Sperrvermerk

Die vorliegende Arbeit beinhaltet interne und vertrauliche Informationen der Firma dotSource GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma dotSource GmbH. Die Arbeit ist nur Mitgliedern des Prüfungsausschusses zugänglich zu machen.

Abstract

Die Motivation der vorliegenden Masterarbeit ist der Zeitverlust beim wiederholten *Deployment* von Java-Anwendungen auf dem Anwendungsserver im Softwareentwicklungsprozess.

In der Arbeit werden verschiedene *OpenSource* Softwareprodukte, welche versuchen den Zeitverlust zu minimieren, untersucht und verglichen, mit dem Ziel das qualifizierteste Produkt auszuwählen und zu evaluieren. Zwar gibt es bereits eine qualifizierte Software mit Namen *JRebel*, doch diese ist ein kostenintensives, kommerzielles Produkt und somit nicht für jeden erschwinglich.

Für detaillierte Untersuchungen wurde die *DCEVM* als geeignetstes Softwareprodukt ausgewählt und anschließend evaluiert. Ihre Einrichtung auf den Servern der dotSource GmbH wurde dokumentiert und eigene Anpassungen erläutert. Im Rahmen der Arbeit wurden zunächst theoretische Nachforschungen angestellt und anschließend die praktischen Prozesse bei der Einrichtung der Software und dem Programmieren eines eigenen Plugins erläutert.

Gerichtet ist die Masterarbeit an Firmen und Privatpersonen, welche über die Einführung einer *OpenSource Hot Deployment* Software nachdenken, und an Interessierte, die mehr über das Thema erfahren möchten.

I Inhaltsverzeichnis

II Abbildungsverzeichnis

III Tabellenverzeichnis

IV Quellcodeverzeichnis

1 Einleitung

Nach mittlerweile zwei Stunden Entwicklung glaubt Max endlich den Bug gefunden zu haben, der die Anwendung davon abhält korrekt zu funktionieren. Jetzt nur noch schnell die Datei speichern, die Anwendung kompilieren und schon sind die Änderungen sichtbar? Falsch. Da Max in der Webentwicklung tätig ist, muss die Anwendung zunächst auf einem Anwendungsserver deployed[1] *werden. Dazu muss ein Server-Neustart durchgeführt werden, was durchaus einige Minuten dauern kann. Die Zeit nutzt Max, um sich Gedanken über weitere Verbesserungen zu machen. Nach sechs Minuten Wartezeit ist es endlich soweit. Der Server ist gestartet, die Anwendung läuft. Gespannt ruft Max die Webseite auf, nur um festzustellen, dass die Anwendung eine* Exception *wirft und abbricht. Die Enttäuschung ist groß, die Verbesserungen sind vergessen und die Konzentration ist dahin. Größer noch wird die Enttäuschung sein, sobald Max feststellt, dass ein simpler Tippfehler die* Exception *verursacht hat. Wieder ist Warten angesagt.*

1.1 Problemstellung

Was im ersten Moment nach einer Ausnahmesituation klingt ist tatsächlich Alltag im Leben vieler Webentwickler. Beim Aktualisieren von Java-Anwendungen muss jedes Mal die komplette Anwendung *deployed* werden, statt nur der geänderten Dateien (Klassen). Unabhängig vom Zeitverlust durch das Kompilieren, Übertragen und Installieren der kompletten Anwendung impliziert das *Deployment* meist[2] einen Neustart des Anwendungsservers. Der Zeitverlust durch den wiederholten Neustart des Anwendungsservers nach praktisch jeder kleinsten Änderung im Code (z. B. Tippfehlern) ist enorm (siehe Abschnitt 2.2).

Dadurch stellte sich die in der Arbeit durch das Schlagwort *Hot Deployment* (siehe Abschnitt 2.1) umschriebene Herausforderung, ein System zu entwickeln, welches diesen Vorgang verbessert und den entstehenden Zeitverlust minimiert. Da die Herausforderung schon in mehreren Forschungsarbeiten behandelt wurde, gibt es bereits Systeme, die dazu imstande sind, diese umzusetzen. Durch die Evaluierung einer *OpenSource Hot Deployment* Software für Java im Softwareentwicklungsprozess soll eines der allgemein zugänglichen Produkte vorgestellt, untersucht und bewertet werden, welches die zuvor genannte Herausforderung bewältigt. Zusätzlich zu dem Schlagwort *Hot Deployment*, finden sich unter *Dynamic Code Evolution*, *Dynamic*

[1] *Deployment* umfasst im Bereich Webentwicklung alle Prozesse, die zur Installation einer Anwendung auf dem Anwendungsserver nötig sind

[2] Abhängig vom Anwendungsserver, *Build-Tools*, Frameworks etc.

Software Updates, *Runtime Evolution* oder *Hotswapping* weitere Informationen zu der Herausforderung.

1.2 Vorstellung der dotSource GmbH

Die dotSource GmbH wurde im Jahr 2006 (vgl. [dota]) von den Geschäftsführern Christian Otto Grötsch und Christian Malik gegründet und erlebt seither ein stetiges Wachstum. Nach nur 10 Jahren arbeiten in dem Unternehmen bereits über 130 Mitarbeiter, Tendenz steigend. Ziel und gleichzeitig Vision der dotSource GmbH ist, anderen Unternehmen einen Weg in die digitale Zukunft von Marketing und Vertrieb zu ebnen. Zahlreiche Unternehmen aus Deutschland, Österreich und der Schweiz wurden und werden bei der digitalen Transformation und der Inszenierung ihrer Marken im Internet unterstützt. Das Leistungsspektrum der dotSource GmbH umfasst nach dem Grundsatz des Unternehmens „digital success right from the start“ alle Aspekte von der ersten Idee (Strategieberatung) über die Entwicklung und Umsetzung innovativer Konzepte bis hin zur Betreuung der Unternehmen nach dem Livegang. Inzwischen hat sich die dotSource GmbH als eine der führenden Digitalagenturen im deutschen Sprachraum etabliert. Sie gehört zu den Top 20 der größten Unternehmen der Branche (vgl. [int]) und erzielte Platz 2 im Einzelranking der wachstumsstärksten Agenturen des Internetagenturrankings des BVDW (2012).

Mittlerweile vertrauen zahlreiche, teilweise multinationale Unternehmen wie Swarovski, Cornelsen, Hagebau, Würth und Music Store auf die Leistungen der dotSource GmbH. Allerdings ist die Bereitstellung und Betreuung von Online-Shops nicht das einzige Gebiet in dem die dotSource sich engagiert. Zusätzlich veröffentlicht die dotSource GmbH zahlreiche Publikationen wie den Weblog Handelskraft.de, das jährlich erscheinende Trendbuch sowie verschiedene Whitepaper und organisiert Veranstaltungen wie die Handelskraft Konferenz. Hierbei werden aktuelle Trends und Perspektiven im Digitalmarkt vorgestellt und diskutiert. Aktuell betreut die dotSource GmbH über 100 Shops und Webseiten für 80 Kunden aus 30 Ländern (vgl. [dota], [dotb]) und kommuniziert dabei in acht verschiedenen Sprachen. Sie punktet insbesondere durch Einsatz, Hilfsbereitschaft und hohe Dynamik bei den Kunden und sichert sich so nachhaltigen Erfolg im Digitalmarkt.

Bemüht um einen stetigen Zustrom von neuen, kompetenten Mitarbeitern arbeitet die dotSource GmbH zudem eng mit der Friedrich-Schiller-Universität in Jena zusammen, lässt Auszubildende und Studenten durch Praktika Erfahrungen sammeln und ermöglicht es, wissenschaftliche Arbeiten unter professioneller Betreuung anzufertigen. Da von diesem Arrangement beide Seiten profitieren sollen, beschäftigt sich

die nachfolgende Arbeit mit der Evaluierung und Installation eines Tools, welches dem Unternehmen potenziell finanzielle Erleichterung ermöglicht.

1.3 Zielsetzung und Zweck der Arbeit

Das gegenwärtig ausgereifteste Tool, das die Herausforderung *Hot Deployment* bewältigt, heißt *JRebel* und wird aktuell von der dotSource GmbH genutzt. *JRebel* befasst sich mit der Möglichkeit Java-Klassen zur Laufzeit in die JVM (Java Virtual Machine) zu laden und somit Code-Änderungen sofort sichtbar zu machen. Dabei wird viel Zeit (siehe Abschnitt 2.2) eingespart, da auf der einen Seite nur die Klassen, in denen tatsächlich Änderungen stattfanden, betrachtet werden und auf der anderen Seite Server-Neustarts beinahe vollständig entfallen.

Allerdings ist *JRebel* ein kommerzielles Produkt. Da eine einzelne *JRebel*-Lizenz 475 $ pro Jahr kostet (vgl. [Zera]), bedeutet dies für die dotSource GmbH mit ihren ca. 100 Softwareentwicklern einen Kostenpunkt von 47.500 $[3] pro Jahr. Das mag zwar eine erschwingliche Ausgabe für eine große Softwarefirma sein. Vor allem unter Neuzugängen (Trainees) und einigen der Anfänger (Juniors) sind *JRebel*-Lizenzen jedoch eine Seltenheit, ganz zu schweigen von Praktikanten, Masteranden, Studenten einer Berufsakademie etc., da sich die Ausgabe oft noch nicht rentiert.

Ziele dieser Arbeit sollen daher das Auffinden und Evaluieren einer geeigneten *Open-Source Hot Deployment* Software für die Anwendungsentwicklung mit Java und das Installieren und Anpassen dieser auf den Servern der dotSource GmbH sein. Dies umfasst die Suche nach verschiedenen *OpenSource* Produkten, den Vergleich dieser, den Prozess der Entscheidungsfindung, das Einrichten des entsprechenden Tools auf dem System der dotSource GmbH, die Anpassung des Tools auf die Bedürfnisse der Firma durch die Programmierung eines Plugins und anschließend die Evaluierung des eingerichteten Tools.

Die dotSource GmbH steht mit dem oben genannten Problem nicht allein da. Viele Unternehmen stören sich an den hohen und tendenziell noch steigenden Kosten von *JRebel*-Lizenzen. Dadurch wird die Suche nach einer kostenlosen Alternative auch für sie zu einem Problem von wachsender Dringlichkeit. Daher liegt der zusätzliche Nutzen dieser Arbeit in der Entscheidungshilfe für Firmen und Privatpersonen, die vor einer ähnlichen Wahl stehen. Zudem werden Kriterien herausgearbeitet, an denen, zusätzlich zu den in der Arbeit vorgestellten Tools, auch zukünftige und selbst entwickelte Tools gemessen werden können.

[3] ca. 45.000 €

1.4 Aufbau und Vorgehensweise

Der erste Schritt zur Umsetzung der Zielsetzung liegt in der Informationsbeschaffung. Diese umfasst unter anderem das Entwickeln eines grundlegenden Verständnisses für *Hot Deployment* und die Recherche nach vorhandenen Lösungsansätzen. Dazu wird in Kapitel 2 zunächst der Begriff *Hot Deployment* definiert (Abschnitt 2.1) und der tatsächliche Zeitverlust ohne *Hot Deployment* in Zahlen erfasst (Abschnitt 2.2). Anschließend werden die verschiedenen Tools untersucht und verglichen (Abschnitt 2.3).

Nachdem genügend Informationen über die Tools gesammelt wurden, wird das potenziell vielversprechendste Tool ausgewählt und auf dessen Grundlage eine detaillierte Untersuchung der *Hot Deployment* Technologie durchgeführt werden. In diesem Zusammenhang werden in Kapitel 3 spezielle Anforderungen und Konzepte von *Hot Deployment* Tools untersucht. Auch die Einbindung des Tools in Anwendungen spielt in diesem Kapitel eine wichtige Rolle.

Kapitel 4 beschäftigt sich mit der Einrichtung des geeignetsten Tools. Zuerst wird in Abschnitt 4.1 die Wahl des Tools begründet und anschließend wird unter Abschnitt 4.2 seine Installation auf zwei verschiedenen Systemen erläutert. In Abschnitt 4.3 werden anschließend weitere Funktionalitäten des gewählten Tools untersucht und es wird ein eigenes Plugin vorgestellt, welches die Software erweitert.

Das 5. Kapitel (Evaluation) untersucht die Eignung der Software zur Erfüllung des angestrebten Zwecks. Dabei wird insbesondere der praktische Einsatz betrachtet. In den Abschnitten 5.1.1 und 5.1.2 wird das gewählte Tool anhand ausgewählter Kriterien mit dem kommerziellen Produkt *JRebel* verglichen und bewertet und in Abschnitt 5.2 wird der Nutzen des eigenen Plugins evaluiert.

Im sich anschließenden Fazit werden die Erkenntnisse und Ergebnisse dieser Arbeit zusammengefasst.

2 Grundlagen/Stand der Technik

Dieses Kapitel hat den Zweck eine Einführung in die Thematik zu geben. Nachdem in Abschnitt 2.1 erklärt wird, was *Hot Deployment* bedeutet, wird in Abschnitt 2.2 näher ausgeführt wieso es überhaupt notwendig ist. Zu diesem Zweck wird der Zeitverlust, den ein Webentwickler im Alltag erfährt, in Zahlen gefasst. Anschließend werden in Abschnitt 2.3 einige aktuell existierende Tools auf wichtige Eigenschaften untersucht. Ziel ist die Analyse der Tools zum Zweck der Entscheidungsfindung. Die zu beantwortende Frage ist, welches Tool der Firma den potentiell größten Nutzen bringt.

2.1 Definition von Hot Deployment

Das Entwickeln von Anwendungen im Bereich der Webentwicklung umfasst in der Regel folgende vier Phasen, die sich in einem stetigen Kreislauf wiederholen (vgl. [Zerb]):

- Phase 1: Der Entwickler programmiert mithilfe seiner Entwicklungsumgebung. Code-Änderungen finden statt
- Phase 2: Der aktualisierte Code wird kompiliert bzw. gebaut
- Phase 3: Die Anwendung wird *deployed*
- Phase 4: Änderungen werden sichtbar

Problematisch ist, dass Phase 3 meist einige Minuten in Anspruch nimmt und immer wieder durchlaufen wird, da die Phasen einen Zyklus bilden. Obwohl die Anwendung bereits auf dem Server installiert wurde, muss sie nach jeder Änderung neu installiert werden. Der dadurch entstehende Zeitverlust ist groß (Abschnitt 2.2).

Die Idee, die unter dem Begriff *Hot Deployment* zusammengefasst wird, ist, dass das eigentliche *Deployment* (Phase 3) nur einmalig durchgeführt wird. Dabei werden Code-Änderungen auf eine Art und Weise auf den Server übertragen, durch die die Anwendung, welche auf dem Server bereits *deployed* wurde, nicht *redeployed* (erneut *deployed*) werden muss. Stattdessen wird die Anwendung *on-the-fly* aktualisiert, d.h. Code-Änderungen finden zur Laufzeit statt. Phase 3 entfällt.

Tools, die *Hot Deployment* umsetzen, beobachten in der Regel die *.class*-Dateien, die der Compiler erstellt. Ändern sich diese, werden sie nun auch auf dem Server aktualisiert, wobei einige Code-Änderungen einfacher in die laufende JVM übertragen

werden können als andere. Die dazu benötigte Technologie ist kompliziert und soll in Kapitel 3 genauer betrachtet werden.

2.2 Zeitverlust ohne Hot Deployment

Um herauszufinden wie viel Zeit beim Entwickeln ohne *Hot Deployment* verloren geht, führte ZeroTurnaround (Hersteller von *JRebel*) im Jahr 2011 eine Umfrage mit etwa 1100 Java-Entwicklern durch, die auf der ganzen Welt verteilt leben und Fragen über ihren Entwicklungsalltag beantwortet haben (vgl. [JKa]). Wichtig waren insbesondere folgende Fragen:

- Wie lange dauert ein *Redeploy* bzw. Neustart?
- Wie häufig wird ein *Redeploy* durchgeführt?

Allerdings wurden auch Fragen zu den *Build-Tools*, der IDE (Integrated Development Environment), dem *Webcontainer*, dem Java EE[4] Standard und dem *Webframework* beantwortet. Diese lieferten wichtige Informationen über die Frage, welche Plugins bei der Entwicklung Vorrang haben sollten. In den Abbildungen 1 und 2 wurden einige der wichtigsten Ergebnisse zusammengefasst[5].

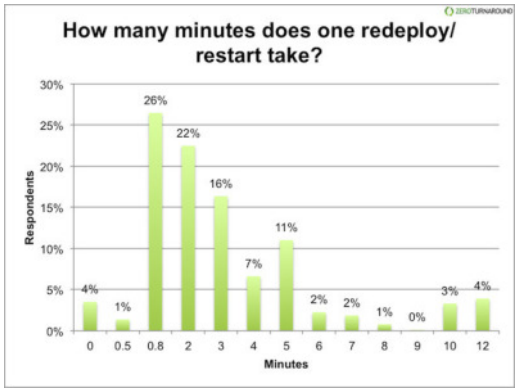

Abbildung 1: Statistik über die Dauer eines *Redeploys*

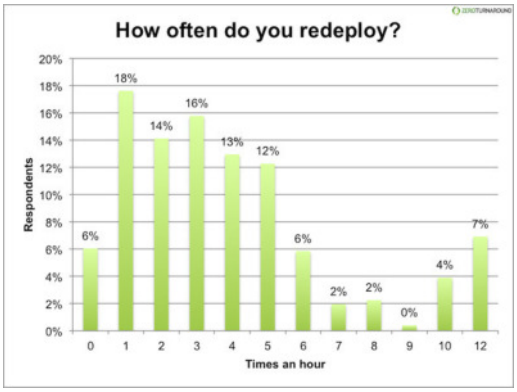

Abbildung 2: Statistik über die Anzahl an *Redeploys*

In Abbildung 1 wird dargestellt, wie viele Minuten ein einzelner *Redeploy* braucht. Berechnet man den Durchschnitt über die Angaben erhält man eine durchschnittliche *Redeploy*-Dauer von etwa 3,1 Minuten. Allerdings beträgt die Standardabweichung 2,8, was bedeutet, dass die Angaben sehr stark variieren.

[4] Java Platform, Enterprise Edition
[5] Beide Abbildungen stammen aus der Quelle [JKa]

Abbildung 2 zeigt, wie oft pro Stunde ein *Redeploy* durchgeführt werden muss. Hier beträgt die durchschnittliche Anzahl vier Mal pro Stunde und die Standardabweichung ist 3,2.

Mithilfe dieser beiden Werte kann berechnet werden, wie viel Zeit der durchschnittliche Entwickler verliert, während er auf einen *Redeploy* wartet. Mit vier *Redeploys* pro Stunde und einer *Redeploy*-Dauer von 3,1 Minuten ergibt sich ein Zeitverlust von 12,4 Minuten pro Stunde. Das entspricht 20,7 % einer Stunde und somit etwa einem Fünftel der Gesamtzeit eines Entwicklers.

Da allerdings die Standardabweichung beider Statistiken sehr groß ist, ist die Frage nach dem Zeitverlust speziell bei der dotSource GmbH von entscheidender Bedeutung. Zu diesem Zweck wurde auch in der dotSource GmbH eine Umfrage durchgeführt, die zu ähnlichen Ergebnissen führte.

In der Firma werden zahlreiche Projekte in den verschiedenen Teams umgesetzt, oft auch mehrere Projekte pro Team. Diese Projekte haben je nach Anforderungen und Komplexität sich sehr stark unterscheidende *Redeploy*-Zeiten und auch die Anzahl der *Redeploys* unterscheidet sich stark. In den kleinsten Projekten wird meißt nur einmal alle ein bis zwei Stunden ein *Redeploy* durchgeführt und dieser dauert im Schnitt unter vier Minuten, was einem durchschnittlichen Zeitverlust von weniger als drei Minuten pro Stunde entspricht. Dagegen stehen die großen Projekte mit den einflussreichen Auftraggebern. Als größter Zeitverlust eines Projekts ergab sich in der Umfrage ein Wert von 36 Minuten pro Stunde, mit vier *Redeploys* pro Stunde und etwa neun Minuten Dauer. Der durchschnittliche Zeitverlust über alle Projekte hinweg beträgt ca. 14,2 Minuten pro Stunde, was etwas mehr ist als der in der Umfrage von ZeroTurnaround ermittelte Zeitverlust.

Natürlich schauen die Entwickler, vor allem in den größeren Projekten, nicht die ganze Zeit beim *Redeploy* zu und tun nichts weiter, sondern sie wenden sich anderen Aufgaben zu. Trotzdem sollte der Zeitverlust durch das ständige Umdenken und das Bearbeiten mehrerer Probleme gleichzeitig nicht unterschätzt werden.

2.3 Vorhandene Lösungsansätze/Tools

Die folgenden Abschnitte sollen dem Leser eine Übersicht über die verschiedenen bereits existierenden Tools geben, die *Hot Deployment* umsetzen. Dabei soll auf ihre Implementierung, ihre Beschränkungen, ihre Stärken und Schwächen und auf ihre Unterstützung der verschiedenen Technologien (*Webcontainer*, *Webframeworks*, ...) näher eingegangen werden.

2.3.1 HotSwap

Im Jahr 2002 führte das Unternehmen *Sun Microsystems* eine neue experimentelle Technologie ein und integrierte sie in die Java 1.4 JVM (vgl. [JKb]). Diese Technologie wird *HotSwap* genannt und wurde in die *Debugger*[6] API (Application Programming Interface) eingebaut. Sie erlaubt es dem *Debugger* Updates von Klassen-Bytecode an Ort und Stelle durchzuführen, ohne dass dabei die Identität der Klasse angefasst wird. Das bedeutet, dass alle bereits existierenden Objekte der aktualisierten Klasse bei einem Methodenaufruf schlichtweg den neuen Code ausführen. Dabei ändert sich die Struktur der Klasse nicht, weshalb keine weiteren Änderungen am Bytecode notwendig sind. Die Technologie wird heutzutage von den meisten IDEs unterstützt und steht seit Java 5 durch die *Instrumentation API* den Java-Anwendungen selbst zur Verfügung.

Die große Schwäche der *HotSwap*-Technologie liegt in ihren Beschränkungen. *HotSwap* ermöglicht lediglich Änderungen im Körper von Methoden. Weder Änderungen ihrer Signatur noch das Hinzufügen/Ändern von Methoden und Feldern oder sonstige Änderungen zur Laufzeit werden von dieser Technologie unterstützt. Der Grund dafür liegt in der Komplexität der JVM, die es schwer macht Änderungen in der Struktur von Klassen durchzuführen. Weitere Informationen dazu liefert die Quelle [JKb] unter der Überschrift: "*Why is HotSwap limited to method bodies?*".

Es folgt eine Tabelle, die auch in den nächsten Abschnitten wiederholt Verwendung finden wird, mit dem Ziel die Funktionalitäten und Beschränkungen der verschiedenen Tools darzustellen und später zu vergleichen.

Zur Laufzeit mögliche Änderungen	**HotSwap**
Änderungen im Körper von Methoden	✓
Hinzufügen von Methoden	**X**
Entfernen von Methoden	**X**
Hinzufügen von Feldern	**X**
Entfernen von Feldern	**X**
Hinzufügen von Klassen	**X**
Entfernen von Klassen	**X**
Hinzufügen von Superklassen	**X**
Entfernen von Superklassen	**X**

Tabelle 1: Übersicht über die zur Laufzeit möglichen Änderungen durch *HotSwap*

[6] Ein Programm zum Auffinden von Fehlern bei der Softwareentwicklung

Tabelle 1 zeigt einige der häufigsten und wichtigsten Änderungen, die ein Entwickler in seinem Alltag durchführt. Deutlich sichtbar ist, dass die meisten Änderungen nicht unterstützt werden, wodurch der Entwickler in diesen Fällen gezwungen ist einen *Redeploy* oder sogar einen Server-Neustart durchzuführen, wenn er die Änderungen im Code übertragen möchte. Dementsprechend ist die Idee hinter der Technologie zwar gut, in der Praxis ist sie allerdings nur von begrenztem Nutzen, da bloße Änderungen im Körper von Methoden meist nicht ausreichen. Um den Zeitverlust (siehe Abschnitt 2.2) entscheidend zu verringern müssten die meisten der in Tabelle 1 dargestellten Änderungen unterstützt werden.

2.3.2 JRebel

Im Jahr 2007 veröffentlichte die Firma *ZeroTurnaround* ein Tool mit dem Namen *JRebel*[7] (damals *JavaRebel*). Ziel war es den zeitaufwendigen *Deployment*-Kreislauf (siehe Abschnitt 2.1) mit einer Dauer von teilweise über 12 Minuten (siehe Abbildung 1) zu verbessern. Ursprünglich war *JRebel* als eine Erweiterung der *Hot-Swap*-Technologie gedacht, die zusätzlich zu dem Aktualisieren der Methodenkörper weitere Code-Änderungen zur Laufzeit erlaubt, wie das Hinzufügen von Methoden, Feldern und Klassen. Doch mittlerweile (Stand: 2012) unterstützt *JRebel* eine viel größere Bandbreite von Änderungen. Dies erlaubt es dem Entwickler beinahe jederzeit das *Packaging*[8] und das *Deployment* zu umgehen. Ausnahmen, in denen es nötig ist beides durchzuführen, gibt es nur wenige. Dadurch konnte sich *JRebel*, nach seinem anfangs bescheidenen Start, zu **dem** Tool für *Hot Deployment* weiterentwickeln und zu einem großen Zeitsparer im Java EE System werden.

Mittlerweile nutzen Zehntausende von Programmierern *JRebel*, Tendenz steigend. Durch die fehlende Konkurrenz konnte sich *JRebel* eine Monopolstellung im Bereich *Hot Deployment* sichern, womit natürlich auch die Möglichkeit Preise nach eigenem Ermessen zu definieren einhergeht. So hat *ZeroTurnaround* im November 2015 den Preis für *JRebel*-Lizenzen von 365 $ um ca. 30 % auf 475 $ pro Entwickler und Jahr erhöht. Ausgehend von Zehntausenden von Kunden entspricht dies einer Erhöhung des Gewinns der Firma von mehreren Millionen Dollar. Natürlich wächst das Unternehmen *ZeroTurnaround*, wodurch auch die Kosten ständig steigen. Da die Gelder jedoch zu einem nicht unerheblichen Teil in die Entwicklung von weiteren Plug-

[7] Quelle für diesen Abschnitt ist der veröffentlichte Bericht [JK12]

[8] Webanwendungen werden vor dem *Deployment* in ein ZIP-Archiv mit einer *.war* oder *.ear* Dateiendung gepackt

ins[9] fließen und aufgrund der rapiden Preisentwicklung[10] suchen mittlerweile viele Unternehmen nach kostengünstigen Alternativen.

Doch obwohl der Preis für *JRebel* hoch ist, hat *JRebel* allen anderen Versuchen *Hot Deployment* umzusetzen einiges voraus. Trotz der vielen Forschungsteams[11], die sich mit *Hot Deployment* beschäftigt haben, war *JRebel* das erste Tool, das die folgenden Herausforderungen umgesetzt hat:

- Implementierung von *Hot Deployment* auf der Standard-JVM mit Unterstützung für verschiedene Anbieter (HotSpot JVM, JRockit JVM, J9 JVM)
- Bewahrung aller Zustände der Anwendung während der Updates
- Kein sichtbarer Effekt auf das Verhalten der Anwendung
- Kein erkennbarer Effekt auf die Performance der Anwendung
- Kein signifikanter Effekt auf die Speichernutzung der Anwendung

Die vier letzten Punkte sind Kriterien, die jedes Tool, das *Hot Deployment* umsetzt, erfüllen sollte, weshalb auch einige der folgenden Tools auf diese Kriterien geprüft werden sollen. Gäbe es sichtbare Effekte auf das Verhalten der Anwendung, wäre es schwer das entsprechende Tool im *Debugging* zu nutzen. Es könnten unvorhergesehene Fehler auftreten, die die Anwendung zum Absturz bringen und einen Neustart des Anwendungsservers erforderlich machen. Zudem würde es zu Verwirrungen bei der eigentlichen Fehlersuche führen, wenn ein Fehler nur manchmal auftritt.

Natürlich sollte das entsprechende Tool, wie im vierten Kriterium beschrieben, auch keinen deutlichen Effekt auf die Performance der Anwendung haben. Gerade beim *Debugging* von Webanwendungen kann es ärgerlich sein, wenn die Zeit für Serveranfragen wahrnehmbar erhöht ist und somit jeder Klick länger dauert. Auch der im fünften Kriterium beschriebene Effekt auf die Speichernutzung der Anwendung sollte möglichst gering ausfallen. Bei einem großen Speicherverbrauch würde der Server früher oder später keinen Speicher mehr zur Verfügung haben (OutOfMemoryException) und es wäre erneut ein Neustart des Anwendungsservers notwendig, was dem Zweck des Tools entgegenwirkt.

Die Bewahrung aller Zustände der Anwendung während der Updates ist schlichtweg eine Voraussetzung für die korrekte Funktionsweise eines *Hot Deployment* Tools. Würden Zustände verloren gehen, könnte auch die Anwendung nicht fehlerfrei lau-

[9] Die meistgenutzten Plugins sind längst abgedeckt
[10] 2010 lag der Preis noch bei 195 $ pro Entwickler und Jahr
[11] Zum Beispiel *Javeleon* (Abschnitt 2.3.3), *Jvolve* (Abschnitt 2.3.4), *DCEVM* (Abschnitt 2.3.5)

fen, sodass auch hier ein Neustart des Anwendungsservers nötig wäre. Zudem könnten zufällige Fehler den *Debugging*-Prozess deutlich erschweren. Übrigens erfüllt die *HotSwap*-Technologie ebenfalls die vier genannten Kriterien.

Inwieweit *JRebel* die *HotSwap*-Technologie erweitert kann man deutlich in Tabelle 2 sehen.

Zur Laufzeit mögliche Änderungen	**JRebel**
Änderungen im Körper von Methoden	✓
Hinzufügen von Methoden	✓
Entfernen von Methoden	✓
Hinzufügen von Feldern	✓
Entfernen von Feldern	✓
Hinzufügen von Klassen	✓
Entfernen von Klassen	✓
Hinzufügen von Superklassen	✓
Entfernen von Superklassen	✓

Tabelle 2: Übersicht über die zur Laufzeit möglichen Änderungen durch *JRebel*

Die wichtigsten Änderungen die ein Entwickler in seinem Alltag durchführt, werden ohne Ausnahme von *JRebel* unterstützt. Somit kann der Programmierer bei der Nutzung von *JRebel* beinahe vollständig auf Server-Neustarts verzichten und sich ohne Unterbrechung um das *Debugging* kümmern. Der Zeitverlust (siehe Abschnitt 2.2) wird deutlich reduziert und die Konzentration des Entwicklers nur noch selten gestört.

2.3.3 Javeleon

Javeleon ist ein Tool, das am *Maersk McKinney Moeller Institute* in der *University of Southern Denmark* im Rahmen der Forschung entwickelt wurde[12]. Obwohl viele Leute an dem Projekt mitgearbeitet haben, gilt Dr. Allan Gregersen als Hauptbegründer von *Javeleon*, wie auch seine zahlreichen Publikationen zeigen ([ARG12], [ARG11b], [ARG11a]).

Anreiz von *Javeleon* ist, wie schon bei *JRebel*, der stetige Entwicklungskreislauf (siehe Abschnitt 2.1) bei der Implementierung von Software mit Java als statisch typisierter Sprache. Trotz der vielen Forschung zum Thema *Hot Deployment* hatte

[12]Hauptquelle für diesen Abschnitt ist [ARG12]

sich herausgestellt, dass die unbeschränkte Code-Evolution[13] noch immer eine große Herausforderung war. Obwohl *JRebel* zur Zeit der Entwicklung von *Javeleon* bereits existierte und auch ein wichtiger Meilenstein war, hatte auch *JRebel* zu dieser Zeit noch viele Schwächen. So unterstützte *JRebel* noch keine Änderungen der Klassenhierarchie. Zwar sind vereinzelte Server-Neustarts bei der Softwareentwicklung nicht weiter kritisch, doch interessierte sich *Javeleon* zusätzlich für den Einsatz in Produktionssystemen. Dort kann diese Schwäche einen Stopp erzwingen, in einem System, das eigentlich 24/7 (24 Stunden am Tag, 7 Tage die Woche) zur Verfügung stehen sollte. Zudem würden solche Beschränkungen den Nutzen des Tools auf Forschungsplattformen, in Gebieten wie der Selbst-Adaption, reduzieren. Ziel von *Javeleon* ist die Bereitstellung eines hochqualitativen Tools zur Umsetzung der Code-Evolution sowohl für die Industrie als auch für die akademische Welt.

Der Kerngedanke hinter *Javeleon* ist, dass auf dem laufenden System gleichzeitig mehrere Repräsentationen der Klassen und Objekte koexistieren können. Dies setzt *Javeleon* um, indem es bei jedem dynamischen Update neue Klassenlader erstellt, wodurch zugleich ein eigenständiger Namensraum für Typen aufgebaut wird. Dieser Ansatz bewirkt eine Inkompatibilität zwischen den verschiedenen Klassen und Objekten, auch Versions-Barriere[14] genannt, sodass *Javeleon* eine Möglichkeit beinhalten muss diese zu überwinden.

Die Architektur von *Javeleon* besteht aus folgenden vier Komponenten:

- JVM Bootstrapping
- Bytecode Transformer
- Core Execution
- Framework Integration Plugin

Die Komponente *JVM Bootstrapping* hat die Aufgabe die Empfindung eines dynamischen und transparenten Prozesses zu erzeugen, indem sie den Bytecode der speziellen Bootstrap-Klassen transformiert. Da diese nicht zur Laufzeit modifiziert werden können, muss bereits zum Start der JVM eine Menge von modifizierten Bootstrap-Klassen bereitgestellt werden. Dabei wird auf statische Vorverarbeitung zurückgegriffen.

[13] Das wiederholte Aktualisieren von Klassen kann auch als Weiterentwicklung/Evolution von Code betrachtet werden

[14] Da der Typ von allen Klassen, die nacheinander von unterschiedlichen Klassenladern geladen wurden, vom Java-Typsystem als verschieden angenommen wird, verbietet dieses die Korrespondenz zwischen Objekten der gleichen Klassen, wenn sie von unterschiedlichen Klassenladern geladen wurden.

Der *Bytecode Transformer* ist verantwortlich für die Transformation des Bytecodes von Klassen, wenn sie in die JVM geladen werden. Genau genommen werden die Klassen für dynamische Updates aktiviert.

Die *Core Execution* Komponente implementiert das zugrundeliegende Modell für dynamische Updates. Sie ist verantwortlich für die Handhabung von Anfragen vom transformierten Bytecode und zwingt die Eigenschaften der Sprache Java in eine versionierte Klassenumgebung.

Das *Framework Integration Plugin* ermöglicht das Integrieren von speziellen Frameworks. Genauer gesagt, erlaubt die Komponente dem Programmierer das Schreiben von Plugins mit dem Zweck alle Probleme zu beheben, die nicht durch einfache Updates der Java-Klassen behandelt werden können. Dementsprechend wird *Javeleon* durch die Plugins um weitere Funktionalitäten erweitert, sodass *Javeleon* auch unter Benutzung spezieller Frameworks verwendet werden kann. Das einzige von *Javeleon* selbst unterstützte Framework ist *NetBeans Platform*.

Tabelle 3 zeigt nun eine Übersicht über die von *Javeleon* bereitgestellten Funktionalitäten[15].

Zur Laufzeit mögliche Änderungen	**Javeleon**
Änderungen im Körper von Methoden	✓
Hinzufügen von Methoden	✓
Entfernen von Methoden	✓
Hinzufügen von Feldern	✓
Entfernen von Feldern	✓
Hinzufügen von Klassen	✓
Entfernen von Klassen	✓
Hinzufügen von Superklassen	✓
Entfernen von Superklassen	✓

Tabelle 3: Übersicht über die zur Laufzeit möglichen Änderungen durch *Javeleon*

Wie schon *JRebel* unterstützt auch *Javeleon* alle Änderungen, die im Alltag eines Entwicklers anstehen. Somit wird die Anzahl an Server-Neustarts stark reduziert, wenn nicht komplett eliminiert. Auch die meisten der in Abschnitt 2.3.2 aufgezählten Kriterien setzt *Javeleon* um. Zustände der Anwendung werden während der Updates bewahrt und einen sichtbaren Effekt auf das Verhalten der Anwendung gibt es bei

[15] Vgl. [ARG11a] - Kap. 3, [ARG11b] - Kap. 3

Javeleon nicht. Der Einfluss auf die Performance ist äußerst gering (vgl. [ARG12] - Kap. 1, Unterpunkt: Performance) und auch der Effekt auf die Speichernutzung ist nicht signifikant.

Wieso *Javeleon* leider nicht als konstengünstige Alternative zu *JRebel* in Frage kommt, kann in Abschnitt 4.1 nachgelesen werden.

2.3.4 Jvolve

Jvolve ist eines der ersten Tools, die entwickelt wurden um *Hot Deployment* umzusetzen. Mit der Entwicklung des Tools beschäftigten sich Suriya Subramanian, Michael Hicks und Kathryn S. McKinley (vgl. [KSM09]). Zur Umsetzung von *Hot Deployment* benutzt *Jvolve* einen neuartigen Ansatz, die Bereitstellung einer modifizierten JVM. Im Speziellen wurde die Jikes RVM (Research Virtual Machine) ausgewählt. Dies ist eine JVM, welche als offene Testumgebung für die Entwicklung von und dem Experimentieren mit neuen VM[16]-Technologien entworfen wurde. Dabei führt *Jvolve* Anpassungen bei verschiedenen Mechanismen wie zum Beispiel der *just-in-time Kompilierung*[17], dem *Scheduling* oder der *Garbage Collection*[18] durch.

Auch *Jvolve* unterstützt einiges mehr als nur Änderungen im Körper von Methoden, wie in Tabelle 4 zu sehen ist.

Zur Laufzeit mögliche Änderungen	**Jvolve**
Änderungen im Körper von Methoden	✓
Hinzufügen von Methoden	✓
Entfernen von Methoden	✓
Hinzufügen von Feldern	✓
Entfernen von Feldern	✓
Hinzufügen von Klassen	✓
Entfernen von Klassen	✓
Hinzufügen von Superklassen	**X**
Entfernen von Superklassen	**X**

Tabelle 4: Übersicht über die zur Laufzeit möglichen Änderungen durch *Jvolve*

Jvolve erlaubt das Hinzufügen, Entfernen und Verändern von Klassen, sowie alle anderen Änderungen die damit einhergehen. Dazu zählen das Hinzufügen, Entfernen

[16] Virtuelle Maschine
[17] *Kompilierung* während der Programmausführung
[18] Automatische Speicherbereinigung - siehe Abschnitt 3.2.2.3

und Modifizieren von Methoden und Feldern, sowie Änderungen der entsprechenden Typ-Signaturen. Dementsprechend unterstützt *Jvolve* alle Standard-Änderungen[19], die sich **innerhalb** einer Klasse abspielen. Diese Änderungen können an jedem Punkt der Klassenhierarchie stattfinden und werden an die Unterklassen weitergereicht.

Unabhängig von den vielen Änderungen die *Jvolve* unterstützt, soll an dieser Stelle nochmals klargestellt werden, dass *Jvolve* ausschließlich als Tool für die Forschung entwickelt wurde. Es modifiziert die Jikes RVM, welche lediglich als Testumgebung dient. Der einzige online verfügbare Release ist speziell für Forscher gedacht (vgl. [SS]). Für Entwickler steht *Jvolve* aktuell nicht zur Verfügung. Trotzdem ist die Idee einer modifizierten JVM ein neuer und interessanter Ansatz, der auch im folgenden Tool wieder aufgegriffen wird.

2.3.5 DCEVM

Genau wie die anderen Tools hat auch die *DCEVM* den Zweck *Hot Deployment* bzw. die *dynamische Code-Evolution* umzusetzen. Die Entwicklung der *DCEVM* geschah durch Dipl.-Ing. Dr. Thomas Würthinger, welcher sie in seiner Dissertation zur Erlangung des akademischen Grades *Doktor* erstmals vorstellte. Die Doktorarbeit wurde am *Institut für Systemsoftware* an der *Johannes Kepler Universität in Linz* (JKU) angefertigt und im März 2011 abgegeben. Sie ([TW11]) bildet gemeinsam mit einem zusammenfassenden Konferenzbericht ([TW10]) die Hauptquelle für diesen Abschnitt. Zusätzlich wurden einige Informationen von der Webseite der JKU (vgl. [JKUL]) verwendet.

Nach seiner Dissertation übernahm er die Rolle als Projektkoordinator, um die Entwicklung weiter voranzutreiben. Unterstützung erfährt er durch die Entwickler Kerstin Breitender und Christoph Wimberger, welche an der *JKU in Linz* arbeiten, sowie durch zahlreiche Ratgeber aus verschiedenen Universitäten und Unternehmen, darunter Größen wie Oracle und Google.

Wie schon der Name andeutet, ist die *DCEVM* (Dynamic Code Evolution Virtual Machine) eine Modifikation der Java $HotSpot^{TM}$-VM. Dabei wird der bereits vorhandene *HotSwap*-Mechanismus der VM um einige komplexere Änderungen erweitert. Die Erweiterung der VM hat viele Vorteile, die in verschiedenen Gebieten Anwendung finden können. Gebiete in denen *dynamische Code-Evolution* eingesetzt werden kann sind:

[19] Nicht: Spring oder ähnliche Frameworks

- Kritische Anwendungen
- Dynamische Sprachen
- Dynamische AOP[20] (aspektorientierte Programmierung)
- Debugging

Dabei hat jedes Gebiet seine eigenen spezifischen Anforderungen. So haben *kritische Anwendungen* das Problem, dass sie nicht einfach für ein Update heruntergefahren werden können. *Dynamische Code-Evolution* würde es ermöglichen ein Update durchzuführen, ohne dass die Anwendung angehalten werden muss. Hierbei sollte der Fokus auf Sicherheit und Korrektheit der Updates liegen.

Zudem werden große Anstrengungen unternommen, um *dynamische Sprachen* auf statisch typisierten VMs zum Laufen zu bringen. Da die *dynamische Code-Evolution* zur Standard-Funktionalität von *dynamischen Sprachen* gehört, würde es die Implementierung dieser wesentlich vereinfachen, wenn ein vergleichbarer Mechanismus in der VM vorhanden wäre. Dabei liegt der Fokus darauf, dass kleine Änderungen schnell umgesetzt werden können.

Auch für die *aspektorientierte Programmierung* gibt es bereits einige dynamische Tools, die durch die Möglichkeiten der Java *HotSpot*^{TM}-VM eingeschränkt werden. Auch diese würden von einer erweiterten JVM profitieren.

Das *Debugging* als letztes Anwendungsgebiet wurde bereits ausführlich als Motivation für die anderen Tools behandelt. Deshalb soll an dieser Stelle lediglich besonderer Wert auf die speziellen Anforderungen beim *Debugging* gelegt werden. Ziel beim *Debugging* ist es die Produktivität zu erhöhen, indem die langen Wartezeiten beim wiederholten Start des Anwendungsservers minimiert werden. Die Hauptanforderungen sind dabei, dass die *dynamische Code-Evolution* **jederzeit** durchgeführt werden kann und dass der Entwickler beim Programmieren **keine zusätzlichen Arbeiten**[21] verrichten muss.

Da jedes Gebiet seine ganz eigenen Anforderungen mit sich bringt, haben sich die Entwickler der *DCEVM* dazu entschieden, den Fokus auf nur eines der Anwendungsgebiete zu legen, das *Debugging*. Somit ist die Motivation, die hinter der Entwicklung der *DCEVM* steht die gleiche, wie bei den anderen Tools. In der folgenden Tabel-

[20] Programmierparadigma für die objektorientierte Programmierung mit dem Zweck generische Funktionalitäten über mehrere Klassen hinweg zu verwenden (z. B. Logging)

[21] Gemeint ist zum Beispiel das Einrichten spezieller Update-Punkte an denen die Code-Evolution durchgeführt werden kann

le (Tabelle 5) findet sich eine Übersicht über die durch die *DCEVM* ermöglichten Code-Änderungen.

Zur Laufzeit mögliche Änderungen	**DCEVM**
Änderungen im Körper von Methoden	✓
Hinzufügen von Methoden	✓
Entfernen von Methoden	✓
Hinzufügen von Feldern	✓
Entfernen von Feldern	✓
Hinzufügen von Klassen	✓
Entfernen von Klassen	✓
Hinzufügen von Superklassen	✓
Entfernen von Superklassen	✓/**X**

Tabelle 5: Übersicht über die zur Laufzeit möglichen Änderungen durch die *DCEVM*

Wie *JRebel* und *Javeleon* unterstützt auch die *DCEVM* die wichtigsten Änderungen, die im Alltag eines Entwicklers anfallen. Alle Änderungen an Klassen können prinzipiell durch die *DCEVM* umgesetzt werden. Allerdings hat die *DCEVM* eine kleine Beschränkung. Da die Programmiersprache *Java* und somit auch die Java $HotSpot^{TM}$ -VM ständig weiterentwickelt werden, muss auch die *DCEVM* als Weiterentwicklung der Java $HotSpot^{TM}$-VM ständig angepasst werden. Bei den meisten Änderungen[22] ist das kein großes Problem. Releases einer leichtgewichtigen *DCEVM* erscheinen zeitnah zu den letzten Java-Updates. Das Entfernen von Superklassen macht die Anpassung allerdings um einiges schwieriger. Aufgrund der Komplexität der Änderungen gibt es aktuell noch keine Vollversion der *DCEVM*, die Java 8 unterstützt. Dementsprechend ist das Entfernen von Superklassen nur bis Java 7 möglich.

Die vier letzten der in Abschnitt 2.3.2 aufgezählten Kriterien für *Hot Deployment* Tools setzt auch die *DCEVM* um. Alle Zustände der Anwendung werden während der Updates bewahrt und die Anwendung kann stets, wie geplant, fortgesetzt werden. Auch sichtbare Effekte auf das Verhalten der Anwendung, wie zum Beispiel unvorhergesehene Fehler, gibt es nicht. Wie ausführlich in Quelle [TW11] - Kapitel 7 - erläutert, ist der Effekt auf die Performance der Anwendung sehr gering und auch der Effekt auf die Speichernutzung ist nicht signifikant.

[22] Allen außer dem Entfernen von Superklassen

3 Konzepte

Nachdem in Kapitel 2 die verschiedenen Tools untersucht wurden, soll in diesem Kapitel die *DCEVM* als potenziell vielversprechendstes Tool genauer betrachtet und ihre Konzepte erläutert werden, bevor schließlich in Abschnitt 4.1 eine endgültige Entscheidung getroffen und begründet wird. Dazu werden in Abschnitt 3.1 einige wichtige Eigenschaften von Systemen, die *Hot Deployment* umsetzen, genannt und es wird untersucht, inwieweit die *DCEVM* diese Eigenschaften besitzt. In Abschnitt 3.2 werden anschließend die technischen Spezifikationen bzw. Konzepte näher erläutert. Zu diesem Zweck werden zuerst in Abschnitt 3.2.1 die verschiedenen Ebenen (Level) der Code-Evolution und die Probleme, die mit diesen einhergehen, beschrieben. Anschließend wird in Abschnitt 3.2.2 erklärt, was bei der Implementierung der *DCEVM* und zur Lösung der Probleme beachtet werden musste. Abschnitt 3.3 beschäftigt sich nun mit der Integration der *DCEVM* in die Anwendungsentwicklung.

Trotz dass in diesem Kapitel speziell die *DCEVM* untersucht wird, welche eine Modifikation der Java $HotSpot^{TM}$-VM ist, beschränken sich die dargestellten Konzepte nicht auf diese. Sie können auf jede VM übertragen werden, die statisch typisierte, objektorientierte Programme ausführt. Der Grund dafür, dass sich die *DCEVM* auf die Java $HotSpot^{TM}$-VM begrenzt, liegt darin, dass sie eine hochperformante VM speziell für die Softwareentwicklung (*Debugging*) ist und sich die *DCEVM* auf ebendiese konzentriert.

3.1 Anforderungen an ein Hot Deployment Tool

In diesem Abschnitt sollen einige Eigenschaften herausgearbeitet werden, die ein *Hot Deployment* Tool besitzen sollte, um von den Softwareentwicklern angenommen zu werden (vgl. [ARG11a], [TW11]). Vorausgreifend sei gesagt, dass die *DCEVM* jede der genannten Eigenschaften besitzt. Im Folgenden werden die Eigenschaften aufgezählt, erläutert und es wird Bezug auf ihre Wichtigkeit genommen.

- Unbeschränkt
- Atomar
- Geräuschlos
- Transparent
- Bereit

- Verständlich
- Leicht nutzbar

Unbeschränkt bezieht sich auf die durch das Tool mögliche Code-Evolution. Änderungen, die auf jeden Fall unterstützt werden sollten, sind Änderungen an der Menge von deklarierten Methoden (siehe Abschnitt 3.2.1), dem Layout von Objekten[23] und der Klassenhierarchie[24]. Insbesondere die korrekte Verbreitung von Änderungen an die Unterklassen ist wichtig.

Atomar bedeutet, dass die Aktualisierung einer Menge von Klassen mit einem Mal ausgeführt wird. Die Wichtigkeit dieser Eigenschaft ergibt sich schlichtweg aus dem Sachverhalt, dass die Änderungen zweier Klassen voneinander abhängig sein können, sodass es keinen Weg gibt, ein korrektes Programm durch sequenzielle Anwendung der Änderungen zu erhalten. Zudem sollte das Tool vor der Code-Evolution eine Sicherung anlegen, zu der es zurückkehren kann, sollten Teile der Änderung ungültig sein. Es sollten stets alle oder keine der Änderungen übertragen werden.

Geräuschlos beschreibt das Verhalten der VM bei der normalen Ausführung der Anwendungen. Dass bei der Code-Evolution selbst Zeit verloren geht, ist unvermeidbar, doch vor und nach der Aktualisierung sollte die Anwendung keinen merklichen Geschwindigkeitsverlust hinnehmen müssen. Schließlich würde ein solches Defizit die Einführung des Tools zur Softwareentwicklung deutlich erschweren.

Transparent bezieht sich auf den Vorgang des *Debugging* selbst. So sollten zum Beispiel keine Umwege im Java-*Stacktrace* zu sehen sein. Optimalerweise sollte der *Debugger* nicht mitbekommen, ob er noch die Originalversion der Anwendung ausführt oder ob bereits eine aktualisierte Version läuft. Wichtig ist, dass die Nutzererfahrung beim *Debugging* in keiner Weise negativ beeinflusst wird.

Bereit Änderungen umzusetzen ist die VM immer dann, wenn die Java-*Threads* unterbrochen werden können. Von besonderer Wichtigkeit ist dabei, dass die Möglichkeiten der Code-Evolution nicht durch aktuell aktive Methoden oder lebendige *Heap*-Objekte restringiert werden. Dementsprechend soll der Entwickler, wenn er die Ausführung einer Anwendung durch einen Breakpoint unterbricht, die Möglichkeit besitzen eine Änderung anzuwenden und das Ergebnis unmittelbar in Kraft treten zu sehen.

[23] Ändert sich die Struktur einer Klasse, müssen auch die bereits existierenden Objekte dieser aktualisiert werden

[24] Änderungen in der Klassenhierarchie lösen in der Regel eine Kette weiterer Änderungen aus

Verständlich für den Entwickler sollte die Semantik hinter den Änderungen sein. Wichtig ist dabei, dass der Entwickler weiß, welche Werte die Objekte einer aktualisierten Klasse besitzen. Die Strategie beim Initialisieren neuer Felder sollte stets klar sein. Zudem steht es dem Entwickler frei eigene Methoden zum Transformieren der Objekte zu benutzen.

Leicht nutzbar befasst sich mit der Nutzerfreundlichkeit des Tools. Dem Nutzer ist es wichtig, dass er seinen Entwicklungsvorgang möglichst wenig an das neue Tool anpassen muss. Je einfacher ein System zu nutzen ist, desto einfacher fällt die Umstellung und desto mehr Entwickler werden es im Endeffekt verwenden. Idealerweise würden die Updates geschehen, ohne dass der Nutzer überhaupt erst mit dem System interagieren muss.

3.2 Technische Spezifikationen

In diesem Abschnitt soll die Technologie, die hinter der *DCEVM* steckt, näher beleuchtet werden. Zu diesem Zweck werden verschiedene allgemeine Herausforderungen und Probleme angesprochen, mit denen sich ein *Hot Deployment* Tool auseinandersetzen muss, und es werden Konzepte und Lösungen vorgestellt um diese zu bewältigen. Doch auch wenn die geschilderten Lösungen nur Möglichkeiten darstellen, die speziell auf den Ansatz der *DCEVM* ausgerichtet sind, müssen die beschriebenen Herausforderungen und Probleme auch von jedem anderen Tool angegangen werden, welches die dynamische Code-Evolution mit Java umsetzen möchte.

3.2.1 Ebenen der Code-Evolution

3.2.1.1 Übersicht

In Abschnitt 2.1 wurde das Stichwort *Hot Deployment* als eine Idee definiert, deren Umsetzung es ermöglicht Code-Änderungen in einer Anwendung zur Laufzeit auf den Server zu übertragen. Das ist zwar richtig, beschreibt allerdings nicht einmal im Ansatz die Komplexität eines solchen Verfahrens. In Abschnitt 2.3 konnte man sehen, dass einige Änderungen von weniger Tools unterstützt werden als andere und somit offenbar schwerer umzusetzen sind. In diesem Abschnitt sollen nun die verschiedenen Code-Änderungen anhand der Komplexität ihrer Implementierung in Ebenen eingeteilt werden. Dabei umfasst eine Ebene, die für genau eine Code-Änderung steht, alle Modifikationen der darüberliegenden Ebenen plus weitere Modifikation, die durch die spezielle Code-Änderung nötig werden, wie in Abbildung 3 zu sehen ist. Das heißt, soll die unterste Ebene von einem Tool abgedeckt werden, müssen

zuerst alle anderen Ebenen umgesetzt worden sein. Quellen für diesen Abschnitt sind der Konferenzbericht [TW10] und die Diplomarbeit [TW11].

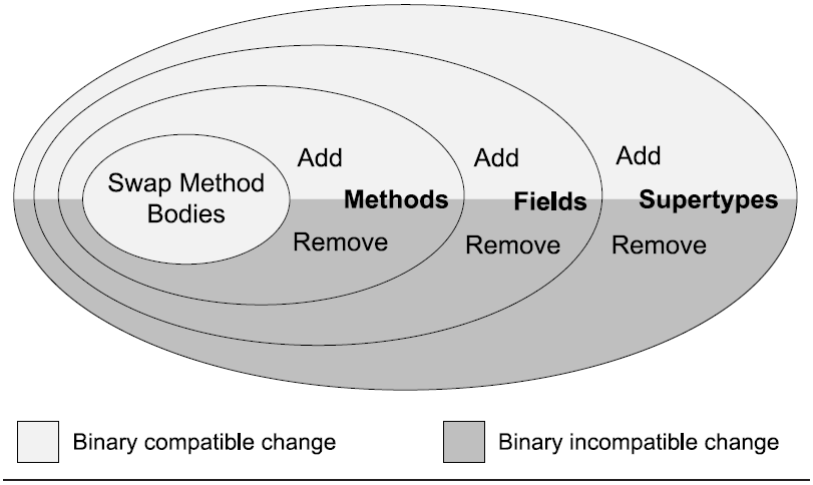

Abbildung 3: Ebenen der Code-Evolution ([TW10])

Abbildung 3 zeigt eine Übersicht über die wichtigsten Ebenen der Code-Evolution. Dabei werden die möglichen Code-Änderungen in vier Ebenen unterteilt. Die oberste Ebene umfasst Änderungen im Körper von Methoden. Darauf folgen das Hinzufügen und Entfernen von Methoden, Feldern und Superklassen - in dieser Reihenfolge. Zusätzlich werden die dargestellten Änderungen anhand ihrer *binären Kompatibilität* unterschieden. Das Hinzufügen von Methoden, Feldern und Superklassen ist *binär kompatibel* und das Entfernen ebendieser ist *binär inkompatibel*.

3.2.1.2 Ebenen

Änderungen im Körper von Methoden sind als oberste Ebene die am leichtesten umzusetzenden Code-Änderungen. Da kein anderer Bytecode von der genauen Implementierung der Methoden abhängig ist, können diese Änderungen isoliert durchgeführt werden, sodass der Rest des Systems keinen negativen Einflüssen unterliegt. Da diese Änderungen bereits durch die *HotSwap*-Technologie umgesetzt werden[25], bauen *Hot Deployment* Tools in der Regel auf der vorhandenen Technologie auf und benutzen sie als oberste Ebene in ihrer eigenen Technologie. Diese wird dann auf weiteren Ebenen um komplexere Code-Änderungen erweitert, deren Durchführung Effekte auf das Verhalten des Systems hat.

Um das **Hinzufügen und Entfernen von Methoden** umzusetzen, muss der Bytecode modifiziert werden können, wie es die Ebene *Änderungen im Körper von*

[25] Welche unter anderem auch in die $HotSpot^{TM}$-VM integriert wurde

Methoden bereits ermöglicht. Allerdings treten bei dieser Code-Änderung weitere Probleme auf, die durch das Tool behoben werden müssen. Da eine JVM in der Regel für jede Klasse eine Datenstruktur anlegt, die eine *virtuelle Methoden-Tabelle*[26] und eine *Interface-Methoden-Tabelle*[27] enthält, bewirken Änderungen an der Menge der Methoden stets auch Änderungen innerhalb der Tabellen. Diese können sowohl die Einträge der Tabellen als auch deren Größe betreffen. Zudem können die Änderungen auch die Methoden-Tabellen der Unterklassen betreffen. Ein weiteres Problem ergibt sich dadurch, dass sich die Indizes der *virtuellen Methoden-Tabellen* ändern können, was wiederum Maschinencode mit fester Codierung ungültig macht. Auch Maschinencode, der statische Verlinkungen zu Methoden enthält, muss entkräftet oder neu berechnet werden.

Das **Hinzufügen und Entfernen von Feldern** ist die erste Ebene, in der Code-Änderungen nicht mehr lediglich die Metadaten der VM betreffen. Da Felder in den Objekten selbst gespeichert werden[28], müssen diese auch modifiziert werden, wenn die Änderungen ihre Klasse oder eine ihrer Superklassen betreffen. Das heißt, das alte Objekt muss zu einer neuen Version konvertiert werden, die nun andere Felder mit abweichender Größe enthalten kann. Schwierig wird die Konvertierung dann, wenn sich die Größe des Objekts erhöht[29], denn dadurch reicht der dem Objekt zugewiesene Speicherblock nicht mehr aus. Die *DCEVM* löst das Problem durch die Implementierung einer modifizierten *Garbage Collection* (vgl. [TW11] - Abschnitt 3.5). Zudem werden, vergleichbar mit den Indizes der *virtuellen Methoden-Tabellen*, *Offset*-Werte im Interpreter und im kompilierten Maschinencode genutzt. Diese müssen angepasst oder entkräftet werden. Eine letzte Herausforderung dieser Ebene ist die Initialisierung von Feldern. Diese werden in der *DCEVM* mit *0*, *null* oder *false* initialisiert (vgl. [TW11] - Abschnitt 3.4).

Das **Hinzufügen und Entfernen von Superklassen** ist die komplexeste Änderung bei der dynamischen Code-Evolution. Für eine Klasse kann diese sowohl Änderungen ihrer Methoden als auch ihrer Felder bedeuten. Des Weiteren müssen die Metadaten der Klasse modifiziert werden um die neue Klassenhierarchie zu reflektieren.

[26] Eine Tabelle die das Konzept der Vererbung umsetzt. Enthält einen Eintrag für jede Methode, die von Unterklassen überschrieben werden kann. Die Tabelle der Unterklasse startet als Kopie der Tabelle der Superklasse. Anschließend werden bei überschriebenen Methoden die entsprechenden Einträge geändert.

[27] Enthält pro implementiertem Interface eine Untertabelle mit einem Eintrag pro Methode

[28] Genau genommen bekommt jedes Objekt beim Erzeugen einen Speicherbereich zugeteilt, in dem es zusammen mit allen Feldern abgelegt wird. Auf diese kann durch *Offset*-Werte zugegriffen werden

[29] Zum Beispiel durch Hinzufügen eines Feldes

Die **binäre Kompatibilität** zwischen zwei Programm-Versionen ist ein weiterer Aspekt in dem sich Code-Änderungen unterscheiden, wie in Abbildung 3 zu sehen ist. Man bezeichnet eine Code-Änderung als *binär kompatibel*, wenn die Gültigkeit des ursprünglichen Codes nicht durch die Änderung beeinträchtigt wird. Dabei kann der Schritt der Code-Evolution jederzeit durchgeführt werden, auch wenn der entsprechende *Thread* inmitten der Ausführung ist. Dementsprechend kann der ursprüngliche Code auch nach dem Update noch ausgeführt werden.

Binär inkompatible Code-Änderungen dagegen können den ursprünglichen Code ruinieren und dessen weitere Ausführung unmöglich machen. Das Problem ist, dass weder die Spezifikationen der Sprache Java noch die JVM Spezifikationen die Möglichkeit der Code-Evolution berücksichtigen und sie daher von Annahmen ausgehen, welche nach der Code-Evolution nicht länger korrekt sind. Binär inkompatible Code-Änderungen können in zwei Kategorien unterteilt werden:

- Entfernen von Methoden oder Feldern
- Entfernen von Superklassen

Beim **Entfernen von Methoden oder Feldern** kann ein Problem bei der Referenzierung auftreten. Schließlich kann der Bytecode von bereits gelöschten oder ersetzten Methoden immer noch Referenzen zu Elementen der Klasse enthalten, die in der neuen Version des Codes nicht länger existieren. Wenn die VM bei der Ausführung eine solche Stelle im Bytecode erreicht, muss sie entscheiden, wie bei dem Aufruf von gelöschten Methoden oder dem Zugriff auf gelöschte Felder verfahren werden soll.

Das **Entfernen von Superklassen** kann eine wichtige Invariante für die Ausführung eines Java-Programmes verletzen, da der statische und der dynamische Typ einer Variablen möglicherweise nicht länger eine Beziehung zwischen Untertypen[30] besitzen. Zudem kann das Objekt, das einen dynamischen Aufruf empfängt, nicht länger kompatibel mit der Klasse der aufrufenden Methode sein.

3.2.2 Implementierung

Ziel dieses Abschnittes soll es sein eine Schritt für Schritt Übersicht über die Funktionsweise des implementierten Algorithmus zu geben und **nicht** diesen vollständig zu erläutern, da dies den Rahmen der Masterarbeit sprengen würde. Daher werden die Modifikationen des *Garbage Collectors* und die *State Invalidation*[31] nur kurz

[30] engl.: *subtype relationship*
[31] Außerkraftsetzung von Zuständen

behandelt und auch binär inkompatible Code-Änderungen werden außen vor gelassen. Eine detaillierte Ausführung der Implementierung findet sich in den Arbeiten [TW10] und [TW11].

Wie bereits erwähnt, wurde die *DCEVM* als eine Modifikation der Java $HotSpot^{TM}$-VM entwickelt, einer hochperformanten VM die einen Interpreter und zwei *just-in-time* Compiler enthält. Ziel ist es den vorhandenen *HotSwap*-Mechanismus um weitere Änderungen zu erweitern und dabei die Modifikationen der VM möglichst gering zu halten. Diese betreffen weder die *just-in-time* Compiler noch den Interpreter. Betroffen sind der *Garbage Collector*, das *Systemverzeichnis*[32] und die *Klassen-Metadaten*. Dabei wird die VM in ihrer eigentlichen Funktionsweise nicht beeinträchtigt.

3.2.2.1 Übersicht

Eine Übersicht über den implementierten Algorithmus findet sich in Abbildung 4.

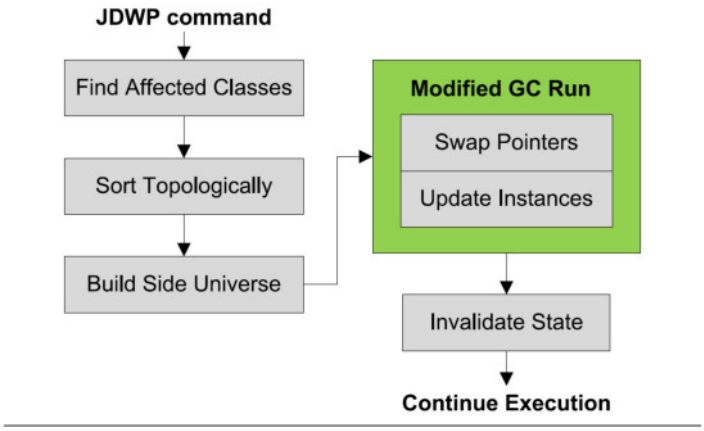

Abbildung 4: Übersicht über die Schritte des Code-Evolution Algorithmus ([TW10])

Ausgelöst wird dieser durch das JDWP (Java Debug Wire Protocol) Kommando. In den folgenden drei Schritten wird nun besonderer Wert auf die Klassenhierarchie und die durch sie anfallenden Herausforderungen gelegt. So müssen im ersten Schritt die von der Code-Evolution betroffenen Klassen gefunden[33] und anschließend im zweiten Schritt, anhand ihrer neuen Hierarchie, sortiert werden. In Schritt drei werden die Klassen geladen und zum *Typ-Universum* der VM hinzugefügt. Dadurch entsteht ein *Neben-Universum*, das die aktuelle Klassenhierarchie enthält. Schließlich wird

[32] engl.: *system dictionary*
[33] Zusätzlich zu den geänderten Klassen müssen gegebenenfalls ihre Unterklassen aktualisiert werden

das eigentliche Update durch eine vollständige *Garbage Collection* durchgeführt. Nachdem im letzten Schritt diejenigen Zustände entkräftet[34] wurden, die durch die neuen Klassen nicht länger konsistent sind, kann die Ausführung des Programmes fortgesetzt werden.

Das **JDWP Kommando** ist eine Spezifikation für das Interface zwischen einer Java-VM und einem Java-*Debugger*. Daher kann die *DCEVM* von jeder Java-Entwicklungsumgebung genutzt werden, die das JDWP Protokoll zum *Debugging* verwendet. Dazu zählen unter anderem *NetBeans* und Eclipse. Damit das JDWP Kommando ausgelöst wird, ist es notwendig, dass alle geänderten Klassen bereits zuvor in die VM geladen wurden. Allerdings ist für eine bisher nicht geladene Klasse auch kein Update notwendig, da sie als Anfangsversion in die VM geladen werden kann. Ein entsprechendes Kommando wird durch die *DCEVM* implementiert.

3.2.2.2 Herausforderung - Klassenhierarchie

Die in diesem Abschnitt beschriebenen Schritte können parallel zur gewöhnlichen Programmausführung durchgeführt werden. Erst bei der *Garbage Collection* ist dies nicht länger möglich.

Das **Finden von betroffenen Klassen** wird bei Änderungen notwendig, die über den Austausch des Körpers einer Methode hinausgehen, da diese auch die entsprechenden Unterklassen betreffen können. Wird einer Klasse ein neues Feld hinzugefügt, so erhalten auch alle ihre Unterklassen dieses Feld. Änderungen an der Menge von Methoden können gegebenenfalls die *virtuellen Methoden-Tabellen* der Unterklassen beeinflussen. Daher erweitert der implementierte Algorithmus die Menge der betroffenen Klassen stets um alle ihre Unterklassen. Ein Beispiel zeigt Abbildung 5.

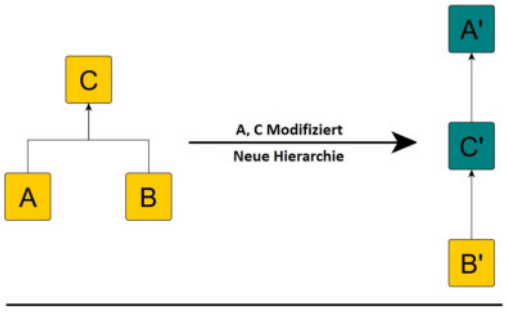

Abbildung 5: Beispiel für eine Code-Evolution inkl. Änderung der Klassenhierarchie (vgl. [TW10])

[34] engl.: invalidate state

Im Beispiel werden die Klassen A und C modifiziert. Da Klasse B allerdings eine Unterklasse von C ist, zählt auch sie zu den betroffenen Klassen und muss durch B' ersetzt werden. Diese basiert zwar auf dem gleichen Code wie B, erbt allerdings unterschiedliche Eigenschaften und muss deshalb ebenfalls aktualisiert werden. Das gleiche Prinzip wird auch bei Modifikationen von Interfaces umgesetzt.

Eine **topologische Sortierung** muss der Algorithmus selbst durchführen, da das JDWP Kommando keine Ordnung festlegt, in der die Klassen aktualisiert werden sollen. Diese basiert auf der Klassenhierarchie, denn eine Klasse[35] muss stets vor ihren Unterklassen aktualisiert werden. Schließlich kann eine Unterklasse inkompatibel mit der ursprünglichen Version ihrer Oberklasse sein. Zudem muss die Sortierung anhand der neuen Klassenhierarchie[36] erfolgen. Da der VM allerdings erst nach dem Laden der Klasse Informationen über die Klassenhierarchie zur Verfügung stehen, müssen Teile der Klassen vorher *geparst* werden. Die entsprechende *topologische Sortierung* für das Beispiel aus Abbildung 5 wäre $A' \rightarrow C' \rightarrow B'$.

Die Idee hinter dem **Aufbau eines Neben-Universums** basiert auf dem Wunsch, veralteten Code, der von den Eigenschaften der ursprünglichen Klassen abhängt, weiterhin ausführen zu können. Zudem ist es die einzige Möglichkeit mit zyklischen Abhängigkeiten zwischen Änderungen[37] umzugehen, da sich das *Typ-Universum* stets in einem konsistenten Zustand befinden muss. Durch den *Aufbau eines Neben-Universums* kann die veraltete Klasse nicht den Ladevorgang einer neuen Klasse beeinflussen. Sofort nach Abschluss des Ladevorgangs einer Klasse werden auch die Einträge im *Systemverzeichnis* der Java $HotSpot^{TM}$-VM aktualisiert. Dadurch wird, wenn die Klasse C' im Beispiel von Abbildung 5 geladen werden soll, die Klasse A' beim *Lookup* nach der Oberklasse ausgegeben, da diese zuvor aktualisiert wurde. Allerdings werden die neu geladenen Klassen A', B' und C' mit ihren alten Versionen A, B und C jeweils beidseitig verknüpft, sodass bei der im nächsten Schritt folgenden *Garbage Collection* weiterhin auf die alten Versionen zugegriffen werden kann.

3.2.2.3 Garbage Collector & State Invalidation

Der Hauptteil des Algorithmus wurde als eine Modifikation des *mark-and-compact Garbage Collection* Algorithmus implementiert. Dabei funktioniert jeder *Garbage*

[35] Interfaces eingeschlossen
[36] Die Klassenhierachie nach der Code-Evolution
[37] Eine Code-Änderung setzt voraus, dass A vor B aktualisiert werden muss und eine andere, dass B vor A aktualisiert werden muss

Collection Algorithmus nach einem einfachen Prinzip. Alle lebendigen Objekte werden in einen neuen, fortlaufenden Speicher kopiert[38] und anschließend wird der alte Speicher weggeworfen. Dadurch werden die unbenutzten Objekte gelöscht und die Lücken zwischen den lebendigen Objekten geschlossen.

Der implementierte Algorithmus funktioniert nach dem gleichen Prinzip. Hierbei wird im ersten Schritt für jedes lebendige Objekt im *Heap* ein *Vorwärtszeiger* berechnet, welcher auf die Adresse des Objekts nach der Verdichtung des Heaps zeigt[39]. In dem im zweiten Schritt folgenden *Heap*-Durchlauf werden nun alle Referenzen[40] so angepasst, dass sie auf die neu berechnete Adresse zeigen. In diesem Zusammenhang sorgt der modifizierte Algorithmus dafür, dass die Zeiger auf die alte Klasse durch Zeiger auf die neue Klasse ersetzt werden. In der abschließenden *Verdichtungsphase* werden die Objekte zu ihren neuen Adressen verschoben.

Für das Aktualisieren der Instanzen selbst muss eine Strategie zur Initialisierung ihrer Felder umgesetzt werden. Dabei müssen sowohl gleichbleibende Felder als auch neue Felder initialisiert werden. Der implementierte Algorithmus vergleicht zu diesem Zweck die Felder anhand ihrer Namen und ihres Typs und initialisiert gleichbleibende Felder mit ihren alten Werten. Neue Felder werden mit *0*, *null* oder *false* initialisiert. Detaillierte Informationen über das Aktualisieren der Instanzen finden sich in Quelle [TW10] - Abschnitt 3.5.2.

State Invalidation wird notwendig, da der Prozess der Code-Evolution einige Invarianten in der VM verletzt. Bei der Entwicklung der Java $HotSpot^{TM}$-VM war die Möglichkeit zum Aktualisieren von Java-Klassen zur Laufzeit noch nicht publik, weshalb Annahmen getroffen wurden, die durch die Code-Evolution nicht länger gültig sind. Die Invarianten betreffen folgende Bereiche:

- Kompilierter Code
- Konstantenpool Cache
- Werte der Klassen-Zeiger

Kompilierter Code bezieht sich auf Maschinencode, welchen der *just-in-time* Compiler vor der Code-Evolution generierte (vgl. [TW10] - Abschnitt 3.6.1). Folglich muss dieser auf Gültigkeit überprüft werden. Zu den nach der Code-Evolution häufig

[38] Diese Phase nennt man auch *Verdichtung*

[39] Man bemerke, dass die Größe der Objekte an dieser Stelle bereits aus den neuen Klassen berechnet werden muss, da diese in einem fortlaufenden Speicherbereich abgelegt werden sollen

[40] Beinhaltet Zeiger-Felder und Klassenzeiger

fehlerhaften Informationen zählen die Indizes der *virtuellen Methoden-Tabellen*, *Offset*-Werte von Feldern und Annahmen über die Klassenhierarchie und deren Aufrufe.

Der **Konstantenpool Cache** ist ein *gecachtes*, modifiziertes Abbild des Konstantenpools. Die Java $HotSpot^{TM}$-VM benutzt ihn zur Erhöhung der Geschwindigkeit des Interpreters. Während die ursprünglichen Einträge lediglich symbolische Referenzen sind, beinhaltet der *Konstantenpool Cache* direkte Referenzen zu den Objekten der Metadaten. Von der Code-Evolution verletzt werden können Einträge über Felder (*Offset*-Werte) und Methoden.

Da einige Datenstrukturen der Java $HotSpot^{TM}$-VM von den genauen Adressen der Metadaten-Objekte der Klassen abhängen, muss sichergestellt werden, dass die entsprechenden Datenstrukturen neu initialisiert und die **Werte der Klassen-Zeiger** aktualisiert werden.

3.3 Möglichkeiten der Integration

Trotz der vielen Möglichkeiten, welche die *DCEVM* bietet, ist sie letztlich nur eine Erweiterung der Java $HotSpot^{TM}$-VM und noch kein vollständiges Programm. Sie ist vergleichbar mit einer Klasse, durch deren Nutzung man die Code-Evolution umsetzen kann. Da die *DCEVM* eine komplizierte und neuartige Technologie ist, wird sie noch nicht durch die verschiedenen IDEs unterstützt und zu ihrer Integration wird ein zusätzliches Programm benötigt. Dieses Programm muss Änderungen in den Klassen registrieren und anschließend den Code-Evolution Algorithmus der *DCEVM* über das JDWP Kommando auslösen.

Zu diesem Zweck wurde der *HotswapAgent* entwickelt. *HotswapAgent* ist, wie schon die *DCEVM*, ein *OpenSource* Projekt, mit dem Ziel *Hot Deployment* zu verwirklichen. In Zusammenarbeit schaffen sie es die Code-Evolution für den alltäglichen Entwicklungsprozess bereitzustellen. Um dies zu erreichen, verbindet sich der *HotswapAgent* mit zwei Komponenten - der IDE und der JVM des Anwendungsservers. Die Einbindung in die IDEs geschieht, wie schon beim Standard *HotSwap*-Mechanismus, über die *Debugger* API. Durch diese erfährt der *HotswapAgent* alles Nötige über die Code-Änderungen.

Die Integration in den Anwendungsserver funktioniert durch die *Javaagent* Technologie. Ein *Javaagent* ist im Prinzip ein JVM Plugin. Als speziell angefertigte *.jar* Datei nutzt er die *Instrumentation* API, welche die JVM zur Verfügung stellt (vgl. [OS]). Diese ermöglicht dem Agenten die Neudefinition des Inhalts von zur Laufzeit

geladenen Klassen. Von der JVM geladen werden *Javaagents* durch die *-javaagent* Kommandozeilen Option. Dies erfolgt zum Zeitpunkt des Serverstarts.

Zudem ist der *HotswapAgent* ein Plugin-Container[41] (vgl. [JB]). Er enthält einen Plugin-Manager, eine Plugin-*Registry* und eine Vielzahl von Agenten-*Services*. Zusätzlich zum Laden der Klassen sucht der Agent im Klassenpfad nach Klassen mit einer *@Plugin* Annotation, bindet Agenten-*Services* ein[42] und registriert Events bei denen Updates durchgeführt werden.

Die zahlreichen Plugins, die vom *HotswapAgent* verwaltet werden, sind in der Regel auf spezifische Frameworks ausgerichtet. Da beim Entwickeln von Anwendungen häufig Frameworks verwendet werden, bringt eine Unterstützung dieser beim *Hot Deployment* einen großen Nutzen mit sich. Zur Umsetzung lösen die Plugins erfahrungsgemäß ein Nachladen der entsprechenden Framework-Konfiguration aus. Hibernate, Logback, Seam, Spring und Weld sind nur einige der Frameworks die der *HotswapAgent* auf diese Weise partiell oder vollständig unterstützt.

[41] Bei Containern werden Anwendungen zusammen mit allem, was sie zum Funktionieren brauchen, in ein einzelnes Paket gebündelt. Auch Plugins können nach diesem Prinzip verwaltet werden

[42] engl.: inject

4 Umsetzung

Nachdem in Kapitel 2 verschiedene *Hot Deployment* Tools untersucht und anschließend in Kapitel 3 die *DCEVM* und ihre Konzepte erläutert wurden, beschäftigt sich das 4. Kapitel mit der Einrichtung des geeignetsten Tools auf den Servern der dotSource GmbH. In Abschnitt 4.1 werden die Tools miteinander verglichen und beurteilt, sodass die Frage nach dem vielversprechendsten Tool beantwortet werden kann. Abschnitt 4.2 setzt sich mit den Herausforderungen bei der Einrichtung des Tools auf einem Server auseinander und liefert eine Anleitung inklusive Erklärung der notwendigen Schritte. Anschließend sollen in Abschnitt 4.3 Funktionen erläutert werden, die über die grundlegende Funktionalität eines *Hot Deployment* Tools hinausgehen. In diesem Zusammenhang wird unter Abschnitt 4.3.2 ein eigenes Plugin vorgestellt, welches eine weitere Funktion hinzufügt.

4.1 Begründung für die Wahl des Tools

Obwohl im Rahmen der Masterarbeit auch *JRebel* untersucht wurde, lag der Fokus stets auf dem Finden eines geeigneten *OpenSource Hot Deployment* Tools. *JRebel* dient in dieser Arbeit, als das aktuell fortschrittlichste Tool, lediglich als Referenzwert, der eine bessere Evaluierung der anderen Tools ermöglichen soll. Tabelle 6 zeigt eine Übersicht über alle beschriebenen *Hot Deployment* Tools.

Änderung	**HotSwap**	**JRebel**	**Javeleon**	**Jvolve**	**DCEVM**
Körper von Methoden	✓	✓	✓	✓	✓
+ Methode	**X**	✓	✓	✓	✓
- Methode	**X**	✓	✓	✓	✓
+ Feld	**X**	✓	✓	✓	✓
- Feld	**X**	✓	✓	✓	✓
+ Klasse	**X**	✓	✓	✓	✓
- Klasse	**X**	✓	✓	✓	✓
+ Superklasse	**X**	✓	✓	**X**	✓
- Superklasse	**X**	✓	✓	**X**	✓/**X**

Tabelle 6: Vergleich der zur Laufzeit möglichen Änderungen

Da kein anderer Code von der genauen Implementierung der Methoden abhängt, sind Änderungen im Körper von Methoden vergleichsweise einfach umzusetzen, was bereits 2002 durch die *HotSwap*-Technologie geschah. Dies reicht allerdings nicht aus

um die Anzahl an *Redeploys* merklich zu verringern. Daher ist sie kein geeignetes Tool.

JRebel und *Javeleon* sind beides vielversprechende Tools und unterstützen die wichtigsten Änderungen, die ein Entwickler in seinem Alltag durchführt, allesamt. Da *JRebel* keine *OpenSource* Software ist und somit entfällt, scheint also *Javeleon* die beste Wahl zu sein. Es hat einen intelligenten Ansatz, bietet zahlreiche Funktionen und wird in Konferenzberichten ausführlich beschrieben (vgl. [ARG11a], [ARG11b], [ARG12]). Doch auch *Javeleon* ist kein *OpenSource* Projekt. Ziel von *Javeleon* war es stets die im Rahmen der Forschung entwickelte *Hot Deployment* Technologie zu kommerzialisieren. Zudem wurde *Javeleon*, als *JRebels* damals größter Konkurrent, 2013 von *ZeroTurnaround* aufgekauft. Die Technologie von *Javeleon* wurde in *JRebel* integriert und *Javeleons* Gründer *Allan Gregersen* und *Michael Rasmussen* sind dem *JRebel* Team beigetreten.

Jvolve und die *DCEVM* sind somit die letzten *Hot Deployment* Tools, die als *OpenSource* Alternative in Frage kommen. Beide verwenden den gleichen Ansatz - die Umsetzung von *Hot Deployment* durch eine modifizierte JVM. Doch wo *Jvolve* vorrangig für die Forschung entwickelt wurde und eine Modifikation der *JikesRVM*[43] ist, wurde die *DCEVM* speziell für das *Debugging* entwickelt. Sie ist eine Modifikation der hochperformanten, für die Softwareentwicklung bereitgestellten Java $HotSpot^{TM}$-VM.

Somit steht nach dem Ausschlussverfahren nur noch die *DCEVM*, in Kombination mit dem *HotswapAgent*, als *OpenSource Hot Deployment* Software für Java zur Verfügung. Doch ein Mangel an geeigneten Alternativen sagt noch nichts über die Software selbst aus. Trotz dass die *DCEVM* eine *OpenSource* Software ist, kann sie an einigen Punkten mit *JRebel* konkurrieren und besitzt auch Vorteile gegenüber *JRebel*.

In Tabelle 6 ist zu sehen, dass auch die *DCEVM* die wichtigsten Änderungen im Alltag eines Entwicklers teilweise oder vollständig unterstützt. Lediglich das Entfernen von Superklassen wird in Java 8 derzeit noch nicht unterstützt. Ein großer Vorteil gegenüber *JRebel* ist die ausführliche Dokumentation der *DCEVM*. Da sie im Rahmen einer Doktorarbeit entwickelt wurde, wurden ihre Ansätze, Konzepte und Funktionen wissenschaftlich geschildert und evaluiert. Dies ermöglicht ein fortgeschrittenes Verständnis der benutzten Software. Zudem entstand die *DCEVM* durch enge Zusammenarbeit mit Oracle und wird ständig weiterentwickelt und auf dem neuesten

[43] Dient als Testumgebung

Stand gehalten[44]. Als mögliche Lösung der *Hot Deployment* Problematik wurde bereits ein JEP[45] (JDK Enhancement Proposal) gestellt, um die Möglichkeiten der *DCEVM* endgültig in die Java *HotSpot*^{TM}-VM zu integrieren.

4.2 Einrichtung des Tools

Das Installieren von Software auf einem System ist eine Prozedur, die im Normalfall problemlos und schnell abgeschlossen werden kann. Selbst unerfahrene Menschen haben mit dem Installationsvorgang nur selten Probleme, da dieser meist nur durch eine *.exe* oder einen Kommandozeilenaufruf gestartet werden muss und anschließend selbsterklärend oder automatisiert ist. Doch mit der Komplexität des Systems steigt auch die Schwierigkeit der Installation. Wenn man zusätzlich keine Berechtigung hat, tief greifende Änderungen am System (z. B. Aktualisieren der Java-Version) vorzunehmen, kann die Herausforderung zu einem echten Problem heranwachsen, insbesondere wenn der Fehler nicht hinreichend eingegrenzt werden kann[46].

4.2.1 Im Minimalbeispiel

Das Einrichten der *DCEVM* als *Hot Deployment* Tool in einem Minimalbeispiel ist vergleichsweise unkompliziert. Zur Installation auf den häufig verwendeten Systemen gibt es Anleitungen und Programme, die den Vorgang ausführlich erklären und vereinfachen. Das verwendete Minimalbeispiel ist eine Spring MVC (Model[47] - View[48] - Controller[49]) Webanwendung. Sie wird auf einem, in die Eclipse Mars IDE integrierten, *Apache Tomcat* Server ausgeführt (Version 8.0.36). Dieser läuft auf einer JRE (Java Runtime Environment[50]) mit Version 8u92. Das gewählte Spring Framework hat die Version 4.1.7 und als Betriebssystem dient Windows 10. Das Minimalbeispiel hat den Zweck auftretende Fehler in einer kleinen und übersichtlichen Umgebung nachzuempfinden und die Suche nach den Ursachen zu erleichtern. Zum Nachbau der Anwendung kann das Tutorial „*Simplest Spring MVC Hello World Example/Tutorial*“ von der Internetseite `http://crunchify.com/` verwendet werden (Stand: 24.11.2016).

[44] Da die *DCEVM* auf der Java *HotSpot*^{TM}-VM basiert, muss sie bei Java-Änderungen angepasst werden
[45] Antrag auf Erweiterung des Java Development Kits
[46] Da die *DCEVM* und der *HotswapAgent* zusammen *Hot Deployment* umsetzen ist es häufig schwer festzustellen, in welcher Komponente der Fehler auftritt
[47] Datencontainer ohne Logikfunktionen
[48] Zeigt die Daten aus dem Model in einer graphischen Benutzeroberfläche an (in der Regel HTML-Ausgabe)
[49] Bearbeitet Benutzeranfragen, generiert ein geeignetes Model und reicht es an den View weiter
[50] dt.: Java-Laufzeitumgebung

Verglichen mit dem Aufsetzen des Minimalbeispiels, ist das Installieren der *DCEVM* und des *HotswapAgents* in der beschriebenen Umgebung eher unkompliziert. Dazu muss im ersten Schritt der *Release* der *DCEVM* heruntergeladen werden, welcher der installierten Java-Version entspricht[51]. Die *Releases* findet man auf der *GitHub* Seite des *DCEVM* Projekts (`https://github.com/dcevm/dcevm/releases` - Stand: 24.11.2016). Nachdem die *.jar* Datei heruntergeladen wurde, kann diese über die Konsole durch den Befehl *java -jar dateiname.jar*[52] aufgerufen werden. Es öffnet sich eine graphische Benutzeroberfläche, welche in Abbildung 6 zu sehen ist. Mit ihrer Hilfe kann man die *DCEVM* durch einen Klick installieren. Empfohlen wird die Installation als alternative JVM (*altjvm*) und es sollte die JRE/JDK gewählt werden, auf welcher der Server letztendlich laufen soll.

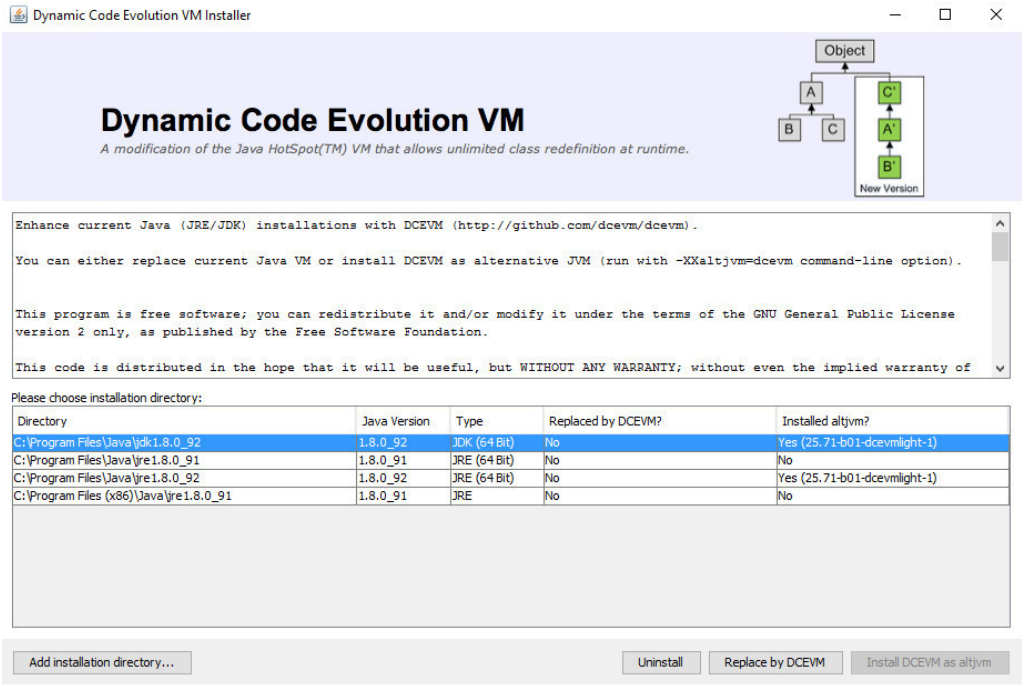

Abbildung 6: Graphische Benutzeroberfläche der *DCEVM*

Nachdem die *DCEVM* installiert wurde, muss sichergestellt werden, dass der Server mit der richtigen Laufzeitumgebung gestartet wird. Unter *Tomcat Server* → *Runtime Environment* → *JRE* ist die entsprechende Einstellung zu finden. Für genauere Anweisungen gibt es Guides im Internet[53].

[51] Optimalerweise stimmen beide Versionen überein. Ist dies nicht der Fall sollte der nächsthöhere *DCEVM Release* verwendet werden

[52] Zum Beispeil *java -jar DCEVM-light-8u92-installer.jar*

[53] Zum Beispiel `https://github.com/HotswapProjects/HotswapAgent/wiki/Eclipse-setup` - Stand 30.11.2016

Da die *DCEVM* jetzt einsatzbereit ist, muss nur noch der *HotswapAgent* eingerichtet werden um *Hot Deployment* zu ermöglichen. Dazu muss der letzte *Release* von der Projektseite (`https://github.com/HotswapProjects/HotswapAgent/releases` - Stand: 01.12.2016) als *.jar* Datei heruntergeladen werden und diese muss beim Serverstart geladen werden. Die Integration des *HotswapAgents* in den Server geschieht über die *Javaagent* Technologie, welche bereits in Abschnitt 3.3 untersucht wurde.

Zu diesem Zweck müssen im Minimalbeispiel die VM Argumente des *Apache Tomcat* Servers in Eclipse geändert werden. Unter *Tomcat Server* → *Open launch configuration* → *Arguments* → *VM arguments* können die entsprechenden Anpassungen vorgenommen werden. Durch *-XXaltjvm="dcevm"* wird die JVM instruiert die Bibliothek der alternativen JVM mit dem Namen *"dcevm"* zu nutzen und das Kommando *-javaagent="path-to-agent\hotswap-agent.jar"*[54] bewirkt das Laden des *HotswapAgents*. Diesem können auch noch weitere Optionen mitgegeben werden, wie der Pfad zur *hotswap-agent.properties* Datei, welche weitere Konfigurationen enthält, oder die Namen nicht zu nutzender Plugins. Dabei wird das erste [*Option*]=[*Wert*] Paar über ein Gleichheitszeichen an den Pfad zur *.jar* Datei angehängt und alle weiteren werden über ein Komma angefügt[55].

Zuletzt muss noch sichergestellt werden, dass die Serveroptionen *Modules auto reload by default* und *Automatically publish when resources change* aktiviert sind und dass der *Auto Reload* von Webmodulen im *Apache Tomcat* Server deaktiviert ist. Wird nun der Server im *Debug*-Modus gestartet, werden Änderungen zur Laufzeit auf den Server übertragen, sobald die jeweilige Klasse in Eclipse gespeichert wurde.

4.2.2 Auf dem hybris-Server

Bevor in Abschnitt 4.2.2.2 ein Weg beschrieben wird den *HotswapAgent* und die *DCEVM* auf dem *hybris*[56]-Server einzurichten, soll zunächst näher auf die Probleme beim Finden dieses Weges eingegangen werden. Die Schwierigkeit der Einrichtung lag nicht im Aufwand an sich, sondern in der Menge von Ungewissheiten und der Tatsache, dass die Fehlerquelle(n) nur schwer einzugrenzen war(en). Als mögliches Feedback gab es meist nur zwei Möglichkeiten, entweder das *Hot Deployment* hat funktioniert und beim Aufruf wurden die entsprechenden Änderungen angezeigt

[54] Achtung: Windows benutzt "\" für Pfadangaben, Linux benutzt "/"

[55] *-javaagent="path-to-agent\hotswap-agent.jar"=[Option1]=[Wert1],[Option2]=[Wert2],...*

[56] *hybris* ist ein deutsches Unternehmen, das intelligente Softwarelösungen im Bereich von *E-Commerce* und *PCM* (Product Content Management) anbietet. Die Software ist flexibel und lässt sich auf individuelle Bedürfnisse und Geschäftsabläufe anpassen

oder dies war nicht der Fall und der Server läuft noch mit dem alten Code. Bestenfalls wurde eine Meldung[57] ausgegeben, dass beim *Hot Deployment* ein Fehler aufgetreten ist und es wurde ein kurzer Fehlercode angezeigt, der im Zusammenhang meist keinen Sinn ergab oder nicht weiterhalf. Daher mussten alle potenziellen Fehlerquellen gleichzeitig untersucht werden. Dazu gehörten die Konfiguration des *HotswapAgents*, die Konfiguration von Eclipse Platform und Fehler beim Einrichten der *DCEVM*. Zudem stieg die Dauer eines Serverstarts durch die fehlerhafte Konfiguration stark an, sodass ein einzelner Serverstart nach Integration des Tools um die 30 Minuten dauerte, statt der ca. 5 Minuten vorher. Somit bedeutete jeder Versuch der Fehlerbehebung einen großen Zeitaufwand, was die Fehlersuche zu einem langsamen und nervenaufreibenden Prozess machte, der mehrere Wochen in Anspruch nahm.

4.2.2.1 Fehlersuche

Die erste Vermutung war, dass eine fehlerhafte bzw. unvollständige Konfiguration des *HotswapAgents* für das Versagen der Code-Evolution verantwortlich war. Speziell die Möglichkeit, dass der Agent die betroffenen Klassen nicht findet und somit keine Änderungen registriert, wurde in Betracht gezogen. Die Vermutung schien sich dadurch zu bestätigen, dass bei dem Tool *JRebel*, welches vorher auf dem System lief, in jedem Projekt der Pfad zu den *.class*-Dateien angegeben wurde. Dies geschah in einer Datei *rebel.xml* im Ordner *resources*. Ein entsprechender Nachbau wurde durch die *hotswap-agent.properties* Datei versucht. Doch egal in welchem Ordner die Datei stand, sie wurde nicht vom *HotswapAgent* erkannt. Nach einiger Suche im Quellcode wurde eine Möglichkeit entdeckt, den Pfad zur *hotswap-agent.properties* Datei direkt beim Einbinden des Agenten als *-javaagent* mit anzugeben[58]. Die Konfigurationen[59] wurden nun erkannt und umgesetzt, doch auch nach Angabe der Pfade für die *.class*-Dateien wurde die Code-Evolution nicht erfolgreich durchgeführt.

Noch vor der Fehlersuche beim *HotswapAgent* wurde die Konfiguration von Eclipse Platform durchgeführt. Da im Minimalbeispiel der *Apache Tomcat* Server direkt in Eclipse integriert war, war auch der *HotswapAgent* bereits mit Eclipse verbunden. Dies ist beim *hybris*-Server und Eclipse Platform nicht der Fall. Somit müssen der Server und der *Debugger* von Eclipse auf eine andere Art und Weise miteinander

[57] Zum Beispiel *Timeout!* oder *Änderungen der Klassenhierarchie nicht implementiert!*, während lediglich Methoden einer Klasse geändert wurden
[58] Die Möglichkeit wurde nicht dokumentiert, doch im Code wurde sie abgefangen
[59] Zum Beispiel Änderung des *Logger*-Levels

verbunden werden. Nach einiger Recherche fand sich die Möglichkeit einen *JPDA*[60] *Debugger* Server zu starten, der mit Eclipse Platform verbunden werden kann. Dies geschieht durch das *-agentlib:jdwp*[61] Kommando als VM Argument. Über den spezifizierten Port kann durch den Eclipse *Debugger* eine lokale Verbindung aufgebaut werden. Auch eine fehlerhafte Konfiguration der Verbindung wurde als mögliche Fehlerquelle in Betracht gezogen.

Zudem musste auch der Möglichkeit, dass der *HotswapAgent* nicht ausgereift genug für solch ein komplexes System ist, Beachtung geschenkt werden. Zusätzlich entsprach die Java-Version des Servers (Java 8u77) nicht der Java-Version des *DCEVM Releases* (Java 8u92). Dies stellte eine weitere Ungewissheit dar, da es stets empfohlen wird diese gleich zu halten, die Java-Version des Servers in der Firma aber nicht ohne weiteres geändert werden kann. Auch wenn beides falsche Fährten waren, mussten sie doch recherchiert und untersucht werden.

Die letzte potenzielle Fehlerquelle war die Installation der *DCEVM*. Da der *hybris*-Server der dotSource GmbH in einem *Docker*[62] Container ausgeführt wird und in diesem keine graphischen Oberflächen vorgesehen sind, konnte die *DCEVM* nicht auf die gleiche Art und Weise installiert werden wie im Minimalbeispiel. Es wurde ein *gradle* Script genutzt, welches ebenfalls von den Entwicklern der *DCEVM* zur Installation bereitgestellt wurde. Doch obwohl die *DCEVM* als alternative JVM eingerichtet wurde[63] und der normale Betrieb der Anwendung noch funktionierte (wenn auch sehr langsam), muss die Installation fehlerhaft oder unvollständig abgelaufen sein. Die Lösung des Problems schaffte eine zufällige Entdeckung bei der Recherche. Durch die Information, dass bei der Installation lediglich ein neuer Ordner angelegt und eine Datei aus der *.jar* Datei in die JRE kopiert wird, konnte die Installation auch manuell im *Docker* Container durchgeführt werden. Dies ermöglichte schließlich den reibungslosen Ablauf der Code-Evolution.

4.2.2.2 Lösungsweg

Der hier vorgestellte Lösungsweg bezieht sich speziell auf einen in der dotSource GmbH gängigen Systemaufbau. Der *hybris*-Server wird in einem *Docker* Container ausgeführt und als Betriebssystem dient *Debian 8* (Linux 3.16.0-4-amd64). Als IDE fungiert Eclipse Platform (Mars 4.5), zum Bauen[64] wird *Apache Ant* genutzt und als

[60] Java Platform Debugger Architecture
[61] Zum Beispiel *-agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n*
[62] Eine *OpenSource* Software mit deren Hilfe Anwendungen in Containern isoliert werden können
[63] Das zeigte sich dadurch, dass neue Dateien in der JRE angelegt wurden
[64] Für den *Build*-Prozess

Webserver dient *Apache Tomcat* (7.0.59). Bei abweichendem Systemaufbau können einige der Anweisungen unnötig kompliziert oder nicht durchführbar sein. Doch da die einzelnen Schritte möglichst ausführlich erläutert und mit Erklärungen versehen werden, lohnt sich das Lesen definitiv.

Zuerst muss der entsprechende *DCEVM Release* als *.jar* Datei heruntergeladen[65] werden. Optimalerweise stimmen dabei sowohl die Java-Version als auch die Nummer des Updates (z. B. u92) der *DCEVM* und der installierten JRE[66] überein. Ist dies nicht möglich, sollte der *DCEVM Release* eine höhere Update-Nummer besitzen als die installierte JRE. Die heruntergeladene Datei soll nun in einen Ordner verschoben werden, welcher in den *Docker* Container gespiegelt wird. Mit einer graphischen Benutzeroberfläche könnte der Installer nun über die *Docker* Konsole ausgeführt und die *DCEVM* installiert werden. Da diese allerdings standardmäßig nicht vorhanden ist, muss die Installation im Container durch die folgenden Operationen von Hand durchgeführt werden:

- Entpacke den Installer. Unter *installer/linux_amd64_compiler2/product* findet sich die Datei *libjvm.so*, welche die modifizierte JVM repräsentiert
- Starte die Docker Konsole
- Ermittle das Java Home-Verzeichnis (z. B. durch *java -XshowSettings:all* unter *java.home*) des *Docker* Containers
- Öffne das Verzeichnis *java.home/lib/amd64*
- Erstelle ein neues Verzeichnis *dcevm*
- Kopiere die *libjvm.so* in das neu angelegte Verzeichnis

Das Verfahren lässt sich prinzipiell auf einen anderen Systemaufbau übertragen, auch mit anderen Betriebssystemen. Allerdings weichen dabei die angegebenen Pfade sowie die zu kopierende Datei ab. Für Linux Systeme wird die Datei *libjvm.so* genutzt, Windows Systeme verwenden die Datei *jvm.dll* und für Mac OS Systeme wird die Datei *libjvm.dylib* bereitgestellt.

Unter `https://github.com/HotswapProjects/HotswapAgent/releases` - Stand: 01.12.2016 - kann nun der aktuelle *Release* des *HotswapAgents* heruntergeladen werden. Dabei sollte stets die *.jar* Datei mit den kompilierten Klassen ausgewählt werden. Auch diese Datei muss wieder in einen Ordner verschoben werden, welcher in

[65] `https://github.com/dcevm/dcevm/releases` - Stand: 01.12.2016

[66] Kann durch den Kommandozeilenbefehl *java -XshowSettings:all* unter *java.version* nachgelesen werden. Vor dem Unterstrich steht die Java-Version und danach die Nummer des Updates

den *Docker* Container gespiegelt wird, damit der Server auf sie zugreifen kann. Dort soll sie entpackt werden, sodass auf die Datei *hotswap-agent.properties* zugegriffen werden kann. Diese dient als Konfigurationsdatei für den *HotswapAgent*. In ihr können zahlreiche Anpassungen vorgenommen werden, wie die Angabe von Ordnern in denen Ressourcen auf Änderungen überwacht werden sollen, das Laden von *HotswapAgent* Plugins aus der Anwendung oder die Durchführung von Code-Evolution ohne Nutzung des *Debuggers*[67]. Auch das *Logger*-Level kann in der Datei angepasst werden.

Nachdem die Konfigurationen in der *hotswap-agent.properties* Datei abgeschlossen wurden, muss der Agent noch in den Server integriert werden. Bei dem in der dot-Source GmbH verwendeten Systemaufbau zur Entwicklung mit *hybris* kann dies durch Anpassung der Konfigurationsdatei *local.properties* erreicht werden. In dieser soll folgende Zeile stehen:

- *tomcat.debugjavaoptions=-Xdebug -Xnoagent -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=8000 -javaagent:"/path-to-hotswap-agent/hotswap-agent.jar-propertiesFilePath="/path-to-hotswap-agent/hotswap-agent/hotswap-agent.properties",disablePlugin=ZK -XXaltjvm=dcevm*

Erstmals erwähnt werden die Optionen *-Xdebug* und *-Xnoagent*. *-Xdebug* ermöglicht das *Debugging* der Anwendung und *-Xnoagent* deaktiviert den *sun.tools.debug* Agenten, sodass der *JPDA Debugger* seinen eigenen Agenten anfügen kann. Optionen, welche bereits unter *tomcat.debugjavaoptions* gesetzt sind, können bei der Anpassung beibehalten werden.

Anschließend muss im Home-Verzeichnis der *hybris*-Plattform der Befehl *ant all* ausgeführt werden, da dieser - zusätzlich zum Bauen - den *Wrapper*[68] des Servers aktualisiert. Nun kann der *hybris*-Server mit integriertem *HotswapAgent* im *Debug*-Modus gestartet werden. Wurde der *HotswapAgent* erfolgreich geladen, erscheint in der Konsole ein Text, vergleichbar mit der weiß markierten Ausgabe aus Abbildung 7.

Im letzten Schritt muss noch der *Debugger* von Eclipse Platform mit dem *JDPA Debugger* Server über Port 8000 verbunden werden, sodass ein ferngesteuertes *Debugging* ermöglicht wird. Die entsprechende Konfiguration findet sich in Eclipse unter *Run→ Debug Configurations*. Dort kann durch einen Rechtsklick auf *Remote Java Application* und der Auswahl des Feldes *new* eine neue Verbindung zu einer entfern-

[67] Letzteres führte auf dem *hybris*-Server allerdings zu Fehlern
[68] dt.: Verpackung/Umschlag/Schnittstelle

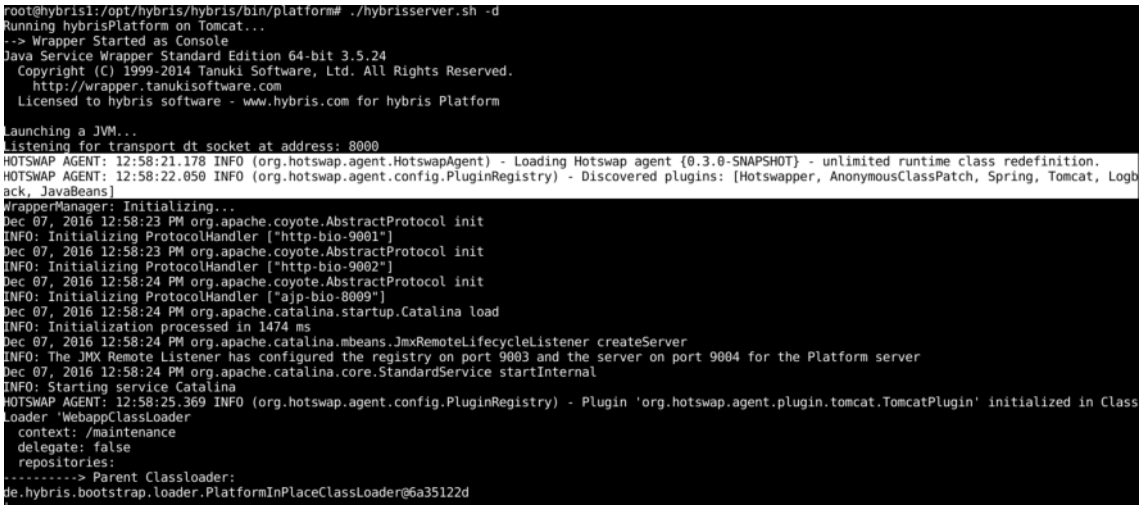

Abbildung 7: Ausgabe der Konsole bei erfolgreicher Integration des *HotswapAgents*

ten Java-Anwendung aufgebaut werden. Diese kann nun im rechten Teil des Fensters konfiguriert werden. Der Name kann beliebig gewählt werden, doch das Feld *Project* sollte geleert werden[69]. Als *Connection Type* muss *Standard (Socket Attach)* gewählt werden, als *Host* - *localhost* und als *Port* der in den VM Argumenten spezifizierte Port *8000*. Nachdem die Änderungen übernommen wurden (*Apply*) kann durch einen Klick auf *Debug* der konfigurierte *Debugger* gestartet werden. Zuletzt sollte noch sichergestellt werden, dass *Project→ Build Automatically* aktiviert ist, sodass eine gespeicherte Datei automatisch gebaut und das *Hot Deployment* ausgelöst wird.

4.3 Zusätzliche Funktionen

Zusätzlich zu dem bisher erläuterten Funktionsumfang von *Hot Deployment* Tools gibt es weitere Funktionen, die ausgereifte *Hot Deployment* Tools abdecken sollten. Grund dafür ist, dass bei der professionellen Entwicklung von Java-Anwendungen häufig nach bestimmten Rahmenstrukturen programmiert werden soll. Das sorgt für eine Übersichtlichkeit und Lesbarkeit der Software, was gerade bei der Entwicklung in Teams von besonderer Wichtigkeit ist. Um diese Art der Programmierung zu ermöglichen und zu unterstützen wurden Frameworks entwickelt.

Diese wurden allerdings nicht unter Berücksichtigung der Möglichkeit von *Hot Deployment* entworfen, sodass die Durchführung von Code-Evolution zu Fehlern in der Anwendung führen kann. Um diesen entgegenzuwirken und dynamische Updates von Framework-spezifischem Code zu ermöglichen, benutzen sowohl *JRebel* als auch

[69] Dies ermöglicht Änderungen in allen Projekten statt nur im angegebenen Projekt

der *HotswapAgent* Plugins. Plugins sind optionale Softwaremodule, die den Zweck haben bestehende Software zu erweitern oder zu verändern.

4.3.1 Spring Plugin

4.3.1.1 Spring und Hot Deployment

Das wohl wichtigste Framework, welches in der dotSource GmbH Verwendung findet, ist das Spring Framework. Dieses stellt ein verständliches Modell zur Gestaltung und Programmierung von Java-Anwendungen in Unternehmen zur Verfügung. Eine der Hauptaufgaben von Spring liegt in der *Dependency Injection*[70].

Die *Dependency Injection* beschreibt ein Entwurfsmuster in der objektorientierten Programmierung. Dabei werden Modulen (Klassen, Objekte) ihre Abhängigkeiten von einer externen Instanz zugewiesen, was auch als Injektion bezeichnet wird (vgl. [ITW]). Ganz nach dem bekannten Prinzip *„Don't call us, we'll call you!"*[71] wird die Verantwortlichkeit bei der Programmausführung auf eine zentrale Komponente, in diesem Fall das Spring Framework, übertragen. Dieses Vorgehen wird auch als *Inversion of Control*[72] (IoC) bezeichnet. Anhand ihrer Implementierung lässt sich die *Dependency Injection* nach drei Arten unterscheiden, der Injektion über *Setter*, Konstruktoren und *Interfaces*.

Da der Aufbau der Spring Konfiguration allerdings in der Regel beim Serverstart bzw. *Deployment* geschieht und *Hot Deployment* Tools gerade diesen Schritt umgehen, können Code-Änderungen zu Fehlern bei der Programmausführung führen. Dies betrifft im speziellen Änderungen an Spring Annotationen, da diese erst beim *Redeploy* in die Spring Konfiguration übertragen werden. Wird dementsprechend beim *Hot Deployment* ein Objekt über eine Spring Annotation angelegt[73] und anschließend aufgerufen, so wird die Code-Änderung zwar auf den Server übertragen, doch die Spring Konfiguration bleibt dieselbe. Da die entsprechende Spring *Bean* nicht aktualisiert wurde, kann das Objekt nicht erzeugt werden und beim Aufruf des Objekts kommt es zu einer *NullPointerException* (NPE).

4.3.1.2 Erläuterung des Plugins

Um die Code-Evolution von Spring Annotationen zu ermöglichen entwickelten die Programmierer des *HotswapAgents* ein Spring Plugin. Dieses sorgt dafür, dass bei

[70] dt.: Injektion von Abhängigkeiten
[71] Frei übersetzt: „Ruf nicht uns an, wir werden dich anrufen!"
[72] Umkehrung des Kontrollflusses
[73] Zum Beispiel *@Autowired* oder *@Resource*

Änderungen in Spring Annotationen auch die entsprechende *Bean* in der Konfiguration aktualisiert wird. Dabei wird das Nachladen all jener Komponenten unterstützt, die vom *ComponentScan*[74] registriert werden. Das Plugin überwacht die im *ComponentScan* angegebenen Basispakete und registriert Events auf den entsprechenden Klassen. Wird eine Klasse aktualisiert oder neu definiert und die Code-Evolution ausgelöst, so prüft das Plugin zunächst, ob Spring Annotationen von den Änderungen betroffen sind. Ist dies der Fall, wird die Klasse an den Spring Container übergeben und dort verarbeitet. Falls zu der Klasse bereits eine *Bean* in der Spring *Registry* existiert, wird diese entfernt[75], die dazugehörigen Caches werden zurückgesetzt und die neue *Bean* wird registriert.

Doch trotz eines vielversprechenden Ansatzes, ist das Spring Plugin des *Hotswap-Agents* noch nicht vollständig ausgereift und muss folglich in der Praxis nicht unbedingt funktionieren. Insbesondere bei spezielleren Anwendungsgebieten, wie dem Spring *Web MVC Framework*, gibt es Annotationen und Konfigurationen, die das Plugin nicht unterstützt. Deshalb wurden sowohl im Minimalbeispiel als auch auf dem *hybris*-Server Praxistests durchgeführt um den Nutzen des Plugins ausführlich zu untersuchen und zu beurteilen.

Im Minimalbeispiel funktionierte das Erstellen neuer Klassen als Spring *Beans* ohne Probleme, sowohl über die *@Controller* Annotation als auch über die *@Component* Annotation[76]. Auch das automatische Laden[77] von Spring *Beans* über die *@Autowired* Annotation lief fehlerfrei ab. Sogar das Hinzufügen, Entfernen und Ändern von spezifischen Handler-Methoden für das *Mapping* von Webanfragen über die *@RequestMapping* Annotation funktionierte problemlos.

Schwerwiegende Probleme traten erstmals bei Verwendung der *@Resource* Annotation auf. Diese hat den gleichen Zweck wie die *@Autowired* Annotation und unterscheidet sich von dieser lediglich in der Durchführung der Suche nach der gewünschten Spring *Bean* und in ihrer Implementierung. Nicht nur dass das Plugin die Code-Evolution von *@Resource* Annotationen nicht unterstützt, die bloße Verwendung von *@Resource* Annotationen kann beim *Hot Deployment* zu Fehlern führen, selbst wenn die durchgeführten Änderungen diese gar nicht betreffen. So wurde beim *Hot Deployment* gelegentlich die komplette Spring Konfiguration aus einem Fehler her-

[74] Ermöglicht das Erstellen und Verknüpfen von Spring Beans durch Annotationen im Code
[75] engl.: unregistered
[76] Anmerkung: Damit das Plugin funktioniert, musste vorher die Zeile *<mvc:annotation-driven/>* in die *...-servlet.xml* Datei eingetragen werden
[77] Automatisierte *Dependency Injection*

aus neu aufgebaut, was zum Stillstand des Servers führte. Ein Server-Neustart wurde unmöglich, lediglich ein Neustart der Eclipse IDE schaffte Abhilfe.

Bei der Suche nach der Fehlerursache war die Frage, was die *@Resource* Annotation von den anderen Annotationen zum automatischen Laden von Spring *Beans* unterscheidet, da sowohl *@Autowired* als auch *@Inject* problemlos funktionierten. Schließlich fand sich ein Unterschied auf Code-Basis, der eine Erklärung für die Fehler lieferte. Während *@Autowired* und *@Inject* den *AutowiredAnnotationBeanPostProcessor* zur *Dependency Injection* nutzen, nutzt die *@Resource* Annotation dafür den *CommonAnnotationBeanPostProcessor*. Auch wenn beide eine beinahe identische Funktionsweise besitzen, sind es doch unterschiedliche *PostProcessor* Klassen, die im Plugin beide behandelt werden müssen. Eine kurze Suche im *HotswapAgent* Code lieferte die Bestätigung. Der *AutowiredAnnotationBeanPostProcessor* wird im Plugin eingebunden und seine Caches werden zurückgesetzt, doch der *CommonAnnotationBeanPostProcessor* kommt im Code nicht vor. Dementsprechend kann die *@Resource* Annotation unmöglich vom Plugin unterstützt werden. Da die *@Resource* Annotation allerdings ohne Probleme durch *@Autowired* ersetzt werden kann, setzt das Plugin die Code-Evolution für alle durch Annotationen bereitgestellten Spring MVC Funktionalitäten um. Somit wird die Nutzbarkeit des Spring Plugins nur in geringem Maße beeinträchtigt.

4.3.1.3 Nutzung auf dem hybris-Server

Weitere Probleme gab es bei der Nutzung des Plugins auf dem *hybris*-Server. Abgesehen davon, dass es bei *hybris* gängige Praxis ist ausschließlich *@Resource* Annotationen zu verwenden, funktioniert das Spring Plugin auch bei *@Autowired* und *@Inject* nicht auf dem *hybris*-Server. Auch an dieser Stelle nahm die Suche nach dem Grund wieder viel Zeit in Anspruch. Die erste Idee war, dass der Grund in der Konfiguration der *.xml* Dateien liegt, da das Problem nur beim *hybris*-Server auftritt und das Projekt ein komplexes Spring Setup besitzt. Durch die große Zahl an *Extensions*[78] gibt es zahlreiche Anwendungskontexte[79] und beim Anlegen einer *Bean* wird meist ein *Alias* verwendet. Da sich das Spring Plugin in die *ComponentScan* Komponente von Spring integriert und diese nicht in jeder *Extension* verwendet wird, war der erste Schritt das Einfügen der Komponente in die zu testende *Extension*. Als dieser Schritt nicht die erwarteten Resultate brachte, folgte eine lange

[78] dt.: Erweiterung
[79] engl.: *ApplicationContext*

Suche nach der korrekten Konfiguration im richtigen Anwendungskontext[80] und die Interpretation einiger Fehlermeldungen, bis sich schließlich die Vermutung festigte, dass das Problem beim Plugin selbst liegt. Im Folgenden sollen die wichtigsten Beobachtungen und Erkenntnisse beschrieben werden, die bei der Eingrenzung des Problems halfen.

Die erste wichtige Beobachtung war, dass auch bei aktiviertem Spring Plugin die eigentliche Code-Evolution problemlos abläuft. Dementsprechend musste das Problem zwangsläufig beim Spring Plugin liegen und hatte nichts mit der sonstigen Funktionsweise des *HotswapAgents* zu tun. Die zweite wichtige Erkenntnis war bei aktiviertem Spring Plugin und erst nach Einfügen der *ComponentScan* Komponente in die zu testende *Extension* beobachtbar. Das Spring Plugin reagiert nicht auf allgemeine Änderungen im Code. Erst wenn Spring Annotationen selbst oder ihr Inhalt betroffen sind, reagiert das Plugin und gibt eine entsprechende Meldung in der Konsole aus. Demnach entscheidet das Plugin korrekt, ob ein Nachladen der Spring Konfiguration notwendig ist und das Problem liegt nicht im Auslösen des Nachladens sondern in der Umsetzung. Da diese allerdings im Minimalbeispiel ohne Probleme funktionierte, kam die entscheidende Idee auf, dass die Änderungen in den falschen Anwendungskontext geladen werden.

Diese führte auf die richtige Spur. Durch professionelle Hilfe aus der Firma konnte das Problem weiter eingegrenzt werden. Zwar wurden die *Beans* neu angelegt, die alten *Beans* entfernt und die entsprechenden Caches zurückgesetzt, doch konnten die Verlinkungen nicht auf allen Ebenen aktualisiert werden. Im Anwendungskontext der *Extension* wurden sie richtig angepasst, aber im Anwendungskontext des *Controllers* war noch immer die alte *Bean* hinterlegt, welche bereits entfernt wurde. Somit konnten die entsprechenden Objekte nicht länger angelegt werden und beim Aufruf des Objekts wurde eine *NullPointerException* geworfen. Zur Bestätigung und für weitere Testzwecke wurde das Problem im Minimalbeispiel nachgebaut, indem als zusätzlicher Anwendungskontext ein Eltern-Kontext hinzugefügt wurde. Die Beobachtungen wiederholten sich nun auch im Minimalbeispiel und das Problem war gefunden.

Die Ursache für das Problem liegt in der Vererbungshierarchie von Anwendungskontexten in Spring. Zwar stehen die Funktionen der Eltern-Kontexte auch ihren Kind-Kontexten zur Verfügung, doch ergibt es aus der Perspektive von Spring keinen Sinn den Eltern-Kontexten Zugriff auf ihre Kind-Kontexte zu gewähren. Durch die eingeschränkte Sichtbarkeit kann das Spring Plugin zwar die Verlinkungen im

[80] Es gibt Eltern-Kontexte und Kind-Kontexte und die Abhängigkeiten waren nicht immer klar

Eltern-Kontext[81] aktualisieren, hat allerdings keinen Zugriff auf die Verlinkungen im Kind-Kontext[82]. Interessanterweise bewirkt ein Nachladen der Klasse des *Controllers* im Minimalbeispiel nachträglich ein Aktualisieren der Verlinkungen im Anwendungskontext des *Controllers*, wodurch Objekte wieder korrekt angelegt werden können.

Eine Lösung des Problems würde viel Aufwand erfordern, da sich an der Sichtbarkeit in den Anwendungskontexten von Spring nichts ändern lässt. Die einzige effektive Möglichkeit das Problem zu umgehen, wäre der Aufbau einer eigenen und vollständigen Kontexthierarchie im Spring Plugin. Diesen Weg wählte auch *JRebel*. Deshalb ist das Spring Plugin auf einem *hybris*-Server vorerst nicht nutzbar.

4.3.2 Eigenes Plugin

Die Idee ein eigenes Plugin zu schreiben entstand lange bevor klar war, dass das Spring Plugin auf dem *hybris*-Server nicht genutzt werden kann. Da das Spring Plugin lediglich das Nachladen von Spring Konfigurationen basierend auf Annotationen ermöglicht, fehlt ihm eine wichtige Funktionalität. Zum Aufbau der Spring Konfiguration werden immer auch *.xml* basierte Konfigurationsdateien verwendet. Selbst wenn das Erzeugen aller *Beans* und die automatisierte *Dependency Injection* vollständig über Annotationen geschehen, muss zuerst der *ComponentScan* über die *.xml* Datei konfiguriert werden. Zudem bieten die *.xml* Dateien erweiterte Möglichkeiten, wie die Definition eines *Alias* oder das Initialisieren von primitiven Datentypen. Auf dem *hybris*-Server der Firma werden die meisten *Beans* über *.xml* Dateien angelegt und die automatisierte *Dependency Injection* geschieht durch Annotationen. Deshalb wurde die Entwicklung eines eigenen Plugins zum Nachladen von *.xml* basierter Spring Konfiguration zu einer wesentlichen Aufgabe bei der Einrichtung der *DCEVM* auf dem *hybris*-Server.

Um das Nachladen von *.xml* basierter Spring Konfiguration zu ermöglichen, setzt das entwickelte Plugin folgende drei Schritte um:

- Initialisierung
- Beobachtung der *.xml* Dateien auf Änderungen
- Nachladen der Spring Konfiguration

[81] Anwendungskontext der *Extension*
[82] Anwendungskontext des *Controllers*

4.3.2.1 Initialisierung

Benutzerdefinierte Plugins können beim *HotswapAgent* direkt als Teil der Anwendung entwickelt werden. Damit das Plugin beim Start des *HotswapAgents* geladen wird, muss in der *hotswap-agent.properties* Datei die Zeile *pluginPackages=your.plugin.package* eingefügt werden. Zudem muss zum Kompilieren die *JAR Dependency* des Agenten in die Anwendung eingefügt werden[83], da der *HotswapAgent* selbst nicht als Teil der Anwendung geladen werden darf. Über die *@Plugin* Annotation wird die Klasse als Plugin gekennzeichnet und es können zusätzliche Informationen angegeben werden. Die eigentliche Initialisierung geschieht in einer statischen, durch *@OnClassLoadEvent* annotierten Methode *transform*, welche im folgenden Codeausschnitt (Listing 1) zu sehen ist.

```
1  @Plugin(name = "SpringXmlPlugin", description = "Hotswap agent plugin for spring
2    xml reload.",testedVersions="Spring 4.1.7", expectedVersions="Spring 4.1.7")
3  public class SpringXmlPlugin {
4
5    public static final String HOOK_INTO = "org.springframework.web.servlet.
         DispatcherServlet";
6    private static AgentLogger LOGGER = AgentLogger.getLogger(SpringXmlPlugin.
         class);
7    Object servletObject;
8
9    public void init(Object servletObject) throws IOException {
10     this.servletObject = servletObject;
11     LOGGER.info("Plugin {} initialized on class {}", getClass(), this.
           servletObject);
12   }
13
14   @OnClassLoadEvent(classNameRegexp = HOOK_INTO)
15   public static void transformDispatcherServlet(CtClass ctClass) throws
         NotFoundException, CannotCompileException {
16     String src = PluginManagerInvoker.buildInitializePlugin(SpringXmlPlugin.
           class);
17     src += PluginManagerInvoker.buildCallPluginMethod(SpringXmlPlugin.class, "
           init", "this", "java.lang.Object");
18     for (CtConstructor constructor : ctClass.getDeclaredConstructors()) {
19       try {
20         constructor.insertBeforeBody(src);
21       } catch (CannotCompileException e) {
22         e.printStackTrace();
23       }
24     }
25     LOGGER.debug(HOOK_INTO + " has been enhanced.");
26   }
27 }
```

Listing 1: Initialisierung des Plugins

[83] Genaue Anweisungen: `http://hotswapagent.org/mydoc_custom_plugins.html` - Stand: 21.12.2016

Die *@OnClassLoadEvent* Annotation (Zeile 14) sorgt für den Aufruf der Methode *transformDispatcherServlet*, sobald die unter *classNameRegexp* angegebene Klasse geladen wird. Das Argument *ctClass* enthält die entsprechende Klasse. Die Idee für die Initialisierung von *HotswapAgent* Plugins ist, dass der notwendige Code in eine für das Plugin wichtige Klasse eingefügt wird. Im vorgestellten Plugin hat die gewählte Klasse keine über die Initialisierung hinausgehende Bedeutung. Allerdings sollte im Idealfall der Konstruktor der Klasse nur einmal benutzt werden, da nur eine Instanz des Plugins benötigt wird.

Zur Initialisierung des Plugins wird der Code zuerst in den Zeilen 16 und 17 erzeugt und anschließend in den Zeilen 18 bis 24 zu den Konstruktoren der Klasse *ctClass* als letzte Anweisung hinzugefügt[84]. Dabei wird in Zeile 17 die Methode *init* (Zeile 9 bis 12) übergeben, welche von der Klasse aufgerufen werden soll. In dieser Methode können weitere Operationen zur Initialisierung ausgeführt werden, wie beispielsweise die Speicherung des Objekts bei dessen Entstehung das Plugin initialisiert wurde (Zeile 10) oder die Ausgabe eines Textes in der Konsole (Zeile 11).

4.3.2.2 Beobachtung der .xml Dateien

Damit der *HotswapAgent* eine Datei beobachten kann, muss er wissen wo sie liegt. In der *hotswap-agent.properties* Datei können durch den Eintrag *watchResources=...* Pfade zu den Ordnern angegeben werden, in denen die *.xml* Dateien liegen. Anschließend können mithilfe der *@OnResourceFileEvent* Annotation sogenannte *EventWatcher* registriert werden. Standardmäßig überwachen diese alle in der Datei *hotswap-agent.properties* angegebenen Ressourcen[85]. Allerdings können in der *@OnResourceFileEvent* Annotation die Pfade beschränkt und die Ergebnisse gefiltert werden.

Wie in der ersten Zeile von Codeausschnitt 2 zu sehen ist, geschieht die Beschränkung der Pfade durch die Variable *path*. Die Pfade können auf Unterordner oder auf eine spezielle Datei eingegrenzt werden. Der Schrägstrich steht für keinerlei Begrenzung. Somit werden alle Ressourcen überwacht und anschließend die *.xml* Dateien von Spring herausgefiltert. Das geschieht durch die Variable *filter*. Diese bekommt als Wert einen regulären Ausdruck übergeben, auf dessen Einhaltung alle vorläufigen Ergebnisse geprüft werden. Dabei werden allerdings nicht nur die Dateinamen, sondern die kompletten *URIs*, mit dem regulären Ausdruck verglichen. Da der Ausdruck *.** für eine unbekannte Anzahl beliebiger Zeichen steht, können durch ihn beliebi-

[84] Genauer: Es wird ein *String* von Java-Anweisungen aufgebaut und anschließend über die Java Reflection API in die Klasse eingefügt

[85] Inklusive der Ressourcen aller Unterordner

ge Pfade akzeptiert werden. Registriert der *EventWatcher* eine Änderung in den überwachten und gefilterten Ressourcen, wird die Funktion aufgerufen und der *URI* der Ressource wird der Funktion als Parameter übergeben.

```
1  @OnResourceFileEvent(path = "/", filter = ".*spring.*.xml")
2  public synchronized void watchForResourceChange(URI uri) {
3    currentEventTime = System.currentTimeMillis();
4    if(Math.abs(currentEventTime - lastEventTime) < timeout) {
5      lastEventTime = System.currentTimeMillis();
6    } else {
7      refreshContextMagic(getContextsForUriMagic(uri));
8      LOGGER.info("A change happened: " + uri.toString());
9      lastEventTime = System.currentTimeMillis();
10   }
11 }
```

Listing 2: Beobachtung der *.xml* Dateien

Die Zeilen 3 bis 6 und 9 bis 10 sind notwendig, da in der Praxis mehrere Funktionsaufrufe bei nur einem *Event* stattfinden[86]. Dies lässt sich durch die Einführung eines manuellen *Timeouts* umgehen. Es wird die Zeit des aktuellen *Events* mit der Zeit des letzten *Events* verglichen und nur wenn die Differenz einen gewissen Wert (*timeout*) überschreitet, werden die gewünschten Operationen durchgeführt. Die Anweisung *synchronized* erzwingt einen sequenziellen Aufruf der Methode, sodass immer nur ein *Thread* zur gleichen Zeit die Methode betritt und der *Timeout* garantiert funktioniert.

Schließlich wird in Zeile 7 das Nachladen der Spring Konfiguration durchgeführt. Zuerst werden alle von den Änderungen betroffenen Anwendungskontexte ermittelt und anschließend werden diese aktualisiert. Der eigentliche Code, welcher später in Zeile 7 stehen wird, wird am Ende von Abschnitt 4.3.2.3 beschrieben, da zum Verständnis noch weitere Klassen und Konzepte eingeführt werden müssen.

4.3.2.3 Nachladen der Spring Konfiguration

Für das das Nachladen der Spring Konfiguration gibt es verschiedene Ansätze mit unterschiedlicher Komplexität. Der effizienteste Ansatz wäre der manuelle Vergleich der alten Spring Konfiguration mit der neuen Konfiguration. Dabei müssten spe-

[86] Theoretisch besitzt die Annotation *@OnResourceFileEvent* ein Element *timeout*, das benutzt wird um mehrere Events bis zu einem bestimmten Zeitabstand zu verschmelzen. Doch in der Praxis wird die Funktion zeitgleich von mehreren *Threads* aufgerufen

zielle Spring Klassen genutzt werden um kleinste Änderungen Schritt für Schritt umzusetzen, wie zum Beispiel das Hinzufügen, Entfernen, Modifizieren und Aktualisieren von *Beans*. Dieser Ansatz ist allerdings sehr aufwändig in der Umsetzung und würde den Rahmen der Masterarbeit sprengen.

Ein leicht umzusetzender Ansatz wäre das Aktualisieren der vollständigen Spring Konfiguration. Dieser ist allerdings höchst ineffizient, da der Aufbau der Spring Konfiguration ein Hauptbestandteil des Serverstarts ist und dadurch nur unwesentlich weniger Zeit benötigt als der komplette Serverstart. Ein Kompromiss zwischen Effizienz und Komplexität fand sich in einem dritten Ansatz.

Da der *hybris*-Server aus zahlreichen *Extensions* besteht, die in der Regel ihren eigenen Anwendungskontext aufbauen, bedeutet ein automatisches Aktualisieren der wenigen betroffenen Kontexte bereits eine deutliche Zeitersparnis gegenüber dem Serverstart. Im Optimalfall muss nur der Anwendungskontext aktualisiert werden, welcher durch die geänderte *.xml* Datei aufgebaut wurde. Wenn dieser jedoch Kind-Kontexte besitzt, müssen auch diese aktualisiert werden, sodass die Änderungen bis zum *Controller* weitergereicht werden. Allerdings tritt an dieser Stelle das in Abschnitt 4.3.1.3 beschriebene Problem der mangelnden Sichtbarkeit zwischen Eltern-Kontext und Kind-Kontext wieder auf. Da der Eltern-Kontext seine Kind-Kontexte nicht kennt, können diese auch nicht aktualisiert werden. Abhilfe schaffen die Klassen *CallRegistry* und *ApplicationContextRegistry*.

```
1  public class CallRegistry implements ApplicationContextAware {
2    public void setApplicationContext(ApplicationContext applicationContext)
         throws BeansException {
3      try {
4        ApplicationContextRegistry.registerApplicationContext(applicationContext);
5      } catch (SecurityException | NoSuchMethodException | IllegalAccessException
           | IllegalArgumentException | InvocationTargetException e) {
6        e.printStackTrace();
7      }
8    }
9  }
```

Listing 3: Aufruf der *ContextRegistry*

Die Klasse *CallRegistry* implementiert das von Spring bereitgestellte Interface *ApplicationContextAware* (Zeile 1) und ermöglicht daher das Anlegen einer *Bean*, welche sich des Anwendungskontextes bewusst ist, der sie erzeugt hat. Beim Erzeugen der *Bean* wird die Methode *setApplicationContext* (Zeile 2) aufgerufen, welche den Anwendungskontext als Parameter übergeben bekommt. Dies ermöglicht den Zugriff auf die Kontexthierarchie. Zur korrekten Funktionsweise des Plugins muss in jedem

Anwendungskontext eine *Bean* der Klasse *CallRegistry* erzeugt werden[87], sodass die Kontexthierarchie vollständig erfasst werden kann. Die statische Klasse *ApplicationContextRegistry* kümmert sich nun um die Zuordnung der *.xml* Dateien zu den von einer Änderung betroffenen Anwendungskontexten. In Zeile 4 von Codeausschnitt 3 wird dazu eine ihrer Methoden aufgerufen.

```
1  public class ApplicationContextRegistry {
2
3    private static final AgentLogger LOGGER = AgentLogger.getLogger(
         SpringXmlPlugin.class);
4    private static final Map<String, ArrayList<
         AbstractRefreshableConfigApplicationContext>> APPLICATION_CONTEXTS = new
         HashMap<>();
5
6    public static void registerApplicationContext(ApplicationContext context)
         {...}
7    public static void handleParentContext(ApplicationContext parent, ArrayList<
         AbstractRefreshableConfigApplicationContext> childList) {...}
8
9    public static ArrayList<AbstractRefreshableConfigApplicationContext>
         getContextsForURI(URI uri) {
10     String path = uri.toString();
11     String[] pathField = path.split("WEB-INF");
12     path = "/WEB-INF" + pathField[1];
13     return APPLICATION_CONTEXTS.get(path);
14   }
15 }
```

Listing 4: Verwaltung der *ContextRegistry*

Die Zuordnung der *.xml* Dateien zu den Anwendungskontexten erfolgt über die in Zeile 4 von Codeausschnitt 4 angelegte *Map*. Eine *Map* ist ein Objekt, welches Schlüssel auf Werte abbildet. Im Plugin wird jeweils ein *String* auf eine Liste von Kontexten abgebildet. Ziel ist, dass bei Änderung einer *.xml* Datei lediglich ihr Pfad angegeben werden muss und die *ContextRegistry* alle Kontexte zurückgibt, die von der Änderung betroffen sind. Für den Aufbau der *Map* sind die Methoden *registerApplicationContext* (Zeile 6) und *handleParentContext* (Zeile 7) zuständig, welche hier aufgrund ihrer Komplexität nur angedeutet wurden.

registerApplicationContext ist die von der Klasse *CallRegistry* aufgerufene Methode und bekommt als Argument den Anwendungskontext übergeben. Im ersten Schritt ermittelt sie die Pfade der zum Kontext gehörenden *.xml* Dateien. Da die Methode *getConfigLocations*[88] nicht zugänglich (*protected*) ist, wird dazu die Java Reflection API genutzt. Anschließend werden die Pfade der *.xml* Dateien als Schlüssel für

[87] Durch Definieren der *Bean* in den entsprechenden *.xml* Dateien
[88] Spring intern

die *Map* benutzt und der übergebene Anwendungskontext wird jeweils[89] in einer zugehörigen Liste abgespeichert. Dabei wird allerdings noch nicht die Kontexthierarchie betrachtet. Dafür ist die Methode *handleParentContext* zuständig, welche für jeden Kontext aufgerufen wird, der einen Eltern-Kontext besitzt.

handleParentContext ist eine rekursive Funktion. Sie ruft sich selbst mit dem jeweiligen Eltern-Kontext des übergebenen Kontexts auf (wenn vorhanden) und speichert zugleich alle bisher besuchten Kontexte in einer Liste. In jeder Rekursion ermittelt sie die Pfade der zum übergebenen Kontext gehörenden *.xml* Dateien und benutzt sie, wie schon *registerApplicationContext*, um die *Map* zu ergänzen. Allerdings werden hier nicht nur der übergebene Kontext, sondern ebenfalls alle Kind-Kontexte (falls nicht bereits vorhanden) zur Liste hinzugefügt. Da ein Eltern-Kontext jedoch immer vor seinem Kind-Kontext aktualisiert werden sollte, muss beim Hinzufügen zur Liste stets darauf geachtet werden, dass ein Kontext nach allen in der Hierarchie über ihm stehenden Kontexten[90] und vor allen in der Hierarchie unter ihm stehenden Kontexten[91] eingeordnet wird. Wurden alle Anwendungskontexte durch die *CallRegistry Bean* erfasst[92], werden durch dieses Vorgehen ebenfalls alle Zusammenhänge zwischen Eltern- und Kind-Kontexten beachtet.

Die letzte Methode *getContextsForURI* (Zeile 9 bis 14) dient als Schnittstelle zwischen dem *SpringXmlPlugin* und der *Map*. Für den gegebenen *URI* einer *.xml* Datei soll sie die Liste der von einer Änderung betroffenen Anwendungskontexte zurückgeben. Dazu muss der *URI* in einen *String* umgewandelt (Zeile 10) und dieser in die richtige Form gebracht werden (Zeile 11 und 12)[93]. Anschließend kann in Zeile 13 der *String* als Schlüssel für die *Map* genutzt und die Liste von Kontexten zurückgegeben werden.

Damit stehen nun alle Komponenten bereit, die zum Aktualisieren der betroffenen Kontexte gebraucht werden. Der in Zeile 7 von Codeausschnitt 2 verkürzt dargestellte Code wird durch die in Codeausschnitt 5 folgenden Zeilen umgesetzt.

Da der kompilierte Code der Klasse *SpringXmlPlugin* im Klassenlader des *HotswapAgents* ausgeführt wird, kann er nicht auf Klassen der Anwendung oder des Frameworks zugreifen. Zu diesem Zweck muss der Klassenlader der Anwendung verwendet

[89] Unter bestimmten Bedingungen. Ein Anwendungskontext darf z. B. nur einmal in der Liste vorkommen
[90] Nach seinen Eltern und deren Eltern usw.
[91] Vor seinen Kindern und deren Kindern usw.
[92] Theoretisch würden die Blätter aller Kontextbäume ausreichen
[93] Der Code funktioniert garantiert für *.xml* Dateien im Ordner *WEB-INF*. Ansonsten könnten Anpassungen erforderlich werden

werden, welcher durch die Zeilen 1 und 2 von Codeausschnitt 5 initialisiert wird[94]. Die *@Init* Annotation aus Zeile 1 bewirkt die Injizierung der Abhängigkeit (*Dependency Injection*) während der Initialisierung des Plugins. Die Variable *contextList* (Zeile 4) wird zum Zwischenspeichern der Kontexte benötigt.

```
1    @Init
2    ClassLoader appClassLoader;
3
4    ArrayList<AbstractRefreshableConfigApplicationContext> contextList;
5    ...
6        contextList = (ArrayList<AbstractRefreshableConfigApplicationContext>)
             appClassLoader.loadClass("org.hotswap.agent.plugin.springxml.
             ApplicationContextRegistry").getDeclaredMethod("getContextsForURI",
             URI.class).invoke(null, uri);
7
8        for(int i = 0; i < contextList.size(); i++) {
9          LOGGER.info("Kontext {}: {}", i+1, contextList.get(i));
10         appClassLoader.loadClass("org.springframework.context.support.
               AbstractApplicationContext").getDeclaredMethod("refresh", null).
               invoke(contextList.get(i), null);
11       }
```

Listing 5: Aktualisieren der Kontexte

Da die Klasse *ApplicationContextRegistry* ebenfalls vom Klassenlader der Anwendung geladen wird, muss zum Aufruf der Methode *getContextsForURI* die Java Reflection API genutzt werden. Dies geschieht in Zeile 6 durch Verwendung des gespeicherten Klassenladers. Nachdem die zurückgegebenen Kontexte in *contextList* hinterlegt wurden, wird in Zeile 8 über die Kontexte iteriert[95]. Im jeweiligen Kontext wird nun in Zeile 10 der Reihe nach (über Reflection) die Methode *refresh* aufgerufen und der Anwendungskontext neu geladen.

4.3.2.4 Zusammenfassung der Funktionsweise

Um einen besseren Überblick über die Funktionsweise des entwickelten Plugins zu bekommen, soll diese unter Zuhilfenahme von Abbildung 8 zusammengefasst werden. Das Plugin besteht aus zwei Hauptbestandteilen. Die Klasse *SpringXmlPlugin* überwacht *.xml* Dateien und löst gegebenenfalls das Aktualisieren der Anwendungskontexte aus, während sich die Klasse *ApplicationContextRegistry* um die Zuordnung der *.xml* Dateien zu den Anwendungskontexten kümmert.

[94] Die Initialisierung geschieht nicht innerhalb der Methode *watchForResourceChange* sondern als Instanzvariable der Klasse *SpringXmlPlugin*

[95] Dabei kann keine *ForEach* Schleife verwendet werden, da auf die Kontexte nur über Reflection zugegriffen werden kann

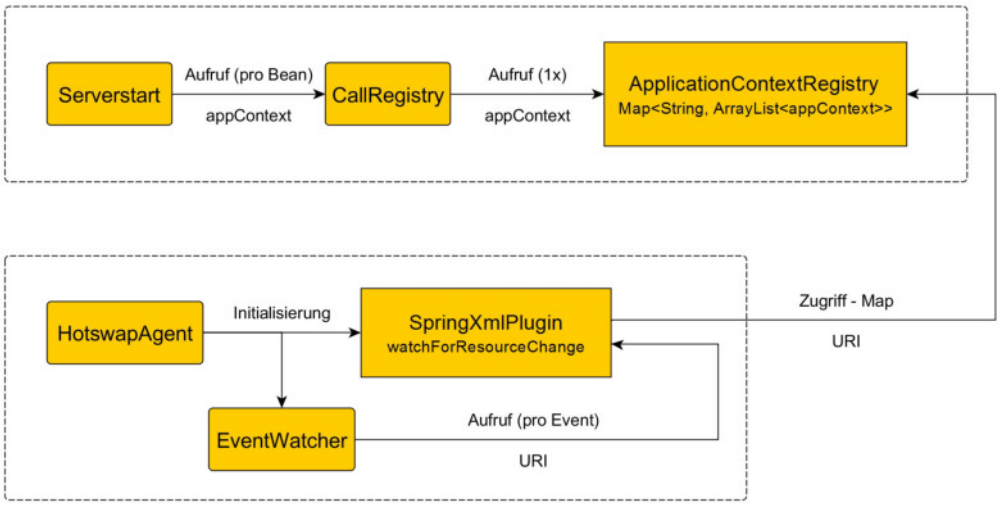

Abbildung 8: Veranschaulichung des *SpringXmlPlugins*

Der Aufbau der *ContextRegistry* geschieht bereits beim Serverstart. Sobald eine Spring *Bean* der Klasse *CallRegistry* erzeugt wird, ruft diese automatisch die statische Klasse *ApplicationContextRegistry* auf und übergibt ihr den Anwendungskontext, in welchem sie erzeugt wurde. Die Klasse *ApplicationContextRegistry* wiederum ermittelt die *.xml* Dateien, die den Kontext aufgebaut haben und verwendet sie als Schlüssel für die *Map*. Als Wert dient der Anwendungskontext. Dieser Schritt wird rekursiv für alle Eltern-Kontexte wiederholt. Wird eine ermittelte *.xml* Datei bereits als Schlüssel genutzt, wird der zugehörige Kontext zur vorhandenen Liste von Anwendungskontexten geordnet[96] hinzugefügt.

Die Klasse *SpringXmlPlugin* wird durch den *HotswapAgent* initialisiert. Aufgrund der *@OnResourceFileEvent* Annotation wird dabei ein *EventWatcher* registriert, der die *.xml* Dateien überwacht und auf Änderungen reagiert. Tritt ein solches *Event* ein, wird die Methode *watchForResourceChange* aufgerufen und bekommt als Parameter den *URI* der *.xml* Datei übergeben. Diese erlangt durch die Methode *getContextsForURI* der Klasse *ApplicationContextRegistry* Zugriff auf die *Map* und erhält alle zum *URI* gehörenden Anwendungskontexte. Anschließend ruft sie die Spring-Methode *refresh* auf und aktualisiert so die Kontexte.

[96] Eltern-Kontexte vor ihren Kind-Kontexten

5 Evaluation

Das Kapitel *Evaluation* setzt sich mit der Frage auseinander, inwieweit die *DCEVM* dazu geeignet ist ihren angestrebten Zweck zu erfüllen. Dieser wurde bereits in Abschnitt 2.1 unter dem Stichwort *Hot Deployment* zusammengefasst und umfasst die Minimierung des, durch wiederholtes *Deployment* entstehenden, Zeitverlustes. Dabei spielen insbesondere empirische Daten wie Umfrageergebnisse und Praxistests eine wichtige Rolle. Daher wurden einige Mitarbeiter der dotSource GmbH zu ihren praktischen Erfahrungen befragt und es wurden eigene Tests durchgeführt. Als Bewertungsmaßstab dient das kommerzielle Tool *JRebel*.

Zudem soll das selbst entwickelte *SpringXmlPlugin* untersucht und bewertet werden. Von besonderer Bedeutung ist dabei die Beantwortung der Frage, wie viel Zeit das Aktualisieren der betroffenen Anwendungskontexte bei Änderung der Spring Konfiguration in den *.xml* Dateien in Anspruch nimmt. Zudem sollen der Konfigurationsaufwand bewertet und eventuell auftretende Probleme beurteilt werden.

5.1 Vergleich mit JRebel

Da sowohl die *DCEVM* als auch *JRebel* aktuell in der dotSource GmbH eingesetzt werden, bietet sich ein Vergleich beider Tools zur Evaluation der *DCEVM* an. Zudem müssen Unternehmen individuelle Entscheidungen über den Einsatz der jeweiligen Tools treffen, was die Bedeutung eines Vergleiches weiter erhöht. Dabei muss die *DCEVM* als *OpenSource Hot Deployment* Software zwar nicht den gleichen Maßstäben gerecht werden wie *JRebel*[97], doch muss der angestrebte Zweck[98] mit einer vergleichbaren Qualität umgesetzt werden.

Um dies zu bewerten sollen zwei Bereiche untersucht werden - die Kernkomponente und die Framework-Unterstützung. Die Kernkomponente umfasst die Bewertung der Software ohne Betrachtung der eingesetzten Frameworks. Im Bereich Framework-Unterstützung sollen die Relevanz der verwendeten Frameworks und deren Auswirkungen auf die Erfüllung des angestrebten Zwecks untersucht werden.

5.1.1 Kernkomponente

Um die Kernkomponente der *DCEVM* zu bewerten, sollen die folgenden Kriterien untersucht werden:

[97] Zum Beispiel beim Support
[98] Minimierung des Zeitverlustes

- Usability
- Zeitersparnis
- Konfigurationsaufwand
- Support
- Kosten

Usability[99] bezeichnet den Umfang, in dem ein Produkt durch bestimmte Benutzer in einem spezifischen Anwendungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen. Effektivität und Effizienz der *DCEVM* wurden bereits in der Diplomarbeit [TW11] ausführlich evaluiert. Nun soll betrachtet werden, inwieweit der Softwareentwickler beim *Debugging* unterstützt wird, ohne dass er im Entwicklungsprozess gestört wird. Prinzipiell ist eine *Hot Deployment* Software dann zufriedenstellend, wenn sie leicht nutzbar ist und somit ihr Einsatz kaum wahrgenommen wird. Dieses Kriterium erfüllt *JRebel* vollständig, doch bei der *DCEVM* wurden in der Praxis (auf dem *hybris*-Server) vereinzelt Probleme festgestellt. So werden *Breakpoints* in ausgetauschten Klassen nicht länger beachtet und gelegentlich bleibt der Server beim Start hängen. Zudem wurden Ausgaben in der Konsole bei erfolgtem Austausch von Dateien vermisst. Diese konnten allerdings durch eine kleine Änderung im Code des *HotswapAgents* hinzugefügt werden, was auf einen Vorteil der *DCEVM* gegenüber *JRebel* hinweist: Da die *DCEVM* eine *OpenSource* Software ist, lassen sich kleine Anpassungen einfach umsetzen, wodurch unter anderem auch die *Usability* verbessert werden kann.

Die **Zeitersparnis** ist eines der wichtigsten Kriterien, da es dem Zweck der Tools sehr nahe kommt. Da sowohl die *DCEVM* als auch *JRebel* die normale Programmausführung nur minimal behindern, weißt die eingesparte Zeit keine nennenswerten Unterschiede auf, solange beide Tools problemlos funktionieren. Da dies allerdings nicht immer der Fall ist, soll der Fokus auf den Änderungen liegen, die einen Server-Neustart erfordern. Es wurde eine Umfrage durchgeführt um unter anderem zu ermitteln, wie häufig solche Änderungen auftreten. Da hier nur die Kernkomponente betrachtet wird, bleibt als Problemquelle das Entfernen oder Ersetzen von Superklassen, welches die *DCEVM* (für Java 8) noch nicht implementiert. Laut den Entwicklern der dotSource GmbH beinhalten dies nur wenige bis sehr wenige Änderungen, durchschnittlich etwa 15 Prozent. Trotzdem ist der Effekt spürbar und zeigt einen großen Nachteil der *DCEVM* gegenüber *JRebel* auf.

[99] dt.: Benutzerfreundlichkeit oder Gebrauchstauglichkeit

Auch im **Konfigurationsaufwand** gibt es Unterschiede zwischen den beiden Tools. Hauptproblem bei der *DCEVM* sind fehlende Anleitungen für die Einrichtung auf speziellen Systemen[100]. Allerdings soll die vorliegende Masterarbeit diesem Problem entgegenwirken. Ist die Frage nach dem *Wie* geklärt, lässt sich die *DCEVM* vergleichsweise einfach in eine bestehende Entwicklungsumgebung integrieren, da die Pfade zu den *.class*-Dateien automatisch durch den *HotswapAgent* ermittelt werden. Von den Mitarbeitern der dotSource GmbH wurde die Güte der Integration der *DCEVM* im Durchschnitt mit mittelmäßig bis gut bewertet. *JRebel* erhielt eine durchschnittliche Bewertung von gut bis sehr gut, wodurch *JRebel* auch in dieser Kategorie etwas besser abschneidet.

Der **Support** von *JRebel* ist als kommerzielles Produkt deutlich stärker ausgeprägt als der eines vergleichbaren *OpenSource* Produktes. Auf Beiträge im Forum wird jederzeit schnell reagiert und eine Lösung angeboten. Zudem stellt *JRebel* eine spezielle E-Mail Adresse für Support-Anfragen zur Verfügung. Zwar besitzen auch die *DCEVM* und der *HotswapAgent* jeweils ein eigenes Forum, doch lässt eine Lösung des beschriebenen Problems gelegentlich längere Zeit auf sich warten und auf E-Mails wird nicht immer reagiert. Zusätzlich ist nicht immer klar, ob ein Problem von der *DCEVM* oder vom *HotswapAgent* ausgelöst wird. Das Ergebnis der Umfrage wirkt allerdings ernüchternd, denn keiner der Umfrageteilnehmer hat jemals den Support von *JRebel* genutzt, was den Wert des Kriteriums stark reduziert.

Im Bereich **Kosten** liegt die *DCEVM* augenfällig in Führung, da sie als *OpenSource* Software kostenlos und die Nutzung von *JRebel* mit großen Ausgaben verbunden ist. Allerdings sollte nicht außer Acht gelassen werden, dass bei der *DCEVM* kleinere Anpassungen notwendig sind, um auftretende Probleme zu beheben. Der erforderliche Aufwand kann zu den Kosten hinzugerechnet werden, da auch Arbeitskräfte bezahlt werden müssen. Dennoch sind die Kosten von *JRebel* deutlich höher.

Zusammenfassend lässt sich sagen, dass die *DCEVM* ihren angestrebten Zweck erfüllt. Sie gibt als *OpenSource* Software jedem Entwickler die Möglichkeit, den beim *Deployment* entstehenden Zeitverlust zu minimieren. Dennoch besitzt sie einige Nachteile gegenüber dem kommerziellen Tool *JRebel*, sodass eine Entscheidung für oder gegen die *DCEVM* von den Erwartungen des jeweiligen Unternehmens abhängt. Als frei verfügbare und erweiterbare Software bietet sie die Möglichkeit zum produktiven Arbeiten mit Java im Softwareentwicklungsprozess, ohne dafür eine Gegenleistung zu verlangen. Dabei müssen allerdings kleinere Probleme hingenommen oder es muss Zeit in deren Lösung investiert werden. Zudem ist ihre Ef-

[100] Zum Beispiel in einem *Docker* Container

fektivität abhängig von den durchzuführenden Änderungen. *JRebel* dagegen bietet eine stabile und ausgereifte Software mit geringem Konfigurationsaufwand, gutem Support und wenigen Problemquellen. Ob das die Lizenzkosten von 475 $ pro Jahr rechtfertigt, sollte nach eigenem Ermessen beurteilt werden.

5.1.2 Framework-Unterstützung

Die Bedeutung der Framework-Unterstützung ist schwer zu bewerten, da verschiedenartige Unternehmen in unterschiedlichen Projekten grundverschiedene Frameworks verwenden. Zudem müssen einige Framework-Konfigurationen häufiger aktualisiert werden als andere. Wenn sie direkt von Code-Änderungen betroffen sind, kann die Möglichkeit zum Nachladen durch die *Hot Deployment* Software von entscheidender Bedeutung sein. Doch auch dann hängt diese noch von der Häufigkeit der entsprechenden Code-Änderung ab. Deshalb muss die Wichtigkeit der Framework-Unterstützung von jedem Unternehmen selbstständig beurteilt werden. Fest steht, dass *JRebel* mehr Frameworks unterstützt und die entsprechenden Plugins ausgereifter sind.

Ein bedeutsames Framework für den Softwareentwicklungsprozess in der dotSource GmbH ist das Spring Framework. Für dieses stellt der *HotswapAgent* zwar ein Plugin zur Verfügung, allerdings funktioniert es auf dem *hybris*-Server nicht (siehe Abschnitt 4.3.1.3) und es stellt von vornherein kein Nachladen von *.xml* Dateien zur Verfügung. Doch auch das Spring Plugin von *JRebel* funktioniert nicht fehlerfrei. Bei Verwendung des <*alias*> Tags treten gelegentlich Probleme auf, die einen Neustart des Servers erfordern.

Laut Umfrage betrifft in der dotSource GmbH etwa jede dritte Änderung das Spring Framework, was die Zeitersparnis durch die *DCEVM* deutlich reduziert. Dies verleiht dem im folgenden Kapitel evaluierten *SpringXmlPlugin* ein zusätzliches Gewicht, da das Aktualisieren aller betroffen Spring Kontexte zusätzlich Änderungen von Spring Annotationen nachlädt.

Weitere verwendete Frameworks, die bei Änderungen einen Server-Neustart erfordern, sind das ZK-Framework (*hybris-Backoffice*) und das Spring-*Backoffice*-Framework. Diese werden aktuell weder von der *DCEVM* noch von *JRebel* unterstützt.

5.2 Eigenes Plugin

Das *SpringXmlPlugin* unterstützt prinzipiell alle Änderungen der Spring Konfiguration. Es verwendet keine der Spring-Methoden für kleinste Änderungen, sondern

lediglich eine einzelne, von Spring speziell für diesen Zweck bereitgestellte Methode *refresh*, welche die Spring Konfiguration des Anwendungskontextes vollständig aktualisiert. Dementsprechend werden alle möglichen Spring Änderungen unterstützt, inklusive Annotationen. Allerdings gibt es auch Beschränkungen. Wird in der gleichen Änderung eine *Bean* in einer *.xml* Datei angelegt und über eine Annotation eingebunden, wird eine *Exception* geworfen. Eine weitere Aktualisierung behebt dieses Problem.

Zudem ist das Plugin auf Anwendungskontexte beschränkt, die eine Klasse/Unterklasse der Spring Klasse *AbstractRefreshableConfigApplicationContext* sind. Zwar ist dies in der Regel der Fall für Anwendungskontexte, die durch eine *.xml* Datei erzeugt werden, doch kann es nicht garantiert werden. Probleme entstehen, wenn ein generischer Anwendungskontext verwendet wird, wie es auch beim *hybris*-Server der Fall ist. Der *hybris*-Server erzeugt eine eigene Klasse für Anwendungskontexte, den *CloseAwareApplicationContext*[101], welcher keine Methode enthält um die zugehörigen *.xml* Dateien zu identifizieren. Somit kann die Änderung einer *.xml* Datei keinem Anwendungskontext zugeordnet werden und das Plugin kann seinen Zweck nicht erfüllen. Eine Lösung des Problems ist möglich, erfordert jedoch ein tiefes Verständnis von *hybris* und weitere Entwicklungsarbeit, was über den Rahmen der Masterarbeit hinausgehen würde.

Da das *SpringXmlPlugin* nicht auf dem *hybris*-Server funktioniert, wurden die Tests zum Ermitteln der Zeitersparnis im Minimalbeispiel durchgeführt. Dort liegt die durchschnittliche Serverstartzeit bei etwa 4400 ms. Darin enthalten sind die Zeit zum Aufbau des *Servlet*-Kontextes, welche im Schnitt bei ca. 900 ms liegt, und die Zeit zum Aufbau seines Eltern-Kontextes, mit durchschnittlich etwa 910 ms. Somit macht der Aufbau der Anwendungskontexte ca. 40 % der Serverstartzeit aus[102]. Eine Änderung im Eltern-Kontext benötigt durchschnittlich 223 ms für den Aktualisierungsvorgang, was das Aktualisieren beider Kontexte beinhaltet. Das entspricht 5 % der Serverstartzeit bzw. 12 % der Zeit zum Aufbau der Spring Konfiguration beim Serverstart. Daraus lässt sich folgern, dass die Methode *refresh* nicht den kompletten Anwendungskontext neu aufbaut, sondern nur die Änderungen überträgt. Die Hypothese wird durch die Tatsache bekräftigt, dass das Hinzufügen einer *Bean* deutlich mehr Zeit[103] in Anspruch nimmt als beispielsweise das Entfernen eines Kommentars. Eine Änderung im *Servlet*-Kontext braucht im Schnitt 187 ms für die Aktualisierung, was 4 % der Serverstartzeit bzw. 10 % der Zeit zum Aufbau der Spring Konfigurati-

[101] Packet: de.hybris.platform.spring.ctx.CloseAwareApplicationContext
[102] Auf dem *hybris*-Server sind es zwischen 60 und 65 %
[103] Erhöhungen von bis zu 60 % wurden gemessen

on beim Serverstart entspricht. Zusammengefasst kann je nach Größe der Änderung, Anzahl der Kontexte und Anteil der Spring Konfiguration am Serverstart, mit einer Zeitersparnis von 90 bis 98 % im Vergleich zu einem vollständigen Serverneustart gerechnet werden.

6 Fazit

Eine große Schwäche der Programmiersprache Java gegenüber dynamischen Programmiersprachen ist die fehlende Möglichkeit zur Durchführung von Code-Änderungen zur Laufzeit und der damit verbundene Zeitverlust beim *Deployment* von Webanwendungen. Der aktuelle Trend zur Entwicklung von *Hot Deployment* Softwareprodukten soll dieser Schwäche entgegenwirken. Schnell aufgegriffen wurde dieser von dem kommerziellen Produkt *JRebel*, welches bisher den Wettbewerb dominierte. Doch auch *OpenSource* Produkte wurden stets weiterentwickelt und verfügen mittlerweile über eine vergleichbare Funktionalität.

Als Ergebnis der Untersuchung mehrerer *OpenSource* Produkte wurde in der vorliegenden Arbeit die *DCEVM* als einzige angemessene Software identifiziert und zur Evaluation ausgewählt. An ihrem Beispiel wurden die Konzepte von verschiedenen *Hot Deployment* Tools erläutert. Ihre Installation wurde auf dem *hybris*-Server der dotSource GmbH sowie anhand eines Minimalbeispiels dokumentiert. Zudem wurde die Framework-Unterstützung als wichtige zusätzliche Funktionalität von *Hot Deployment* Tools erkannt und es wurde ein selbst entwickeltes Plugin vorgestellt, welches die Unterstützung des Spring Frameworks durch die *DCEVM* erweitert.

Im Kapitel *Evaluation* wurde die *DCEVM* im praktischen Einsatz anhand von Tests und Umfragen mit dem Produkt *JRebel* verglichen und als eine frei verfügbare und erweiterbare Software bewertet, die ihren angestrebten Zweck erfüllt. Dennoch ist die Entscheidung für oder gegen die *DCEVM* von vielen Faktoren abhängig, wie der verwendeten Frameworks oder der Häufigkeit von nicht unterstützten Code-Änderungen. Schließlich wurde das *SpringXmlPlugin* evaluiert und als ein Plugin bewertet, das mit einer Zeitersparnis von 90 bis 98 % eine große Verbesserung darstellt. Da es allerdings nicht wie geplant auf dem *hybris*-Server der dotSource GmbH Anwendung finden kann, ist es für die Firma aktuell nur von begrenztem Nutzen. Eine Behebung des in diesem Zusammenhang auftretenden Problems erfordert zusätzlichen Zeitaufwand. Weitere unvorhergesehene Probleme im Rahmen der Masterarbeit waren die Schwierigkeit der Installation der *DCEVM* auf dem *hybris*-Server der Firma und die fehlende Ausgereiftheit des vom *HotswapAgent* bereitgestellten Spring Plugins.

Offen geblieben ist die Nutzung des *SpringXmlPlugins* in Projekten, die keinen *hybris*-Server verwenden, sowie die Durchführung umfangreicher Praxistests. Zudem gibt es zahlreiche weitere Möglichkeiten die *DCEVM* bzw. den *HotswapAgent* zu erweitern oder sie auf die eigene Anwendung anzupassen. Auch eine Weiterentwicklung

des vom *HotswapAgent* bereitgestellten Spring Plugins wäre denkbar, allerdings sollte dies in Zusammenarbeit mit den Entwicklern geschehen, um Überschneidungen zu vermeiden.

Da sich *OpenSource Hot Deployment* Tools schnell weiterentwickeln, während sich die Verbesserungen von *JRebel* großteils auf die Unterstützung weiterer Frameworks beschränken und die Software zugleich tendenziell teurer wird, geht der Trend immer mehr in Richtung *OpenSource*. Dies ist auch in der engen Zusammenarbeit der *DCEVM* mit der Firma *Oracle* begründet und darin, dass bereits eine Anfrage zur Integration der *DCEVM* in Java[104] gestellt wurde. Eine Umsetzung dieser würde kommerziellen *Hot Deployment* Tools schaden und gleichzeitig die Programmiersprache Java attraktiver machen.

[104] JEP 159

7 Literaturverzeichnis

[ARG11a] Allan-Raundahl-Gregersen: Implications of Modular Systems on Dynamic Updating. In: *Proceedings of the 14th international ACM Sigsoft symposium on component based software engineering*, ACM, 2011. – S. 169-178

[ARG11b] Allan-Raundahl-Gregersen, Bo Norregaard J.: Run-time Phenomena in Dynamic Software Updating: Causes and Effects. In: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on software evolution*, ACM, 2011. – S. 6-15

[ARG12] Allan-Raundahl-Gregersen, Bo Norregaard Jorgensen; H. Hadaytullah; Kai K.: Javeleon: An Integrated Platform for Dynamic Software Updating and its Application in Self-* systems. In: *2012 Spring Congress on Engineering and Technology*, IEEE, 2012. – S. 1-9

[dota] dotSource.GmbH: *Digital Success right from the start.* `https://www.dotsource.de/download/content/1996/16937/dotSource-Flyer.pdf`. – Zugriff: 28.09.2016

[dotb] dotSource.GmbH: *Vision-Mission.* `https://www.dotsource.de/Agentur/Vision-Mission`. – Zugriff: 28.09.2016

[int] internetagenturranking.de: *Subranking 2016: E-Commerce.* `http://www.internetagentur-ranking.de/ranking_2016/sub-9.html`. – Zugriff: 28.09.2016

[ITW] ITWissen: *Dependency Injection.* `http://www.itwissen.info/definition/lexikon/Dependency-Injection-dependency-injection-DI.html`. – Zugriff: 14.12.2016

[JB] Jiri-Bubnik: *Hotswap Agent.* `http://hotswapagent.org/`. – Zugriff: 17.11.2016

[JKa] Jevgeni-Kabanov: Java EE Productivity Report 2011. `https://zeroturnaround.com/rebellabs/java-ee-productivity-report-2011/`. – Zugriff: 28.09.2016

[JKb] Jevgeni-Kabanov: Reloading Java Classes 401: HotSwap and JRebel - Behind the Scenes. `http://zeroturnaround.com/rebellabs/reloading_java_classes_401_hotswap_jrebel/`. – Zugriff: 28.09.2016

[JK12] Jevgeni-Kabanov, Varmo V.: A thousand years of productivity: the JRebel story. In: *Software: Practice and Experience* (2012). – Veröffentlicht in der Wiley Online Library

[JKUL] Johannes-Kepler-Universität-Linz: *Dynamic Code Evolution VM.* `http://ssw.jku.at/dcevm/`. – Zugriff: 14.10.2016

[KSM09] Kathryn S. McKinley, Suriya Subramanian; Michael H.: Dynamic Software Updates: A VM-centric Approach. In: *ACM SIGPLAN NOTICES* (2009). – S. 1-12

[OS] Oleg-Shelajev: How-to guide to writing a javaagent. `http://zeroturnaround.com/rebellabs/how-to-inspect-classes-in-your-jvm/`. – Zugriff: 17.11.2016

[SS] Suriya-Subramanian: *Jvolve.* `http://suriya.github.io/jvolve/`. – Zugriff: 14.10.2016

[TW10] Thomas-Würthinger, Christian Wimmer; Lukas S.: Dynamic Code Evolution for Java. In: *Proceedings of the 8th International Conference on the principles and practice of programming in java*, ACM, 2010. – S. 10-19

[TW11] Thomas-Würthinger: *Dynamic Code Evolution for Java*, Technisch-Naturwissenschaftliche Fakultät an der Johannes Kepler Universität in Linz, Diplom, 2011

[Zera] ZeroTurnaround: *Buy JRebel.* `https://zeroturnaround.com/software/jrebel/buy/`. – Zugriff: 28.09.2016

[Zerb] ZeroTurnaround: *What developers want: An end to App-Server restarts.* `http://files.zeroturnaround.com/pdf/JRebelWhitePaper2014.pdf`. – Zugriff: 28.09.2016