Programmteilen aus. Es wird allgemein angestrebt eine geringe Kupplung zu erreichen. So können Programmteile leichter aus dem Kontext gelöst und wiederverwendet werden.

Opcode

Ein Opcode ist ein einzelner Befehl in Class-Dateien. Java kennt 256 verschiedene Opcodes, die durch ein Byte codiert werden. Dies führt dazu, dass von Bytecode gesprochen wird.

Stack

Ein Stack *(deutsch: Kellerspeicher)* ist eine in der Informatik übliche Speicherstruktur. Ein Stack ist ein FIFO-Speicher (first in, first out), wobei nur das zuletzt hinzugefüge Element zugreifbar ist. Die Standardbefehle zum Hinzufügen bzw. Entfernen eines Elements lauten *Push* und *Pop*.

Thread

Eine Thread *(deutsch: Faden)* ist eine Unterteilung eines Prozesses in unterschiedliche Ausführungsstränge. Diese können nebenläufig ausgeführt werden.

Abbildungsverzeichnis

2.1 Gliederung des Java-Stacks in unterschiedliche Stack Frames, welche jeweils einem Methodenaufruf entsprechen. 4
2.2 Die Schichten der Java Platform Debugging Architecture. 6
2.3 Aufbau eines JDWP-Pakets. 7
2.4 Setzen der Breakpoints. 12
2.5 Ausschnitt des Stack mit mehreren Frames. Jeder Frame entspricht einem Methodenaufruf. Die oberen zwei Frames werden aufgrund einer Exception vom Stack entfernt. 13
2.6 Die JPDA ist in der Debuggee-Schicht Jassdas gekapselt und kann als Backend betrachtet werden. Das alternative Backend wird sich in die Debugge-Schicht eingliederen, muss aber noch konkretisiert werden. 15
2.7 Schematische Darstellung einer Class-Datei. 16
2.8 Das Besucher-Entwurfsmuster im UML-Diagramm. Die Klasse *JavaClass* enthält die Objektrepräsentation, wobei die Implementierungen des Besuchers unterschiedliche Funktionen darstellen. . . . 20

3.1 Drei mögliche Integrationsweisen des alternativen Backends. Von links sind dies: *Direkte Integration, Implementierung JDI*, *Implementierung JDWP*. Zu implementierende Komponenten sind durch ein Rechteck umschlossen. 26
3.2 Die Integration des alternativen Backends ist eine Ersetzung des Backends aus der JPDA und damit sowohl Backend in der Java Debugging-Architektur als auch für Jassda. 27

4.1 Debugging mit dem bisherigen und dem alternativen Backend. . . . 33
4.2 Die Architektur des alternativen Backends. 35

Literaturverzeichnis

[BLC02] Eric Bruneton, Romain Lenglet und Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. Technischer Bericht, France Telecom R&D, DTL/ASR, 2002.

[Brö02] Mark Brökens. Trace- und Zeitzusicherungen beim Programmieren mit Vertrag. Diplomarbeit, Universität Oldenburg, Department für Informatik, Januar 2002.

[Chi98] Shigeru Chiba. Javassist - A Reflection-based Programming Wizard for Java. In: *ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*. Institute of Information Science and Electronics University of Tsukuba and Japan Science and Technology Corporation, Oktober 1998.

[CN03] Shigeru Chiba und Muga Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In: *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)*, S. 364–376. Springer-Verlag, 2003.

[Dah01] Markus Dahm. Byte Code Engineering with the BCEL API. Technischer Bericht, Freie Universität Berlin, April 2001.

[GHJV94] Erich Gamma, Richard Hahn, Ralph Johnson und John Vlissichs. *Design Patterns, Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.

[HC99a] Cay S. Horstmann und Gary Cornell. *Core Java, Band I - Grundlagen*. Java Series. The Sun Microsystems Press, 1999.

[HC99b] Cay S. Horstmann und Gary Cornell. *Core Java, Band II - Expertenwissen*, Kapitel 11 Native Methoden. Java Series. The Sun Microsystems Press, 1999.

[Jav] *Open source bytecode libraries in Java*. `http://java-source.net/open-source/bytecode-libraries`.

[KND+99] Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath und Scott Hommel. *Java Code Conventions*. Sun Microsystems Inc., 1999.

[Kul04] Eugene Kuleshov. *Using the ASM Toolkit for Bytecode Manipulation*. http://www.onjava.com/pub/a/onjava/2004/10/06/asm1.html?page=1, Juni 2004.

[LY97] Tim Lindholm und Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Rie05] Johannes Rieken. *Proposal: Use of Method{Entry—Exit}Events*. https://sourceforge.net/tracker/index.php?func=detail&aid=1210919&group_id=47658&atid=450168, Mai 2005.

[Ste] Beth Stearns. *The Java Tutorial, Trail: Java Native Interface*. http://java.sun.com/docs/books/tutorial/native1.1/implementing/method.html.

[Sun98] *Java Bug Database*. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4195445, 1998.

[Sun00] Sun Microsystems Inc., http://java.sun.com/j2se/javadoc/writingdoccomments/index.html. *How to Write Doc Comments for the Javadoc Tool*, 2000.

[Sun01] Sun Microsystems Inc., http://java.sun.com/j2se/1.4.2/docs/guide/jpda/conninv.html#Connectors. *JPDA Connection and Invocation*, 2001.

[Sun04a] Sun Microsystems Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. *Binary Code Lincense Agreement*, 2004.

[Sun04b] Sun Microsystems Inc., http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdwp-protocol.html. *Java™ Debug Wire Protocol - Introduction*, 2004.

[Sun04c] Sun Microsystems Inc., http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdwp-spec.html. *Java™ Debug Wire Protocol - Specification*, 2004.

[Sun04d] Sun Microsystems Inc., http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jpda_spis.html. *JPDA Service provider interfaces*, 2004.

[Wik02] Wikipedia, http://de.wikipedia.org/wiki/Entwurfsmuster. *Entwurfsmuster*, 2002.

Index

Access rights, 15
API, 43
ASM, 31
Ausnahme, 43

Backend, 6, 14, 25, 33, 37
BCEL, 29
Beobachter-Muster, 20
Besucher-Muster, 19
Blackbox, 34
Breakpoint, 43
Broker, 8
Bytecode, 14, 29, 36

Class hierarchy, 15
Class-Datei, 14
Classloader, 5
Constant Pool, 15

Debuggee, 5, 8, 36
Debugger, 5

Entwurfsmuster, 18
Event, 6, 8, 26, 35
Exception, 13, 28, 43

Fields, 16
Frontend, 6

Garbage Collection, 5

Header, 15
Heap, 5

JaByBa, 33
Jassda, 8, 26
Java, 3
Java Virtual Machine, 3
Javassist, 30
JDI, 6, 24
JDK, 43
JDWP, 6, 25, 35
JPDA, 5
JRE, 43
JVM, 3

Klassenattribute, 17
Klassenlader, 5
Kohäsion, 34, 43
Kommunikation, 35
Konnektor, 7
Koppelung, 34
Kupplung, 44

Laufzeitumgebung, 43
Listener-Pattern, 20

Method Area, 5
Methoden, 16
Module, 8

Native Method Stack, 5

Observer-Pattern, 20
Opcode, 44

PC (Programm Counter), 4
Programmzähler, 4
Protokoll, 6

Schnittstelle, 34
SDK, 43
Singleton-Pattern, 20
Stack, 4, 44
Stack Frame, 4
Sun Binary License, 28, 39

Thread, 44

Visitor-Pattern, 19

Wächter, 6