Otto von Guericke Universität Magdeburg
Fakultät für Informatik

Bachelorarbeit

JavAdaptor - vererbungshierarchiebeeinflussende Programmänderungen zur Laufzeit

Verfasser:

Alexander Grebhahn

21. Oktober 2010

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Mario Pukall

Institut für Technische und Betriebliche Informationssysteme (ITI)

Grebhahn, Alexander:
JavAdaptor - vererbungshierarchiebeeinflussende Programmänderungen zur Laufzeit
Bachelorarbeit, Otto von Guericke Universität Magdeburg, 2010.

Inhaltsverzeichnis

Abbildungsverzeichnis **vi**

Tabellenverzeichnis **vii**

Quellcodeverzeichnis **ix**

Abkürzungsverzeichnis **xi**

1 Einleitung **1**

2 Grundlagen **3**
2.1 Laufzeitänderungen in Java 3
2.2 JavAdaptor 4
2.2.1 Nachladen einer Klasse 5
2.2.2 Veränderungen in den aufrufenden Klassen 7
2.3 Benutzte Schnittstellen und Werkzeuge 13
2.4 Vererbung in Java 15

3 Anforderungsanalyse **17**
3.1 Überblick zur Veränderung der Vererbungshierarchie 17
3.2 Vererbungshierarchiebeeinflussende Änderungen von Klassen und abstrakten Klassen 18
3.3 Vererbungshierarchiebeeinflussende Änderungen von Interfaces 20
3.4 Vererbungshierachie versus Caller Updates 21

4 Konzept **25**
4.1 Veränderungen der Vererbungshierarchie von Klassen und Interfaces . . 25
4.2 Nachladen von Klassen 25
4.3 Nachladen eines Interfaces 27
4.4 Caller Updates 28
4.4.1 Proxyerstellung für ein Interface 28
4.4.2 Superklassen und ihre abgeleiteten Klassen als aufrufende Klassen 29
4.4.3 Interfaces als aufrufende Klassen 30

5 Implementierung **33**
5.1 Programmablauf - Überblick 33
5.2 Subklassen zu den nachzuladenden Klassen hinzufügen 35

5.3 Nachladen einer Klasse 36
5.4 Zustandsübernahme 37
5.5 Anpassung der Aufrufenden Klassen mittels Expression Editor 39
5.5.1 Zugriffe auf Felder vom Typ der Nachgeladenen Klasse 39

6 Evaluierung **43**
6.1 Aufbau der Testfälle 43
6.2 Vererbungshierarchieänderung an einer Klasse 44
6.3 Vererbungshierarchieänderung an einem Interface 46
6.4 Existierende Vererbungshierarchie in den aufrufenden Klassen 48
6.5 Existierende Vererbungshierarchie im aufrufenden Interface 50

7 Zusammenfassung und Ausblick **53**

A Anhang **57**

Literaturverzeichnis **59**

Abbildungsverzeichnis

2.1 Veränderung der Funktionsweise einer Applikation mittels JavAdaptor. 5
2.2 Umbenennung einer Klasse. 6
2.3 Lokale Aufrufe auf die nachgeladene Klasse. 7
2.4 Globale Felder vom Typ der nachgeladenen Klasse. 8
2.5 Parameter und Rückgaben vom Typ der nachgeladenen Klasse. 10
2.6 Nachladen einer aufrufenden Klasse. 11
2.7 Nötige Veränderungen vor dem Laden einer Klasse. 12
2.8 Expression Editor von Javassist. 14
2.9 Private Felder der Superklasse. 16

3.1 Explizite Vererbungshierarchieänderungen. 18
3.2 Vererbungshierarchie einer Klasse. 18
3.3 Vererbungshierarchie eines Interfaces 20
3.4 Superklasse und abgeleitete Klasse als Aufrufer. 21
3.5 Superinterface und abgeleitete Interfaces als Aufrufer. 22

4.1 Superklassen vor ihren abgeleiteten Klassen Nachladen. 26
4.2 Veränderung der Vererbungshierarchie. 26
4.3 Interface Nachladen. 27
4.4 Gegenüberstellung: Proxies für ein Interface und eine Klasse. 28
4.5 Superklasse und abgeleitete Klasse als Aufrufer. 29
4.6 Interface als Aufrufer einer nachgeladenen Klasse. 30

5.1 Programmablauf von JavAdapor. 34
5.2 Abgeleitete Klassen zur Liste der nachzuladenden Klassen hinzufügen. . 36

5.3 Redefinieren mittels HotSwap. 37
5.4 Zustandsübernahme 38
5.5 Änderung der Feldzugriffe mittels Expression Editor. 40

6.1 Aufbau der Testfälle - CallerCaller. 44
6.2 Vererbungshierarchieänderung einer Klasse. 44
6.3 Quelltextänderungen - Vererbungshierarchieänderung einer Klasse. . . . 45
6.4 Vererbungshierarchieänderung eines Interfaces. 46
6.5 Quelltextänderungen - Vererbungshierarchieänderung eines Interfaces. . 47
6.6 Vererbungshierarchie in den aufrufenden Klassen. 48
6.7 Quelltextänderungen - Vererbungshierarchie in den aufrufenden Klassen. 49
6.8 Vererbungshierarchie im aufrufenden Interfaces. 50
6.9 Quelltextänderungen - Vererbungshierarchie im aufrufenden Interfaces. 51

A.1 Oberfläche der Testumgebung. 58

Tabellenverzeichnis

2.1 Durch HotSwap unterstützte Änderungen an Klassen. 4
2.2 Zustandsübertragung bei Veränderungen von Feldern. 6

Quellcodeverzeichnis

2.1 Bytecodeveränderungen mit der Javassist Quellcode API. 14
2.2 Vererbung in Klassen. 15
2.3 Vererbung in abstrakten Klassen. 15
2.4 Vererbung in Interfaces. 15
A.1 Abgeleitete Klassen zur List der nachzuladenden Klassen hinzufügen. . 57
A.2 Testumgebung - Quellcode der Klasse MyFrame. 58

Abkürzungsverzeichnis

JDI	Java Debug Interface
JPDA	Java Platform Debugger Architecture
JVM	Java Virtual Machine

1. Einleitung

Das Entwickeln neuer Software ist ein kontinuierlicher Prozess, der nicht mit der Fertigstellung der ersten laufenden Version abgeschlossen ist. Es müssen Fehler behoben, neue Funktionen eingebaut und bestehende Funktionen verbessert werden. Das kann nicht immer vor dem produktiven Einsatz der Software geschehen.

Ist eine neue Version dieser Software erschienen, so besteht das Problem, dass die laufende Applikation beendet, die Updates eingespielt und dann das Programm neu gestartet werden muss. Das kann bei Applikationen, die annähernd 24 Stunden am Tag und 7 Tage in der Woche laufen müssen, zu großen Problemen führen. Da durch einen Ausfall solcher Systeme sehr hohe Kosten entstehen können. Es wäre daher praktisch, wenn die Änderungen im laufenden Betrieb eingespielt werden könnten.

In Applikationen, die in dynamischen Programmiersprachen wie z. B. Python [Hug97] geschrieben wurden, kann dies zur Laufzeit geschehen. Da es sich bei Java jedoch um keine dynamische Programmiersprache handelt, sondern statisch typisiert wird, müssen Java Applikationen, wenn Veränderungen an ihnen durchgeführt werden sollen, immer beendet und neu gestartet werden. Da es sich nach [TIO10] bei Java um die am häufigsten benutzte Programmiersprache der Welt handelt und auch viele Applikationen mit hoher Verfügbarkeit wie zum Beispiel *Apache Tomcat* oder *JBoss Application Server* in Java geschrieben wurden, wäre es von Vorteil, wenn auch Java Programme während der Laufzeit verändert werden könnten.

Zu diesem Zweck wurde das Programm JavAdaptor entwickelt, mit dem auch an Java Applikationen umfassende Veränderungen während der Laufzeit durchgeführt werden können.

Zielstellung der Arbeit

Das Ziel dieser Arbeit besteht darin, das bestehende JavAdaptor Konzept dahingehend zu erweitern, das vererbungshierarchieverändernde Programmänderungen während der Laufzeit einer Applikation durchgeführt werden können.

Gliederung der Arbeit

In Kapitel 2 wird das bestehende JavAdaptor Konzept vorgestellt, mit dem Änderungen an Klassen während der Laufzeit durchgeführt werden können. Außerdem werden die verwendeten APIs beschrieben und ein Überblick über die in Java existierenden Arten der Vererbung gegeben. Nachdem dies geschehen ist, werden in Kapitel 3 die Anforderungen gezeigt, die nötig sind, um vererbungshierarchiebeeinflussende Programmänderungen während der Laufzeit durchzuführen. Des Weiteren werden noch Besonderheiten gezeigt, die beachtet werden müssen, wenn eine Vererbung in den aufrufenden Klassen einer nachgeladenen Klasse existiert. In Kapitel 4 wird dann beschrieben, wie das JavAdaptor Konzept erweitert werden muss, um Vererbungen sowohl in den aufrufenden Klassen als auch in den nachgeladenen Klassen zu unterstützen. Ein Teil der Implementierungen, die dazu nötig sind, werden in Kapitel 5 gezeigt. Um die Korrektheit der beschriebenen Erweiterung zu überprüfen, wird in Kapitel 6 eine Evaluierung durchgeführt. In den beschriebenen Testszenarien wird sowohl untersucht, ob Vererbungshierarchieänderungen korrekt durchgeführt werden können, als auch, ob Klassen korrekt nachgeladen werden können, wenn eine Vererbung in ihren aufrufenden Klassen existiert. Anschließend wird diese Arbeit in Kapitel 7 zusammengefasst und es wird ein Ausblick über weitere mögliche Forschungsbereiche gegeben.

2. Grundlagen

Im Gegensatz zu dynamischen Programmiersprachen wie zum Beispiel Python, Ruby oder Smalltalk [PKG+09] ist es bei Java nicht vorgesehen, ganze Programmteile von Applikationen während der Laufzeit zu aktualisieren. Dieses Kapitel stellt dar, wo die Grenzen von Laufzeitänderungen in Java liegen und warum es mittels JavAdaptor möglich ist, Laufzeitänderungen auch über diese Grenzen hinaus durchzuführen.

2.1 Laufzeitänderungen in Java

Zum besseren Verständnis, welche Laufzeitänderungen in Java möglich sind und wo die Grenzen sind, ist es nötig, zu wissen, wie die Bestandteile eines Programms in der Java Virtual Machine (JVM), der Java Laufzeitumgebung, gespeichert werden.
Die Speicherung der Programmzustände erfolgt im *heap* und in einer *method area* [LY99]. Dabei ist der *heap* für die Speicherung objektspezifischer und die *method area* für die Speicherung klassenspezifischer Informationen zuständig.

Wird eine Instanz einer Klasse erzeugt, so wird Speicher innerhalb des *heap* für die Speicherung dieser Instanz zur Verfügung gestellt. Existieren keine Verweise auf eine Instanz, so wird sie vom *garbage collector* gelöscht und der Speicherplatz für die Speicherung anderer Instanzen freigegeben. Bei den klassenspezifischen Informationen, die im Methodenbereich gespeichert werden, handelt es sich unter anderem um Typinformationen über die Klasse, den Konstantenpool der Klasse, Methodeninformationen und Feldinformationen [Ven99]. Diese Informationen werden gespeichert, wenn die Klasse, zu der sie gehören, unter Zuhilfenahme eines Klassenladers in die Applikation geladen wird. Dies muss nicht zum Startzeitpunkt der Applikation geschehen, sondern erst, wenn die Klasse das erste Mal benötigt wird. Dieser Mechanismus nennt sich *lazy classloading* [LB98]. Im weiteren Verlauf dieser Arbeit werden diese Klasseninformationen nach [GJSB05] als Referenztyp der Klasse bezeichnet. Da keine Möglichkeit besteht, den Referenztyp einer Klasse zu verändern, bleibt nur die Möglichkeit, den alten Referenztyp einer Klasse aus der JVM zu löschen und die veränderte Klasse neu in die JVM zu laden. Das Entladen eines Referenztypen geht jedoch nur, wenn keine Verweise auf

den Klassenlader der Klasse und auf jede Klasse, die durch ihn geladen wurde, existieren. Zusätzlich zu der Klasse, die aus der Applikation gelöscht werden soll, werden dann auch alle anderen Klassen des Klassenladers gelöscht. Soll dies mit einer Klasse geschehen, die durch den Applikations-Klassenlader geladen wurde, käme das einem Applikationsstopp gleich.

Java HotSwap. In den ersten Java Versionen war es nicht möglich, Änderungen an Klassen in die Applikation zu übernehmen. Seit Java 1.4 gibt es die JVM Erweiterung *HotSwap* [Dmi01], mit der es möglich ist, Methoden und Konstruktorbäuche einer Klasse während der Laufzeit zu verändern. Obwohl es sich bei HotSwap nicht um eine Standardfunktionalität handelt, existiert dieses Feature in den meisten JVMs, wie zum Beispiel in der HotSpot JVM, der Oracle JRockit und der IBM VM.

Gibt es in einer Klasse Änderungen, die sich nicht auf einen Methodenbauch beschränkt, so muss die Applikation um diese Änderungen zu übernehmen beendet und neu gestartet werden. In Tabelle 2.1 ist ein Teilüberblick über mögliche Änderungen, die an einer Klasse durchgeführt werden können gegeben. Zusätzlich wird gezeigt, welche von ihnen durch HotSwap unterstützt werden. Da nur ein kleiner Teil der möglichen Änderungen mittels HotSwap durchgeführt werden können, besteht noch erheblicher Handlungsbedarf. Die Änderungen, die nicht durch HotSwap unterstützt werden, werden im Nachfolgenden als schemabeeinflussende Änderungen bezeichnet.

Veränderte Konstrukte	durch HotSwap unterstützt
Klassensignatur	-
Felddeklaration	-
Methodendeklaration	-
Konstruktordeklaration	-
Methodenbauch	✓
Konstruktorbauch	✓

Tabelle 2.1: Durch HotSwap unterstützte Änderungen an Klassen.

2.2 JavAdaptor

Bei JavAdaptor handelt es sich um ein Eclipse[1] Plug-In, mit dem es möglich ist, auch schemabeeinflussende Änderungen an Klassen durchzuführen, ohne dabei die Applikation beenden zu müssen. Um diese Funktionalität bereitzustellen, benötigt *JavAdaptor* eine JVM, die HotSwap unterstützt.

In Abbildung 2.1 auf der nächsten Seite wird ein Überblick über die Architektur des Nachladevorgangs im Zusammenhang mit der Eclipse-IDE angegeben. Nachdem eine Applikation gestartet wurde, kann der Entwickler wie gewohnt über die Eclipse-IDE Änderungen am Quellcode der Applikation durchführen. Hat der Entwickler sich dazu entschlossen, diese Änderungen in die laufende Applikation zu übernehmen, so verbindet sich JavAdaptor mit der JVM der Applikation und lädt durch einen eigenen

[1] http://www.eclipse.org/

Thread, der in die Applikation beim Start eingefügt wurde, die veränderten Klassen. Nachdem der Nachladeprozess abgeschlossen ist, wird der Thread beendet, um keine Laufzeiteinbußen zu erzeugen [PKG+09]. Ist dies geschehen, kann die Applikation mit neuer Funktionalität weiter ausgeführt werden.

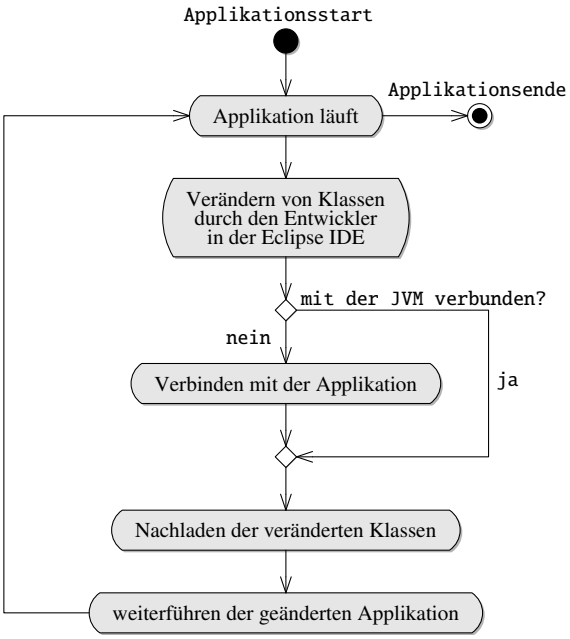

Abbildung 2.1: Veränderung der Funktionsweise einer Applikation mittels JavAdaptor.

2.2.1 Nachladen einer Klasse

Soll eine Klasse mit verändertem Klassenschema nachgeladen werden, so kann dies nicht mittels HotSwap geschehen, da durch HotSwap nur schemaerhaltende Änderungen unterstützt werden. Damit eine Klasse mit veränderten Schema in die JVM geladen werden kann, wird von JavAdaptor ein neuer Referenztyp für der Klasse erstellt. Dies kann entweder durch das Laden der Klasse unter Zuhilfenahme eines anderen Klassenladers oder durch ein Umbenennen der Klasse geschehen.

Wird zum Laden der veränderten Klasse ein anderer Klassenlader genommen, so führt dies zu Laufzeiteinbußen. Diese werden dadurch verursacht, dass zwischen Klassen, die durch verschiedene Klassenlader geladen wurden, mittels Reflexion API kommuniziert werden muss [Caz04]. Das ist ungleich langsamer als die direkte Kommunikation zwischen Klassen. Daher wird in JavAdaptor die neue Version einer nachzuladenden Klasse mit einer Versionsnummer versehen.

Ein Beispiel für die Umbenennung ist in Abbildung 2.2 auf der nächsten Seite zu sehen.

Ein Beispiel für die Umbenennung ist in Abbildung 2.2 auf der nächsten Seite zu sehen, in der die Methode `doSomething()` zu der Klasse `Callee` hinzugefügt wurde, weshalb es eine Schemaänderung der Klasse `Callee` gab. Durch das Umbenennen der Klasse kann zum Laden der gleiche Klassenlader genommen werden, durch den auch die alte Version der Klasse geladen wurde.

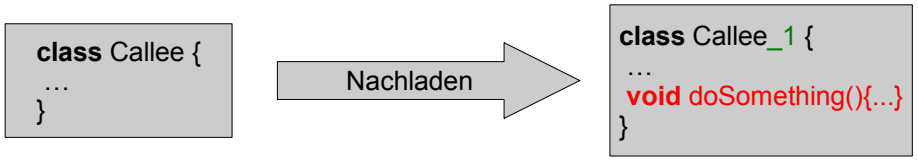

Abbildung 2.2: Umbenennung einer Klasse.

Zustandsübernahme

Da für die veränderte Klasse ein neuer Referenztyp erstellt wurde, ist es notwendig, die Werte statischer Felder in die neue Klasse zu übernehmen. Existieren von der Klasse Instanzen, so müssen auch die Zustände dieser Instanzen auf Instanzen der neuen Klassenversion übertragen werden.

Dabei wird für jede Instanz der alten Klasse eine mittels JavAdaptor eine Instanz der neuen Klasse erstellt. Die Zustände der alten Instanzen werden dann paarweise auf die Instanzen der neuen Klassenversion übernommen. Dabei ist darauf zu achten, dass Instanzen nicht nur Felder besitzen, die in ihrer Klasse deklariert wurden, sondern auch Felder ihrer Superklassen. Die Werte dieser Felder müssen auch in die neu erstellten Instanzen übernommen werden.

Gab es Änderungen in den Feldern der nachgeladenen Klasse, müssen diese von JavAdaptor berücksichtigt werden. In Tabelle 2.2 ist ein Überblick über die möglichen Veränderungen in den Feldern einer Klasse und deren Behandlung mittels JavAdaptor gegeben.

	automatisch	durch den Entwickler
neues Feld hinzugekommen	✓(default Wert)	✓
Feldname hat sich verändert	-	✓
Typ hat sich verändert	-	✓
Basis Typ	✓	-
komplexer Typ	-	✓
Feld gelöscht	✓	-

Tabelle 2.2: Zustandsübertragung bei Veränderungen von Feldern.

Sind der Typ und der Name eines Feldes gleich geblieben, so kann der Wert dieses Feldes von der neuen Klasse (für statische Felder) beziehungsweise von den Instanzen der neuen Klasse (für nicht statische Felder) übernommen werden.

Wurde in der neuen Klassenversion ein Feld hinzugefügt, so muss dieses Feld, in den von JavAdaptor erstellten Instanzen der neuen Klasse, mit einem instanzspezifischen

Default Wert initialisiert werden. Da dieser Wert nicht mittels JavAdaptor ermittelt werden kann, muss er durch den Entwickler gesetzt werden. Handelt es sich bei dem Feld um ein statisches Feld, so wird ihm beim Laden der neuen Klassen, der Wert zugewiesen, der im statischen Konstruktor für dieses Feld definiert ist.

Wenn der Name oder der Typ eines Feldes verändert worden ist, so muss der Entwickler eine Mapping Funktion definieren.

Die Zustandsübertragung wird, aus Performance-Gründen, direkt im durch JavAdaptor hinzugefügten Thread der Applikation durchgeführt. Dabei können auch die Werte von `private` beziehungsweise `final` Feldern übertragen werden.

2.2.2 Veränderungen in den aufrufenden Klassen

Nachdem im vorangegangen Abschnitt erläutert wurde, wie die nachzuladende Klasse unter neuem Namen in die Applikation geladen wird, um eine Schemaänderung an ihr durchzuführen, ist in diesem Abschnitt beschrieben, wie ihre aufrufenden Klassen verändert werden müssen, um die neue Version der nachgeladenen Klasse zu verwenden. Ohne Aktualisierung der Aufrufer, wird weiterhin die alte Klassenversion referenziert und das Nachladen der veränderten Klasse hätte keine Auswirkungen auf die weitere Programmausführung.

Bei den Änderungen in den aufrufenden Klassen ist darauf zu achten, dass es nicht zu einer Änderung ihrer Schemata kommt. Würde dies geschehen, so müssten diese Klassen auch unter neuem Namen in die JVM geladen werden und ihr aufrufenden Klassen müssten so verändert werden, dass sie ihre neue Version benutzen. Im schlimmsten Fall hätte das ein Nachladen aller Klassen der Applikation zur Folge, was einem Stopp der Applikation gleichkäme.

Um dies zu verhindern, gibt es in JavAdaptor zwei Konzepte, durch die es möglich ist, alle nötigen Veränderungen in den aufrufenden Klassen ohne Schemaänderung durchzuführen und diese Änderungen dann mittels HotSwap in die Applikation zu übernehmen.

Lokale Aufrufe auf die nachgeladene Klasse

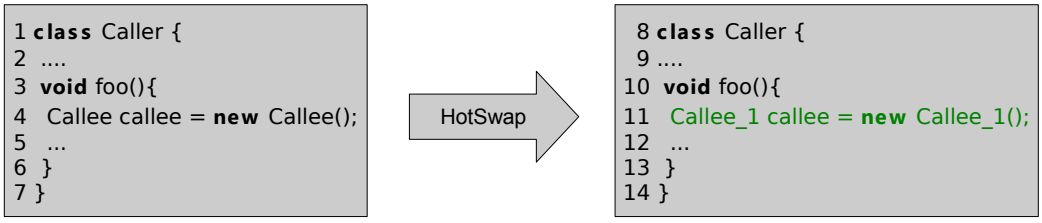

Abbildung 2.3: Lokale Aufrufe auf die nachgeladene Klasse.

Existieren in einer Klasse nur lokale Referenzen auf die nachgeladene Klasse, wie in Abbildung 2.3 auf der vorherigen Seite gezeigt, so können diese, durch Referenzen auf die neue Version der Klasse ersetzt werden. Wie in Zeile 11 in Abbildung 2.3 auf der vorherigen Seite zu sehen ist, reicht es aus, alle Vorkommen der Klasse `Callee` durch die Klasse `Callee_1` auszutauschen. Da sich das Schema der aufrufenden Klasse `Caller` dadurch nicht verändert hat, kann sie mittels HotSwap aktualisiert werden. Bei der nächsten Methodenausführung wird dann mit der neuen Klassenversion (`Callee_1`) gearbeitet.

Globale Felder vom Typ der nachgeladenen Klasse

Existieren in einer aufrufenden Klasse globale Felder vom Typ der nachgeladenen Klasse, wie in Abbildung 2.4 zu sehen, so können nicht einfach alle Referenzen der alten Version der nachgeladenen Klasse auf ihre neue Version aktualisiert werden. Für das Beispiel würde eine einfache Veränderungen aller Referenzen bedeuten, dass der in Zeile 2 gezeigt Quelltext von `Callee callee` zu `Callee_1 callee` verändert werden würde. Dadurch käme es zu einer Schemaänderung in der Klasse `Caller`.

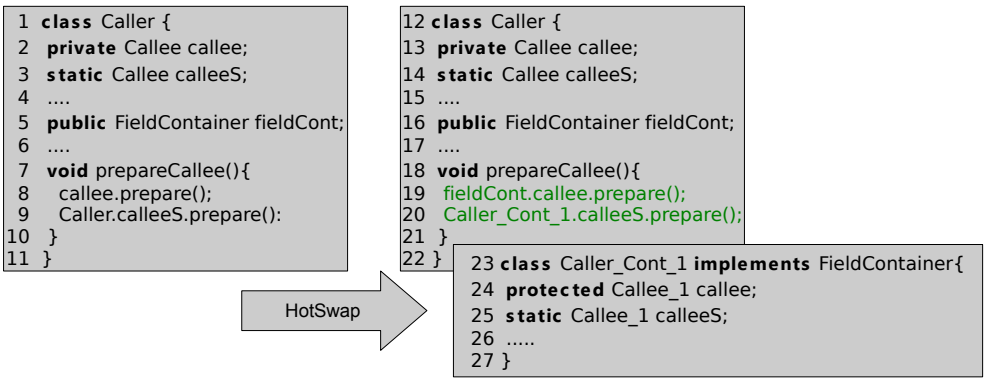

Abbildung 2.4: Globale Felder vom Typ der nachgeladenen Klasse.

Damit Klassen, die die Felder vom Typ einer nachgeladenen Klasse deklarieren, nicht nachgeladen werden müssen, um mit der neuen Version der nachgeladenen Klasse zu arbeiten, werden diese Felder in neu erstellte Klassen ausgelagert. Dies gilt sowohl für statische, als auch für nicht statische Felder vom Typ der nachgeladenen Klasse. Die Klassen, in die die Felder ausgelagert werden, werden im nachfolgenden FieldContainer Klassen genannt. Bei einer dieser neu erstellten Klasse handelt es sich in Abbildung 2.4 um die Klasse `Caller_Cont_1` (Zeile 23 bis 27), in welche die Felder `callee` und `calleeS` der Klasse `Caller` mit aktualisiertem Typ (`Callee_1`) ausgelagert werden (Zeile 24,25).

Nach der Erstellung der Klasse `Caller_Cont_1` müssen die Aufrufe der Felder, die von der Klasse `Caller` in die Klasse `Caller_Cont_1` ausgelagert wurden, insoweit verändert werden, dass nicht die Felder der Klasse `Caller`, sondern die der Klasse `Caller_Cont_1` verwendet werden.

Ist die Sichtbarkeit eines der Felder, dass in die Klasse `Caller_Cont_1` verschoben wurde, durch den Modifier `private` auf die Klasse in der es deklariert ist beschränkt, so muss dieser Modifier auf `protected` gesetzt werden, damit das Feld in der Klasse `Caller` benutzt werden kann. Für das Beispiel aus Abbildung 2.4 auf der vorherigen Seite bedeutet dies, dass der Modifier des Feldes `callee` verändert wurde.

Handelt es sich bei einem der ausgelagerten Felder um ein statisches Feld, so können die Aufrufe diese Feldes über die Klasse `Caller_Cont_1` geschehen (Zeile 20), da bei statischen Aufrufen die FieldContainer Klasse als neuer Halter des statischen Feldes fungiert.

Da auch Aufrufe auf nicht statische Felder unterstützt werden sollen, muss eine Instanz der Klasse `Caller_Cont_1` von jeder Instanz der Klasse `Caller` gehalten werden. Damit jeder Instanz der Klasse `Caller` eine `Caller_Cont_1` Instanz zugeordnet werden kann, muss zu der Klasse `Caller` ein Feld hinzugefügt werden (Zeile 5,16). Dies geschieht schon vor Programmstart, da es sonst zu einer Schemaänderung in der Klasse `Caller` kommt, weshalb sie unter neuem Namen nachgeladen werden müsste. Das hinzugefügte Feld ist vom Interface Typ `FieldContainer`, welches von der Klasse `Caller_Cont_1` implementiert wird. Durch diese Schnittstelle, ist es möglich, beliebig oft verschiedene Felder der Klasse in eine andere Klasse, die das Interface FieldContainer implementiert auszulagern und diese dann bei Aufrufen auf die ausgelagerten Felder anzusteuern, ohne das Schema der Klasse `Caller` zu verändern.

Damit auch zur Laufzeit auf nicht statischen Feldern, die in den FieldContainer ausgelagert wurden, gearbeitet werden kann, muss in jedem Konstruktor der Klasse `Caller` ein Statement hinzugefügt werden. Durch dieses Statement wird eine Instanz der Klasse `Caller_Cont_1` erzeugt und dann dem FieldContainer Feld der Caller Instanz zugewiesen.

Da auch bereits existierende Instanzen der aufrufenden Klasse Aufrufe auf nicht statische ausgelagerte Felder über das `FieldContainer` Feld ausführen müssen, wird mittels JavAdaptor für jede dieser Instanzen eine Instanz der Klasse `Caller_Cont_1` erstellt und ihr dann auch zugewiesen. Den ausgelagerten Feldern werden dann die zuvor erstellten Instanzen der `Callee_1` Klasse zugewiesen.

Parameter und Rückgaben vom Typ der nachgeladenen Klasse

Existieren in der aufrufenden Klasse einer ersetzten Klasse Methoden, wie zum Beispiel `showCallee()` oder `newCallee(...)` (Abbildung 2.5 auf der nächsten Seite), die einen Parameter vom Typ der alten Klassenversion besitzen oder deren Rückgabetyp vom Typ dieser Klasse ist, so besteht das Problem, dass innerhalb der Methoden mit der neuen Klassenversion gearbeitet wird und gleichzeitig Instanzen der alten Klassenversion übergeben beziehungsweise zurückgegeben werden müssen (Abbildung 2.5 auf der nächsten Seite Zeile 11 bis 20).

Dieses Problem wird von JavAdaptor durch **Proxies** gelöst. Beim Proxy handelt es sich nach [GHJV94] um einen Platzhalter oder Stellvertreter. Wird eine Klasse nachgeladen,

so wird für sie eine Proxy Klasse erstellt, durch die es möglich ist, Instanzen der neuen Klassenversion an Methoden zu übergeben, oder von Methoden zurückgeben zu lassen, wenn eine Instanz der alten Klassenversion verlangt ist. Dies wird erreicht, indem der Proxy von der alten Klassenversion erbt. Somit ist er typkompatible zu ihr. Gleichzeitig hält er eine Instanz der neuen Klassenversion. In Abbildung 2.5 in den Zeilen 32 bis 41 sind Ausschnitte des Proxy gezeigt, der zwischen den Klassen `Callee` und `Callee_1` vermittelt.

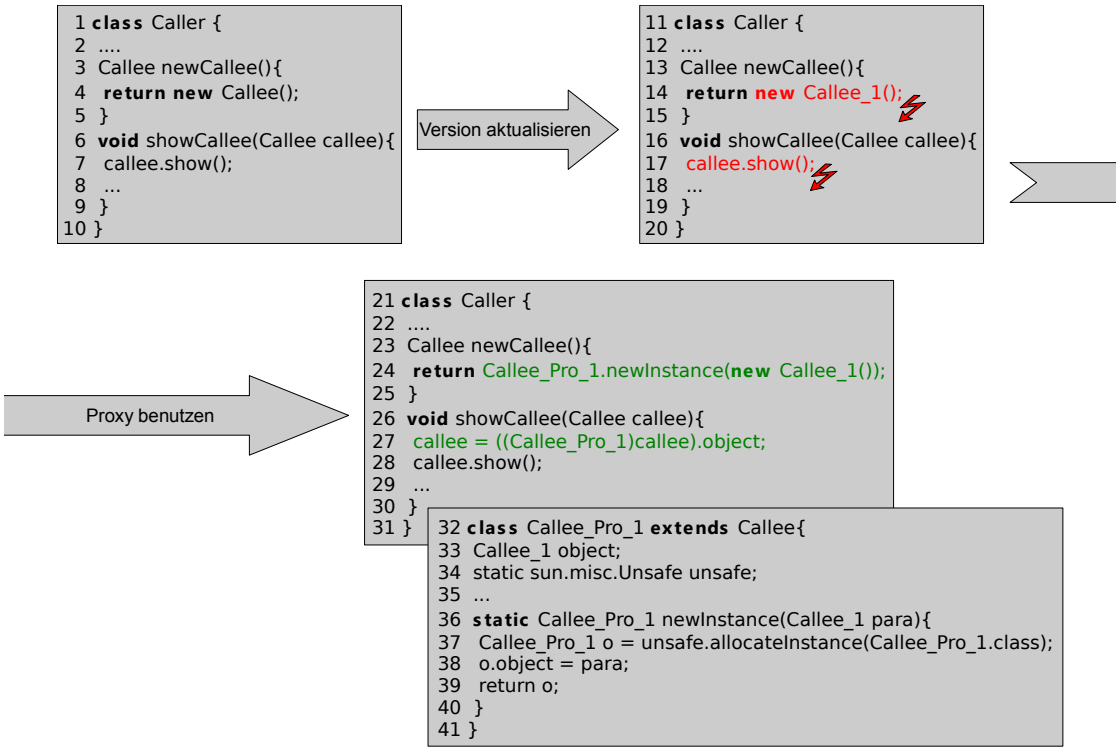

Abbildung 2.5: Parameter und Rückgaben vom Typ der nachgeladenen Klasse.

Dadurch, dass der Proxy von der alten Version der Klasse erbt, wird automatisch in jedem Konstruktor der Klasse `Callee_Pro_1`, wenn nicht anders angegeben der Default Konstruktor der Klasse `Callee` aufgerufen. Um beim Instantiieren der Proxy Klasse keine Seiteneffekte zu verursachen, die durch Aufrufe von Konstruktoren der Superklassen entstehen, wird dies über die Methode `newInstance(...)` vollzogen. Von ihr wird eine neu erstellte `Callee_Pro_1` Instanz zurückgegeben, die unter Zuhilfenahme der Klasse `sun.misc.Unsafe`[2] erstellt wurde. Dabei handelt es sich um eine von Oracle nicht dokumentierte Klasse, mit der es möglich ist, ohne Konstruktoraufruf Speicherbereich im *heap* zu allokieren.

Wird nach dem Einbauen des Proxy die Methode `newCallee(...)`(Zeile 23 - 25 Abbildung 2.5) aufgerufen, wird in ihr die Methode `newInstance(...)` (Zeile 36 - 40) aufgerufen. An diese Methode wird die Instanz der neuen Klassenversion übergeben, die sonst zurückgegeben werden würde. Von der Methode `newInstance(...)` wird dann

[2]http://www.docjar.com/docs/api/sun/misc/Unsafe.html

eine Instanz der Klasse `Callee_Pro_1` erstellt an die Methode `newCallee(...)` zurückgegeben. Diese Methode gibt dann die an sie gegebene Instanz der Klasse `Callee_Pro_1` zurück.

Wird eine Proxy Instanz als Parameter an eine Methode übergeben, so muss die in den Proxy eingepackte Instanz der neuen Klassenversion entpackt werden. Ist dies geschehen, so kann auf dem Parameter gearbeitet werden. Ein Beispiel dafür ist in Abbildung 2.5 auf der vorherigen Seite (Zeile 27 und 28) gezeigt. Da an jeder Stelle, an der die Methode aufgerufen wird, das als Parameter an die Methode übergebene Objekt in eine Proxy Instanz eingepackt wurde, kann das Entpacken des Instanz aus dem Proxy in der Methode stets vollzogen werden.

Proxies erlauben es also die Rückgabe und Übergabe von Objekten der neuen Klassenversion ohne den Parameter oder Rückgabetyp von Methoden auf Aufruferseite und somit das Schema der aufrufenden Klassen zu verändern.

Nachladen einer aufrufenden Klasse

Müssen Änderungen an einer Applikation durchgeführt werden, so ist es selten der Fall, dass sich die Änderungen auf eine Klasse beschränken. Häufiger kommt es vor, dass Klassen, die miteinander interagieren auch gemeinsam geändert werden müssen.

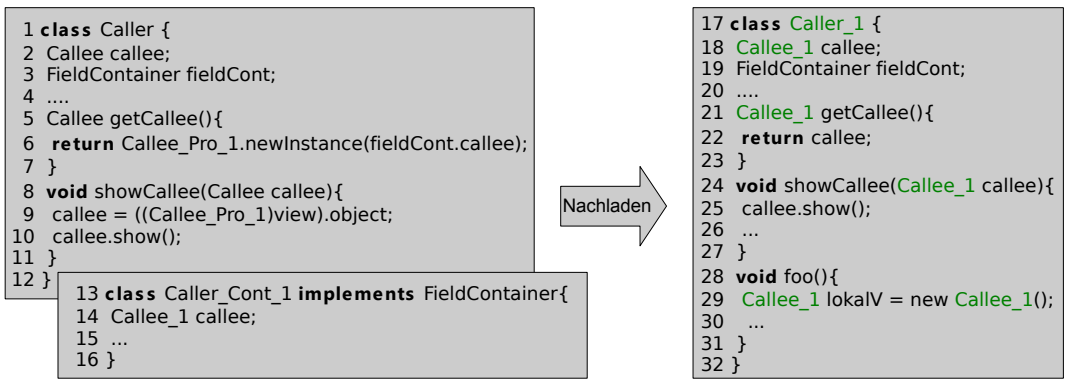

Abbildung 2.6: Nachladen einer aufrufenden Klasse.

Nachdem im vorherigen Abschnitt anhand der Beispielklasse `Caller` erläutert wurde, wie aufrufende Klassen unter Beibehaltung des Klassenschemas aktualisiert werden, wir in diesem Abschnitt beschrieben, was zu beachten ist, wenn die aufrufende Klasse selbst ersetzt werden muss.

Ein Beispiel dafür ist in Abbildung 2.6 zu sehen. In der Abbildung wurde die Klasse `Callee` schon zu einem früheren Zeitpunkt durch die Klasse `Callee_1` ersetzt. Seit diesem Zeitpunkt muss die Klasse `Caller`, dank des Feldes `callee`(Zeile 2) und der Methoden

`getCallee()`(Zeile 5-7) und `showCallee()`(Zeile 8-11) sowohl ihren `Fieldcontainer` (Zeile 13-16), als auch den `Proxy` der Klasse `Callee` benutzen.

Wird nun die Klasse `Caller`, aufgrund einer Schemaänderung, mit einer neuer Version in die Applikation geladen, so werden vorher in ihr alle Referenzen von bereits nachgeladenen Klassen durch Referenzen auf ihre neuesten Versionen ersetzt. Dies kann sowohl für lokale als auch für globale Referenzen geschehen, wie in Abbildung 2.6 auf der vorherigen Seite in Zeile 18,21,24 und 29 zu sehen, bei denen die Referenzen von `Callee` zu `Callee_1` verändert wurden.

Anschließend müssen auch die aufrufenden Klassen der Klasse `Caller` aktualisiert werden. Dies geschieht dann wie in den vorangegangen Abschnitten beschrieben.

Vorkehrungen vor dem Zeitpunkt des Klassenladens

Damit der beschriebene Ansatz zum Aktualisieren laufender Programmfunktionalität auch an allen Klassen einer Applikation durchgeführt werden kann, müssen einige Veränderungen an den Klassen schon vor dem Zeitpunkt des Ladens der Klasse in die JVM durchgeführt werden. Abbildung 2.7 zeigt die nötigen Veränderungen anhand eines Beispiels.

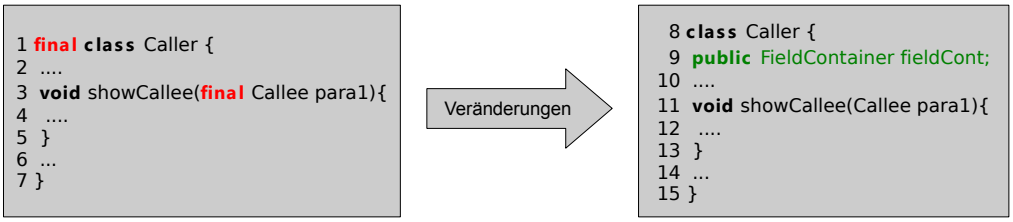

Abbildung 2.7: Nötige Veränderungen vor dem Laden einer Klasse.

Fieldcontainer. Um nach dem Nachladen einer Klasse, in ihren aufrufenden Klassen Felder vom Typ der nachgeladenen Klasse benutzen zu können, muss zu jeder Klasse ein `Fieldcontainer` Feld hinzugefügt werden (Zeile 9). Die Verwendung diese Felder ist in Abschnitt 2.2.2 auf Seite 8 beschrieben.

Finale Klassen. Besitzt eine Klasse der Applikation den Modifier `final`, so muss dieser aus der Klassendefinition entfernt werden. Wird dies nicht gemacht, so kann, wenn die Klasse nachgeladen wird, für sie kein `Proxy` erstellt werden, da Klassen, die den Modifier `final` besitzen nicht abgeleitet werden können.

Final Parameter. Sobald ein Parameter einer Methode den Modifier `final` besitzt (Zeile 5 in Abbildung 2.7), kann sein Wert in der Methode nicht verändert werden. Ist die Klasse `Callee` nachgeladen worden, so muss an die Methode `foo(Callee callee)` eine Proxy Instanz übergeben werden, aus der die Instanz der neuen Klassenversion

ausgepackt werden muss. Da diese Instanz an die Speicherstelle des Parameters gespeichert wird, müssen `final` Modifier aus den Parameterlisten entfernt werden. Geschieht dies nicht, so kommt es schon beim Compilieren der veränderten Klassenversion, in der Proxies für die Parameter verwendet werden zu Fehlern.

Diese Veränderungen müssen nicht nur vor dem Programmstart an allen Klassen der Applikation durchgeführt werden, sondern auch an allen Klassen, die durch JavAdaptor nachgeladen werden.

2.3 Benutzte Schnittstellen und Werkzeuge

Im folgenden Abschnitt werden die Klassenbibliotheken vorgestellt, die in JavAdaptor benutzt werden. Dabei handelt es sich um das *Java Debug Interface (JDI)* und um *Javassist* [CN03], [Chi00].

Java Debug Interface (JDI)

Das JDI ist ein Teil der Java Platform Debugger Architecture (JPDA), es bildet das Interface für Debugger Applikationen [jdk08]. Unter Zuhilfenahme der JDI ist es möglich, sich mit einer bestehenden JVM zu verbinden und auf dieser dann Operationen durchzuführen. Dabei kann es sich um das Aufrufen von Methoden, das Erstellen von Instanzen und das Setzen von Werten sowohl statischer, als auch nicht statischer Felder handeln. Des Weiteren kann des Redefinieren von Klassen durch das Interface mittels HotSwap eingeleitet werden.

Darüber hinaus können Informationen über den Zustand der JVM abgefragt werden. So kann zum Beispiel erfragt werden, wie viele Instanzen von einer Klasse existieren, von welchen Objekten diese Instanzen gehalten werden und welche Werte die Felder dieser Instanzen besitzen.

Javassist

Mit `Javassist` ist es möglich, bestehende class Dateien zu verändern oder neue zu erstellen. Dazu werden dem Benutzer zwei APIs zur Verfügung gestellt, mit denen der Bytecode verändert werden kann.

Dabei handelt es sich zum einen um die Quellcode API, durch die Veränderungen an den class Dateien, unter Vorgabe von Quellcode und damit auch ohne Kenntnisse vom Java Bytecode vorgenommen werden können. Außerdem existiert eine Bytecode API, mit der ohne Abstraktionsschicht direkt Bytecodeanweisungen eingefügt oder verändert werden können.

In Listing 2.1 auf der nächsten Seite ist gezeigt, wie mittels der Javassist Quellcode API ein Methodenbauch verändert werden kann. Dazu wird ein `ClassPool` Objekt benötigt (Zeile 2 Listing 2.1 auf der nächsten Seite), der Verweise auf alle Klassen der Applikation beinhaltet.

```
1 public void setBody(String className) throws Exception{
2  ClassPool cp = ClassPool.getDefault();
3  CtClass ctclass = cp.get(className);
4  CtMethod ctMethod = ctclass.getMethods()[0];
5  ctMethod.setBody("{System.out.println("text");}");
6  ctclass.writeFile();
7 }
```

Listing 2.1: Bytecodeveränderungen mit der Javassist Quellcode API.

Um den Methodenbauch einer Methode zu verändern, muss die Klasse, in der die Methode deklariert ist, aus dem `Classpool` Objekt geholt werden, wie in Zeile 3 zu sehen. Dann kann auf die Methode zugegriffen werden (Zeile 4), um in Zeile 5 den Bauch der Methode dann durch den angegebenen String zu ersetzen, bevor die veränderte Klasse gespeichert wird (Zeile 6).

Sollen Änderungen nur bei spezifischen Bytecodeanweisungen durchgeführt werden, wie zum Beispiel bei Methodenaufrufen oder Feldzugriffen, gibt es in `Javassist` die Möglichkeit, *Expression Editoren* zu definieren und diese dann auf Methoden anzuwenden. In Abbildung 2.8 ist ein Beispiel dafür gezeigt. Da der Expression Editor Bytecode verändert, ist der Bauch der Methode `method()` (Zeile 11 bis 16) als dekompilierter Bytecode dargestellt.

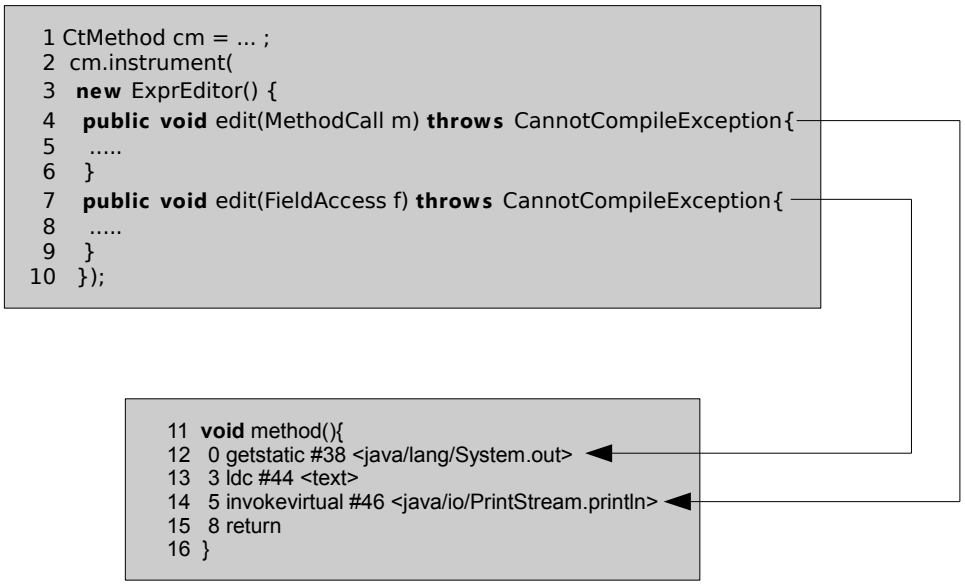

Abbildung 2.8: Expression Editor von Javassist.

Mittels der Anweisung `cm.instrument()` (Zeile 2 - 10) wird der Inhalt der Beispielmethode `method()` (Zeile 11 - 16) durch den Expression Editor (Zeile 3-10) geparst.

Existiert in der Methode ein Feldaufruf wie in Zeile 12, so wird für diese Anweisung die `edit(FieldAccess f)` Methode (Zeile 7 - 9) aufgerufen. In dieser Methode ist definiert, wie der gefundene Feldaufruf modifiziert werden soll. Für Methodenaufrufe wird in diesem Zusammenhang die Methode `edit(MethodCall f)` aufgerufen.

Um in einer `edit()` Methode eines Expression Editors auf die einzelnen Bestandteile der zu verändernden Anweisung zugreifen zu können, wurde eine eigene Syntax definiert, die in [Chi10] beschrieben ist. Auf diese Syntax wird in Abschnitt 5.5 auf Seite 39 genauer eingegangen.

2.4 Vererbung in Java

In Java wurde sowohl das Konzept der strikten Einfachvererbung für Klassen und abstrakte Klassen, als auch das Konzept der Mehrfachvererbung für Interfaces umgesetzt.

```
1 class Subklasse extends Superklasse {
2  ....
3 }
```

Listing 2.2: Vererbung in Klassen.

```
1 abstract class SubAbstKlasse extends Superklasse {
2  ....
3 }
```

Listing 2.3: Vererbung in abstrakten Klassen.

```
1 interface Subinterface extends Superinterface1, Superinterface2 {
2  ....
3 }
```

Listing 2.4: Vererbung in Interfaces.

Dabei ist zu beachten, dass (abstrakte) Klassen nur von anderen (abstrakten) Klassen erben können und Interfaces nur von anderen Interfaces. Außerdem können Klassen und abstrakte Klassen noch beliebig viele Interfaces implementieren.

Wird eine Klasse von einer anderen Klasse abgeleitet, so wird sie im Weiteren als Superbeziehungsweise als Basisklasse bezeichnet. Die abgeleitete Klasse wird in diesem Zusammenhang direkte Subklasse genannt. Erbt eine Klasse von dieser Subklasse, so ist sie eine indirekte Subklasse bezüglich der Superklasse ihrer Superklasse. Ein Interface, das von einem oder mehreren Interfaces abgeleitet wird, wird entsprechend dazu als Superinterface bezeichnet und abgeleitete Interfaces als Subinterfaces.

In jedem Konstruktor einer abgeleiteten Klasse existiert am Anfang des Konstruktors ein Aufruf, durch den ein Konstruktor der Superklasse aufgerufen wird. Wurde durch den Entwickler kein Konstruktor explizit angegeben, so handelt es sich dabei um den Default Konstruktor der Superklasse. Die Konstruktoraufrufe setzen sich rekursiv fort, bis ein Konstruktor der Klasse `Objekt` aufgerufen wurde. Wird eine Instanz einer Klasse erstellt, so wird im *heap* der JVM für die Felder dieser Instanz Speicherplatz bereitgestellt. Dies geschieht für alle nicht statischen Felder, die in der Klasse von deren Typ

die Instanz ist deklariert sind und um alle nicht statischen Felder der direkten und indirekten Superklassen.

Überschreiben von Methoden und Überlagern von Attributen

Wie schon erwähnt, besitzen Instanzen von Subklassen nicht nur Speicherplatz für die nicht statischen Felder, die in ihrer Klasse definiert wurden, sondern auch für die nicht statischen Felder, die ihre Superklassen bereitstellen. Dies gilt auch für Felder der Superklasse, deren Sichtbarkeit durch den `private` Modifier beschränkt ist. Wird ein Feld oder eine Methode mit gleichem Namen und Typ beziehungsweise gleicher Signatur sowohl in der Superklasse als auch in der abgeleiteten Klasse definiert, so ist das Feld oder die Methode der Superklasse überschrieben. Wird ein Feld sowohl in der Superklasse, als auch in der Subklasse deklariert, so ist das Feld der Superklasse überlagert [Ull09]. Wenn ein nicht statisches Feld überlagert worden ist, besitzen Instanzen der abgeleiteten Klasse beide Felder.

Ein Beispiel, wie auf einem `private` Feld der Superklasse gearbeitet werden kann, auch wenn es überlagert ist, ist in Abbildung 2.9 zu sehen. In der Superklasse ist in Zeile 2 ein `private` Feld definiert, das in der `public` Methode `foo()`benutzt wird. Wird die Methode `foo()` nicht von der abgeleiteten Klasse überschrieben, so wird in Zeile 17 die Methode `foo()` der `Supercallee` (Zeile 1-8) aufgerufen. Dabei wird dann auf dem `privateValue` Feld der Klasse `Supercallee` gearbeitet.

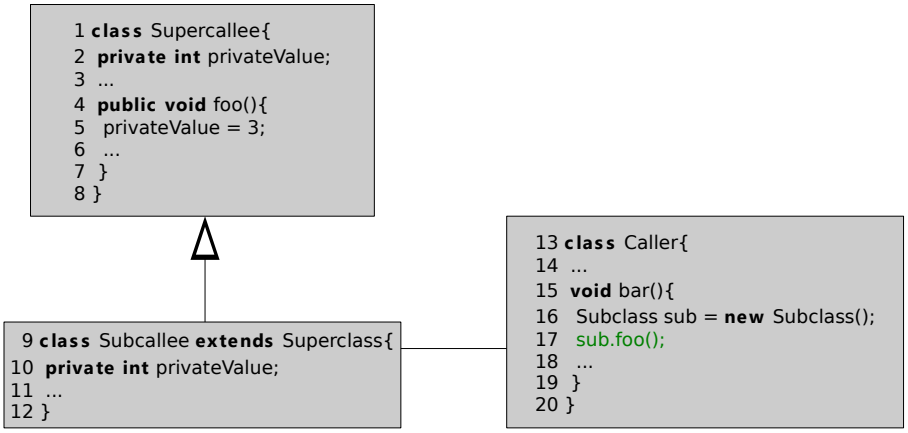

Abbildung 2.9: Private Felder der Superklasse.

3. Anforderungsanalyse

Im vorangegangen Kapitel wurde die Basisfunktionalität von JavAdaptor aufgezeigt, mit der es möglich ist, das Schema von Klassen während der Laufzeit zu verändern. In diesem Kapitel wird nun beschrieben, welche Anforderungen erfüllt sein müssen, wenn vererbungshierarchiebeeinflussende Programmänderungen durchgeführt werden sollen. Dies umfasst sowohl das Nachladen von Klassen und abstrakten Klassen, die von anderen Klassen abgeleitet werden, als auch das Nachladen von Interfaces. Außerdem wird noch analysiert, was beachtet werden muss, wenn eine Vererbungshierarchie in den aufrufenden Klassen einer nachgeladenen Klasse existiert.

3.1 Überblick zur Veränderung der Vererbungshierarchie

Mit JavAdaptor soll es möglich sein, umfassende Änderungen der Vererbungshierarchie während der Laufzeit durchzuführen. Um zu zeigen, welche Fälle dabei auftreten können, ist in Abbildung 3.1 auf der nächsten Seite die Signatur einer Klasse `A`, einer abstrakten Klasse `C` und eines Interfaces `IB` gezeigt.

Bei den zu unterstützenden Änderungen handelt es sich sowohl um einen Austausch der Superklasse(n), als auch um ein Verändern der zu implementierenden Interfaces.

So muss es bei selbstgeschriebenen Klassen `A` und abstrakten Klassen `C` möglich sein, die Superklasse `X` mit jeder anderen Klasse zu ersetzen. Da ein Interface von mehreren Interfaces erben kann, muss bei einem selbstgeschriebenen Interface `IB` die Möglichkeit gegeben werden, während der Laufzeit zusätzlich von einem beliebigen Interface `IY` erben zu können oder ein beliebiges Interface `IZ` aus der Vererbungshierarchie zu entfernen. Außerdem sollen Änderungen in den implementierenden Interfaces von Klassen und abstrakten Klassen unterstützt werden.

Diese Änderungen müssen den Vorgaben der Sprache Java entsprechen. Damit ist es nicht möglich, das zwei oder mehr Klassen gegenseitig, direkt oder indirekt von einander erben.

Abbildung 3.1: Explizite Vererbungshierarchieänderungen.

Grundsätzlich gilt, dass Änderungen an der Vererbungshierarchie das Schema der jeweiligen Klasse ändern. Daher müssen Klassen und Interfaces durch eine neue Version ersetzt werden, wenn es eine Veränderung in ihrer Vererbungshierarchie gibt.

3.2 Vererbungshierarchiebeeinflussende Änderungen von Klassen und abstrakten Klassen

Sollen Vererbungshierarchieänderungen an Klassen und abstrakten Klassen während der Laufzeit durchführt werden, ist es nötig, sowohl Superklassen als auch Subklassen nachladen zu können und deren Vererbung zu ändern. Um zu zeigen, welche Fälle dabei auftreten können und worauf geachtet werden muss, werden anhand des Beispiels in Abbildung 3.2 alle Möglichkeiten für Hierarchieänderungen diskutiert.

In Abbildung 3.2 handelt es sich um die Klasse `SubCallee` die von der Klasse `SuperCallee` erbt. Im Folgenden wird gezeigt, was beachtet werden muss, wenn diese Klassen aufgrund einer Schemaänderung nachgeladen werden müssen.

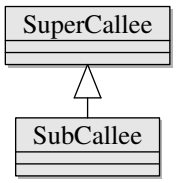

Abbildung 3.2: Vererbungshierarchie einer Klasse.

Hierarchieänderungen an Blattklassen. Wird die Vererbungshierarchie der Klasse `SubCallee` aus Abbildung 3.2 verändert, so kann dies durch das Austauschen der Superklasse `SuperCallee` mit einer anderen Klasse oder durch das Entfernen oder Hinzufügen eines implementierenden Interfaces geschehen. Dadurch kommt es zu einer Schemaänderung der Klasse. Deshalb wird es nötig diese Klasse unter neuem Namen in die Applikation zu laden. Dies muss wie in Abschnitt 2.2.1 auf Seite 5 beschrieben, geschehen.

Existieren zu diesem Zeitpunkt Instanzen der Klasse, so müssen die Zustände dieser Instanzen auf Instanzen der neuen Klassenversion paarweise übertragen werden. Dabei

ist zu beachten, dass es Änderungen an den Feldern, die in der Klasse deklariert sind, gegeben haben kann. Wurde die Superklasse durch eine andere Klasse ersetzt, so muss außerdem noch beachtet werden, dass die Felder, die in der alten Superklasse deklariert wurden, kein Gegenstück in der neuen Vererbungshierarchie der Klasse besitzen. Außerdem können die Felder, die in der neuen Superklasse deklariert werden, nicht automatisch mit einem Wert belegt werden. Sie müssen mit einem Default Wert belegt werden.

Nachdem die nachzuladende Klasse unter neuem Namen geladen wurde und die Zustände ihrer Instanzen von Instanzen der neuen Klassenversion übernommen wurden, ist es noch nötig ihre aufrufenden Klassen dahingehend zu verändern, dass sie die neue Version der Klasse benutzen. Dabei muss darauf geachtet werden, dass es zu keiner Schemaänderung in einer aufrufenden Klasse kommt. Um dies zu gewährleisten, müssen gegebenenfalls FieldContainer und Proxies erstellt und verwendet werden.

Hierarchieänderungen an Superklassen. Gibt es eine Änderung in der Vererbungshierarchie der Klasse `SuperCallee`, so ist es nötig, diese Klasse unter neuem Namen in die Applikation zu laden. Außerdem muss dafür gesorgt werden, dass alle abgeleiteten Klassen von der neuen Klassenversion erben. Da dies nur durch eine Schemaänderung geschehen kann, müssen zusätzlich zu der veränderten Klasse noch alle abgeleiteten Klassen ausgetauscht werden.

Für das Beispiel aus Abbildung 3.2 auf der vorherigen Seite bedeutet dies, dass bei einer schemabeeinflussenden Änderung von `SuperCallee` auch die Klasse `SubCallee` nachgeladen werden muss.

Wird dies nicht getan, entsteht das Problem, dass die Klasse `SubCallee` nicht typkonform mit der neuen Version ihrer Superklasse ist. Das bedeutet, dass zum Beispiel, keine Instanzen der Klasse `SubCallee` an Methoden übergeben werden können, die Instanzen ihrer Superklasse als Parameter verlangen.

Wurden die Klassen `SuperCallee` und `SubCallee` unter neuem Namen in die Applikation geladen, so müssen die Zustände existierender Instanzen der alten Klassenversionen paarweise von Instanzen der neuen Klassenversionen übernommen werden. Dabei muss darauf geachtet werden, dass es Veränderungen in den Feldern der veränderten Klasse gegeben haben kann, die adäquat behandelt werden müssen. Da Instanzen der abgeleiteten Klasse auch die nicht statischen Felder ihrer Superklassen besitzen, müssen diese Veränderungen auch bei der Zustandsübernahme der Instanzen der angeleiteten Klassen beachtet werden. In Abbildung 3.2 auf der vorherigen Seite gilt für Instanzen der Klasse `SubCallee`.

Werden die Aufrufenden Klassen der Klasse `SuperCallee` dahingehend angepasst, dass sie die Klasse `SuperCallee_1` verwenden, so kann dies wie in Abschnitt 2.2.1 auf Seite 5 beschrieben geschehen. Da die Klasse `SubCallee` von der Klasse `SuperCallee` erbt, handelt es sich bei ihr auch um eine aufrufende Klasse. Da sie jedoch genau wie ihre Superklasse mit neuer Version in die Applikation geladen werden muss, können in ihr alle Referenzen der alten Superklasse durch Referenzen auf die neue Version der Superklasse ersetzt werden.

Verallgemeinerung. Muss eine Klasse aufgrund einer Schemaänderung unter neuen Namen in die Applikation geladen werden, so muss dies auch mit ihren direkten und indirekten Subklassen geschehen.

3.3 Vererbungshierarchiebeeinflussende Änderungen von Interfaces

Im Gegensatz zu Klassen, die nur direkt von einer Klasse abgeleitet werden, können Interfaces von beliebig vielen Interfaces ableitet werden. In diesem Abschnitt wird beschrieben, was beachtet werden muss, wenn die Vererbungshierachie eines Interfaces verändert wird. Dazu ist in Abbildung 3.3 eine mögliche Vererbungshierachie von Inteaces gezeigt.

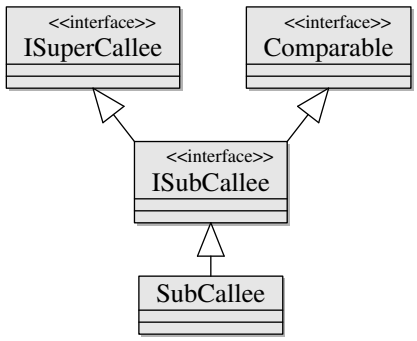

Abbildung 3.3: Vererbungshierarchie eines Interfaces

Hierarchieänderungen an Subinterfaces. Wird die Vererbungshierarchie des in Abbildung 3.3 gezeigten Interfaces `ISubCallee` verändert, so muss es, aufgrund der daraus resultierenden Schemaänderung dieses Interfaces, unter neuem Namen in die Applikation geladen werden. Außerdem muss auch die Klasse `SubCallee` unter neuem Namen nachgeladen werden, damit sie die neue Version des Interfaces implementiert.

Da alle Felder die in einem Interface deklariert wurden, sowohl `final` als auch `static` sind, gibt es keine Möglichkeit, ihre Werte zur Laufzeit zu verändern. Wird der Wert eines dieser Felder durch den Entwickler geändert, so ist der aktuellste Wert nicht in der Applikation, sondern in den veränderten class Dateien. Damit darf keine Zustandsübernahme für Felder gemacht werden, die in einem Interface deklariert wurden.

Nachdem das Interface `ISubCallee` und die Klasse `SubCallee` nachgeladen wurden, müssen ihre aufrufenden Klassen, wie in Abschnitt 2.2.2 auf Seite 7 beschrieben, aktualisiert werden, damit sie die neue Klassenversion referenzieren.

Hierarchieänderungen an Superinterfaces. Wird die Vererbungshierarchie des Interfaces `ISuperCallee` verändert, so müssen dieses Interface, das Interface `ISubCallee` und die Klasse `SubCallee` unter neuem Namen nachgeladen werden. Dabei erfolgt das Nachladen der Interface und der Klasse wie bereits beschrieben (Abschnitt 2.2.1 auf Seite 5).

Verallgemeinerung. Wird ein Interface aufgrund einer Schemaänderung unter neuem Namen in die Applikation geladen, so muss dies auch mit allen direkten und indirekten Subinterfaces geschehen. Das gleiche gilt für alle Klasse, die eines der nachgeladenen Interfaces implementieren.

3.4 Vererbungshierachie versus Caller Updates

In diesem Abschnitt wird dargestellt, worauf geachtet werden muss, wenn innerhalb der aufrufenden Klassen der nachgeladenen Klasse eine Vererbung existiert. In Abbildung 3.4 ist ein Beispiel dafür gegeben, in dem die Klassen `SuperCaller` und `SubCaller` als aufrufende Klassen der nachzuladenden Klasse `Callee` fungieren.

Bei der Behandlung der Referenzen der nachgeladenen Klasse wird unterschieden, ob es sich nur um lokale Referenzen handelt, ob die nachgeladene Klasse in einer Methoden oder Konstruktorsignatur auftaucht oder ein Feld vom Typ der nachgeladenen Klasse in der aufrufenden Klasse deklariert ist.

Gibt es lokale Referenzen der nachgeladenen Klasse in den aufrufenden Klassen, so können diese genau wie in Abschnitt 2.2.1 auf Seite 5 beschrieben mit Referenzen auf die neue Klassenversion ersetzt werden. Dies gilt sowohl für Methoden der Super- als auch der Subklassen.

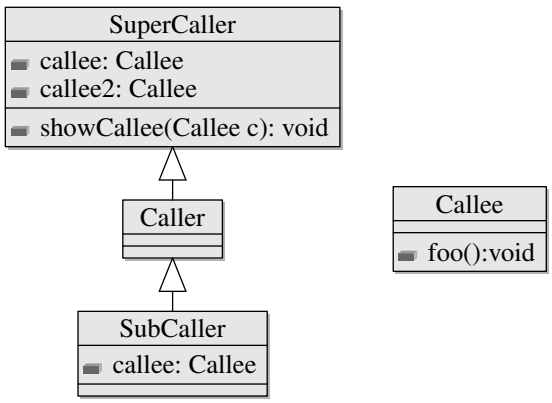

Abbildung 3.4: Superklasse und abgeleitete Klasse als Aufrufer.

Wie schon in Abschnitt 2.2.2 auf Seite 8 gezeigt, müssen alle Klassen, die globale Felder vom Typ der nachzuladenden Klasse deklarieren, eine `FieldContainer` Klasse bekommen, in die diese Felder dann ausgelagert werden. Dabei handelt es sich in diesem Beispiel um die Klassen `SuperCaller` und `SubCaller` für deren Felder die Klassen `SuperCaller_Cont_1` und `SubCaller_Cont_1` erstellt werden.

Aufgrund der Tatsache, dass Instanzen abgeleiteter Klassen auch nicht statische Felder besitzen, die in den Superklassen definiert wurden (Abschnitt 2.4 auf Seite 15), handelt es sich bei der Klasse `Caller` auch um eine aufrufende Klasse der Klasse `Callee`, deren Instanzen Felder der Klasse `Callee` besitzen. Anders als bei den Klassen `SuperCaller` und `SubCaller` muss für sie keine eigene `FieldContainer` Klasse erstellt werden, da sie selbst kein Feld vom Typ `Callee` deklariert.

Da Instanzen der Klasse `SubCaller` sowohl auf das `callee` Feld der Klasse `SuperCaller` als auch auf das `callee` Feld der Klasse `SubCaller` zugreifen können, muss eine Möglichkeit gefunden werden, die es erlaubt, auf diese Felder, nachdem sie in eine `FieldContainer` Klasse ausgelagert wurden, separat zuzugreifen.

Gibt es in den aufrufenden Klassen Methoden, in deren Signatur die nachgeladene Klasse verwendet wird, oder deren Rückgabetyp vom Typ der nachgeladenen Klasse ist, so müssen in dieser Methode Proxies benutzt werden (Abschnitt 2.2.2 auf Seite 9). Dies gilt für alle Methoden mit der nachgeladenen Klasse als Parameter- oder Rückgabetyp. Wurde die Methode in einer Superklasse definiert und in einer abgeleiteten Klassen überschrieben, so müssen sowohl in der Superklasse als auch in der abgeleiteten Klasse Proxies verwendet werden.

Interfaces als aufrufende Klassen

Handelt es sich bei den aufrufenden Klassen, der nachgeladenen Klasse um Interfaces, wie in Abbildung 3.5 zu sehen ist, so müssen diese Interfaces auch dahingehend angepasst werden, dass die neue Version der nachgeladenen Klasse referenziert wird, ohne dabei eine Schemaänderung in dem Interface zu verursachen.

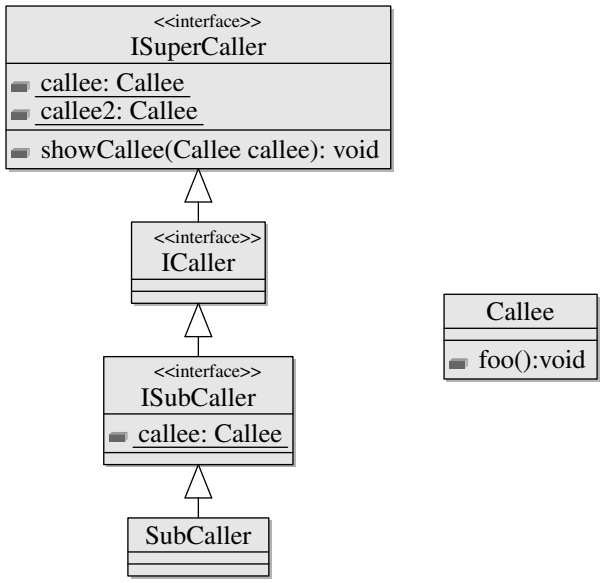

Abbildung 3.5: Superinterface und abgeleitete Interfaces als Aufrufer.

Ein Interface kann nur als Aufrufer einer nachgeladenen Klasse fungieren, wenn die nachgeladene Klasse in einer Methode des Interfaces als Parameter- oder Rückgabetyp existiert oder ein Feld von dem Typ der nachgeladenen Klasse in dem Interface deklariert wurde. Da in einem Interface nur die Signatur und der Rückgabetyp einer Methode definiert wird und keine Implementierung der Methoden existiert, kann es keine lokalen Verwendungen der nachgeladenen Klasse geben. Eine Aktualisierung lokaler Referenzen muss daher nicht betrachtet werden.

Existieren in einem Interface Felder vom Typ der nachgeladenen Klasse, so müssen diese Felder, wie bei aufrufenden Klassen in eine FieldContainer Klasse ausgelagert werden. Für Abbildung 3.5 auf der vorherigen Seite bedeutet das, dass dann für die Interfaces `ISuperCaller` und `ISubCaller` FieldContainer Klassen erstellt werden muss.

Wurde in einem Interface eine Methode definiert, die einen Parameter vom Typ der Nachgeladenen Klasse besitzt oder deren Rückgabetyp von dieser Klasse ist, wie zum Beispiel die Methode `showCallee(...)` des Interfaces `ISuperCaller` (Abbildung 3.5 auf der vorherigen Seite), so werden erst in der Implementierung der Methode Proxies genutzt. Dies muss in den implementierenden Klassen des Interfaces geschehen.

4. Konzept

Nachdem im vorangegangenen Kapitel die Anforderungen analysiert wurden, die erfüllt werden müssen, um vererbungshierarchiebeeinflussende Programmänderungen während der Laufzeit durchzuführen, wird in diesem Kapitel beschrieben, wie das bestehende JavAdaptor Konzept erweitert werden muss, um diese Anforderungen zu erfüllen.

4.1 Veränderungen der Vererbungshierarchie von Klassen und Interfaces

Wurde die Vererbungshierarchie einer Klasse oder eines Interfaces verändert, so verändert sich automatisch das Schema. Damit muss sowohl bei einer Schemaänderung als auch bei einer Veränderung der Vererbungshierarchie die Klasse unter neuem Namen und damit auch mit neuem Referenztypen in die Applikation geladen werden.

4.2 Nachladen von Klassen

Gibt es eine schemabeeinflussende Änderung in einer Superklasse, so müssen alle direkten und indirekten Subklassen dieser Klasse auch unter neuem Namen in die Applikation geladen werden (Kapitel 3 auf Seite 17).

Nachladereihenfolge. Damit keine unnötigen Klassen erstellt werden und diese dann in die JVM geladen werden, muss das Nachladen der Superklasse vor dem Nachladen der abgeleiteten Klassen geschehen. Wird eine Subklasse vor ihrer Superklasse ausgetauscht, so findet keine Hierarchieänderung in der Subklasse statt, da sowohl in der alten, als auch in der neuen Version der Subklasse die alte Superklasse verwendet wird. Um von der neuen Version der Superklasse zu erben, ist es nötig, die Subklasse nochmal, nachdem ihre Superklasse in die JVM geladen wurde, mit neuer Version zu laden. Ein Beispiel dafür ist in Abbildung 4.1 auf der nächsten Seite zu sehen, in dem die Klasse

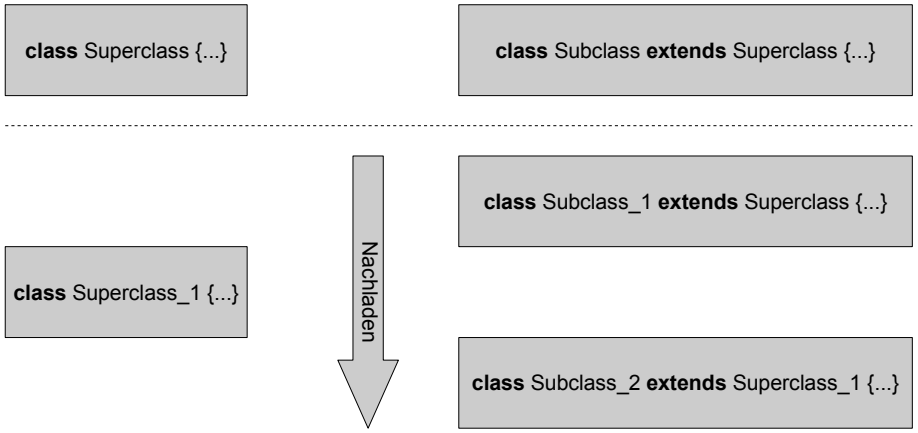

Abbildung 4.1: Superklassen vor ihren abgeleiteten Klassen Nachladen.

`Subclass` mehrmals mit neuer Version erstellt wird, um von der neuesten Version ihrer Superklasse zu erben.

Zustandsübernahme bei Hierarchieänderungen. Wird die Vererbungshierarchie der Klasse `MyGUIObject` aus Abbildung 4.2 durch den Austausch der Superklasse verändert, muss sie unter neuem Namen nachgeladen werden. Existieren zu diesem Zeitpunkt Instanzen dieser Klasse, so müssen die Zustände dieser Instanzen paarweise in Instanzen der neuen Klassenversion übernommen werden. Dabei muss darauf geachtet werden, dass die Zustandsübernahme jedoch nicht für die Zustände aller Felder der Instanzen vollzogen werden kann. Dies kommt daher da durch den Austausch der Superklasse für Instanzen der Klasse `MyGUIObject` neue Felder hinzugekommen oder vorhandene Felder entfernt worden sein können.

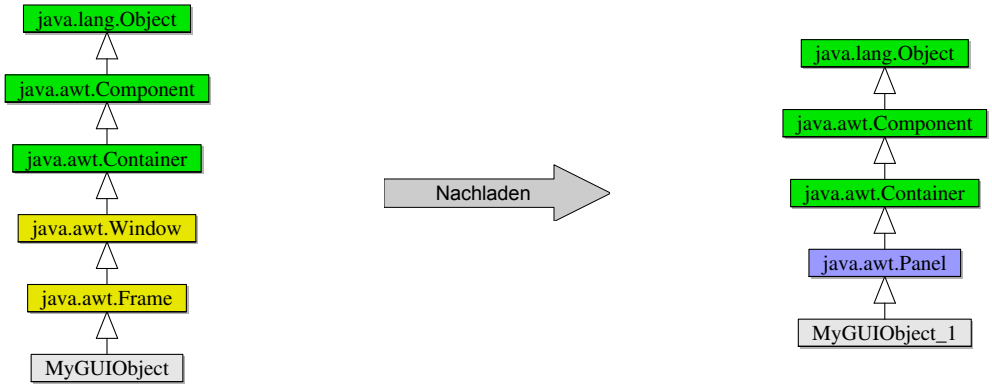

Abbildung 4.2: Veränderung der Vererbungshierarchie.

Ein Zustandsübernahme kann also nur für die Felder durchgeführt werden, deren deklarierende Klasse sowohl in der alten, als auch in der neuen Vererbungshierarchie der

veränderten Klasse existiert. Da sowohl die alte Superklasse (`java.awt.Frame`) indirekt, als auch die neue Superklasse (`java.awt.Panel`) direkt der veränderten Klasse `MyGuiObject` von der Klasse `java.awt.Container` erben, können die Zustände der Felder, die in der Klasse `java.awt.Container` deklariert wurden von Instanzen der Klasse `MyGUIObject` auf Instanzen vom Typ `MyGUIObject_1` paarweise übertragen werden. Das gleiche gilt auch für die Zustände nicht statischer Felder, die in den Klassen `java.awt.Component` und `java.lang.Object` deklariert wurden.

Aufgrund der Tatsache, dass die Klassen `java.awt.Window` und `java.awt.Frame` kein Gegenstück in der neuen Vererbungshierarchie der Klasse `MyGUIObject` besitzen, müssen die Werte der durch sie deklarierten Felder nicht übernommen werden. Bei der Klasse `java.awt.Panel` ergibt sich eine andere Problematik. Da die Felder, die in ihr deklariert wurden, für existierende Instanzen nicht automatisch mit Werten belegt werden können, muss dies durch den Entwickler geschehen.

4.3 Nachladen eines Interfaces

Soll ein Interface mit verändertem Schema in die Applikation geladen werden, so kann dies wie bei Klassen geschehen. Dabei wird das Interface umbenannt und unter neuem Namen in die Applikation geladen. Ein Beispiel für diese Umbenennung ist in Abbildung 4.3 gegeben, in dem das Interface `ICallee` zu `ICallee_1` umbenannt wird.

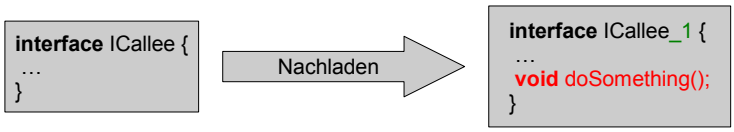

Abbildung 4.3: Interface Nachladen.

Nachdem die neue Version des Interfaces erstellt und in die Applikation geladen wurde, müssen alle aufrufenden Klassen und Interface dahingehend angepasst werden, dass sie die neue Version benutzen, ohne dabei ihr Schema zu ändern. Gibt es nur lokale Verwendungen des Interfaces, so können alle Referenzen der alten Interfaceversion durch Referenzen der neuen Version ersetzt werden. Existiert ein Feld von Typ des nachgeladenen Interfaces, so muss ein `FieldContainer` für die aufrufende Klasse erstellt werden und bei einer Methode mit Parameter oder Rückgabetyp vom Typ des Nachgeladenen Interfaces eine `Proxy` Klasse für das Interface.

Wird das veränderte Interface jedoch von einem anderen Interface erweitert, so kann die Anpassung, dass die neue Version des Interfaces benutzt wird, nur durch eine Schemaänderung im implementierenden Interface vollzogen werden. Damit müssen, wenn ein Interface nachgeladen wird, alle direkten und indirekten Subinterfaces auch unter neuer Version in die JVM geladen werden. Das gleich gilt auch für alle Klasse, die das veränderte Interface oder eines der Subinterfaces implementieren.

Dabei ist wie beim Nachladen von Superklassen und ihren abgeleiteten Klassen darauf zu achten, dass das Nachladen des Superinterfaces vor dem Nachladen der abgeleiteten Interfaces und der implementierenden Klassen geschieht.

4.4 Caller Updates

Ist eine Klasse oder ein Interface unter neuem Namen in die Applikation geladen worden, so müssen alle aufrufenden Klassen und Interfaces dahingehend verändert werden, dass sie die neue Version der Klasse verwenden. Dabei kann es nötig sein, sowohl eine Proxy Klasse für die nachgeladenen Klasse, oder FieldContainer Klassen für die aufrufenden Klassen zu erstellen.

4.4.1 Proxyerstellung für ein Interface

Wird ein nachgeladenes Interface in einer Methode als Parameter oder Rückgabetyp verwendet, so muss ein **Proxy** für das Interface erstellt werden. Dies muss jedoch nur geschehen, wenn die Methode in einer Klasse deklariert wurde, die selbst nicht nachgeladen werden muss.

```
1 interface ICallee{
2  ...
3  void foo();
4  int bar();
5  ...
6 }
```

```
 7 class ICallee_Proxy_1 implements ICallee{
 8  ICallee_1 object;
 9  ...
10  void foo(){}
11  int bar(){return 0;}
12 }
```

```
13 class Callee{
14 ...
15 void foo();
16 int bar();
17 ...
18 }
```

```
19 class Callee_Proxy_1 extends Callee{
20  Callee_1 object;
21  ...
22 }
```

Abbildung 4.4: Gegenüberstellung: Proxies für ein Interface und eine Klasse.

Aufgrund der Tatsache, dass von einer Proxy Klasse Instanzen erstellt werden müssen, muss es sich bei der Proxy Klasse für ein Interface um eine Klasse und nicht um ein Interface handeln. Um mit der alten Version des Interfaces kompatibel zu sein, muss die Proxy Klasse die alte Interfaceversion implementieren und damit auch alle Methoden dieses Interfaces und aller Superinterfaces selbst definieren. Damit in ihr Instanzen von Klassen, die die neue Version des nachgeladenen Interfaces implementieren gespeichert werden können, besitzt die Proxy Klasse ein Feld vom Typ der neuen Interfaceversion. Ein Beispiel dafür ist in Abbildung 4.4 von Zeile 1 - 12 zu sehen, in dem es sich bei der

Klasse `ICallee_Pro_1` (Zeile 7 - 12) um den Proxy des Interfaces `ICallee` handelt. Da die Methoden, die die Proxy Klasse durch das Interface erhält, nie benutzt werden, kann ihr Methodenbauch leer bleiben beziehungsweise den Default Wert des Rückgabetypen zurückgeben.

Damit die Unterschiede zwischen den Proxy Klassen für Klassen und für Interfaces zu sehen sind, sind diese in Abbildung 4.4 auf der vorherigen Seite gegenübergestellt. Dabei handelt es sich um die Klasse `ICallee_Proxy_1`, die die Proxy Klasse zum Interface `ICallee` ist und um die Klasse `Callee_Proxy_1`, die die Proxy Klasse für `Callee` ist. Zur besseren Übersichtlichkeit ist die in Abschnitt 2.2.2 auf Seite 9 beschriebene obligatorische `newInstance(..)` Methode der Proxy Klassen, mit der neue Instanzen der Klasse erstellt und zurückgegeben werden, nicht angegeben.

4.4.2 Superklassen und ihre abgeleiteten Klassen als aufrufende Klassen

Wie im Abschnitt 2.4 auf Seite 15 beschrieben, besitzen Instanzen abgeleiteter Klassen alle nicht statischen Felder, die in ihren direkten und indirekten Superklassen definiert werden. Wird die Klasse `Callee` aus Abbildung 4.5 nachgeladen, so handelt es sich bei den Klassen `SuperCaller` und `SubCaller` um aufrufende Klassen, die Felder vom Typ der nachgeladenen Klassen deklarieren. Da die Klasse `Caller` von der Klasse `SuperCaller` erbt, können Instanzen der Klasse `Caller` auch auf die Felder vom Typ `Callee` zugreifen, die in der Klasse `SuperCaller` deklariert sind. Deshalb ist die Klasse `Caller` auch eine aufrufende Klasse.

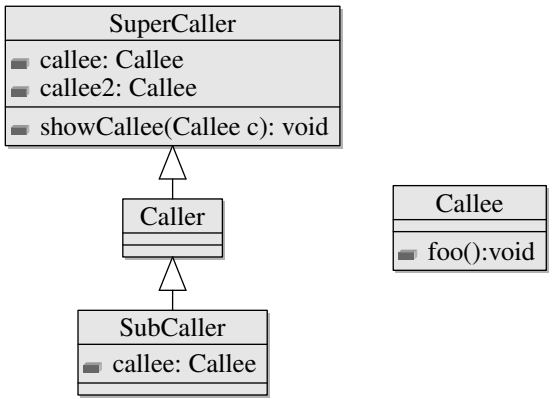

Abbildung 4.5: Superklasse und abgeleitete Klasse als Aufrufer.

Da die Klassen `SuperCaller` und `SubCaller` Felder vom Typ `Callee` besitzen, müssen für diese Felder die FieldContainer Klassen `SuperCaller_Cont_1` und `SubCaller_Cont_1` erstellt werden.

Wird die Klasse `SuperCaller` dahingehend verändert, dass sie die Klasse `Callee_1` referenziert, so kann dies wie in Abschnitt 2.2.2 auf Seite 8 gezeigt, geschehen. Um zur Laufzeit mit den Feldern vom Typ der nachgeladenen Klasse arbeiten zu können, wird das durch JavAdaptor hinzugefügte FieldContainer Feld der Klasse instanzweise mit einer Instanz vom Typ `SuperCaller_Cont_1` belegt.

Soll in Instanzen der Klasse `Caller` zur Laufzeit die Felder vom Typ `Callee_1` benutzt werden, so muss nicht das in ihr definierte FieldContainer Feld mit einer FieldContainer Instanz belegt werden, sondern das FieldContainer Feld, was in der Klasse `SuperCaller` definiert ist. Der Grund dafür liegt in der Tatsache, dass auch die Felder, die in die FieldContainer Klasse ausgelagert wurden, nicht in der Klasse selbst, sondern in ihrer Superklasse definiert sind.

Für Instanzen der Klasse `SubCaller` aus Abbildung 4.5 auf der vorherigen Seite werden nach dem Nachladen der Klasse `Callee` mehrere FieldContainer Instanzen benötigt. Dabei handelt es sich um eine Instanz der Klasse `SuperCaller_Cont_1` und um eine Instanz von `SubCaller_Cont_1`. Durch diese Instanzen ist es dann möglich, auf die aus den Klassen `SubCaller` und `SuperCaller` ausgelagerten Felder separat zuzugreifen.

4.4.3 Interfaces als aufrufende Klassen

Wie in Abschnitt 3.4 auf Seite 21 beschrieben, können Interfaces nur als aufrufende Klassen einer nachgeladenen Klasse existieren, wenn in dem Interface eine Methode deklariert wird, die als Parameter- oder als Rückgabetyp den Typ der nachgeladenen Klasse besitzt, oder ein Feld in dem Interface existiert, dessen Typ die nachgeladene Klasse ist.

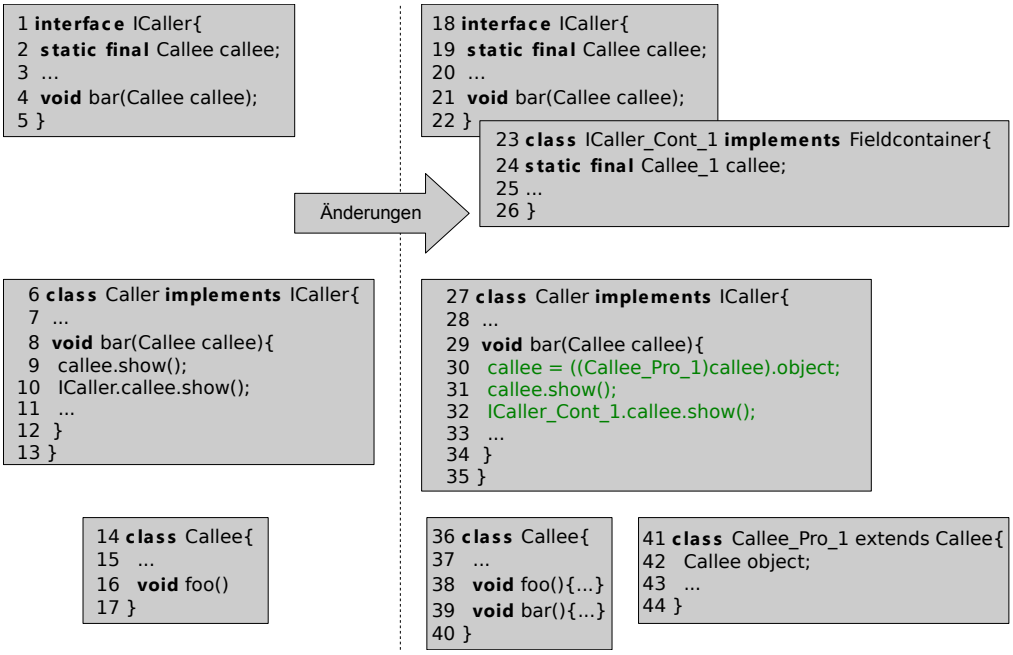

Abbildung 4.6: Interface als Aufrufer einer nachgeladenen Klasse.

Wird die Klasse `Callee` aus Abbildung 4.6 nachgeladen, so handelt es sich bei den Klassen `Caller` und `ICaller` um aufrufende Klassen der nachgeladenen Klasse, da sowohl das Interface `ICaller`, als auch die Klasse `Caller` die Methode `bar(Callee callee)` besitzen.

Da Interfaces Methoden nur deklarieren und nicht implementieren, wird kein `Proxy` für den Parameter vom Typ `Callee` in die `bar(Callee callee)` Methode des Interfaces (Zeile 21) eingebaut.

Aufgrund der Tatsache, das alle Felder, die in einem Interface deklariert sind, den Modifier `static` besitzen, sind die Werte dieser Felder klassenspezifisch. Das bedeutet, dass die Aufrufe dieser Felder direkt über die Klasse, in der die Felder deklariert sind geschehen. Deswegen muss bei Interfaces kein `Fieldcontainer` Feld hinzugefügt werden.

Für das Beispiel aus Abbildung 4.6 auf der vorherigen Seite bedeutet dies, das der Aufruf des im Interface `ICaller` deklarierten Feldes `callee` aus Zeile 2 nach dem Nachladen der Klasse `Callee` über die Klasse `ICaller_Cont_1` geschieht. Um dies zu erreichen, wird der existierende Aufruf `ICaller.callee` (Zeile 10) durch den Aufruf `ICaller_Cont_1.callee` (Zeile 32) ersetzt.

5. Implementierung

In diesem Kapitel werden die wichtigsten Implementierungsdetails beschrieben, durch die mittels JavAdaptor vererbungshierarchiebeeinflussende Programmänderungen während der Laufzeit durchgeführt werden können. Zum besseren Verständnis, in welchen Schritten das Nachladen einer Klasse abläuft, werden die dabei durchlaufenden Phasen Abschnitt 5.1 erläutert. In den nachfolgenden Abschnitten wird die Implementierung einzelner Teilbereiche des Nachladeprozesses aufgezeigt.

5.1 Programmablauf - Überblick

Bei Abbildung 5.1 auf der nächsten Seite handelt es sich um eine genauere Beschreibung des in Abbildung 2.1 auf Seite 5 gezeigten Nachladeprozesses. Wie schon in Abschnitt 2.2 auf Seite 4 beschrieben, kann der Entwickler, nachdem er die Applikation gestartet hat, Änderungen über die Eclipse-IDE in den Quellcodedateien vornehmen und diese Änderungen dann mittels JavAdaptor in die Applikation einspielen.

Sollen die Änderungen, die durch den Entwickler an Klassen durchgeführt wurden, in die Applikation übernommen werden, verbindet sich JavAdaptor mit der JVM. Nachdem dies geschehen ist, werden zu den Klassen, die durch den Entwickler verändert wurden, noch ihre direkten und indirekten Subklassen hinzugefügt. Wie schon beschrieben, muss beim Nachladen der einzelnen Klassen mit neuer Version darauf geachtet werden, dass Superklassen vor ihren Subklassen nachgeladen werden, um die Subklassen nicht mehrfach nachladen zu müssen. Die Klassen dürfen also nicht in willkürlicher Reihenfolge nachgeladen werden.

Wird eine Klasse unter neuer Version in die Applikation geladen, so müssen Veränderungen in der class Datei der Klasse vorgenommen werden. Dabei handelt es sich unter anderem um das Umbenennen der Klasse, das Aktualisieren aller Referenzen auf die jeweils aktuellste Version der referenzierten Klasse und das Hinzufügen des FieldContainer Feldes. Außerdem werden noch alle `final` Modifier aus der Klassendefinition entfernt. Das Gleiche gibt auch für alle `final` Modifier, die das Verändern von Parametern verhindern. Der Grund dafür ist in Abschnitt 2.2.2 auf Seite 12 beschrieben.

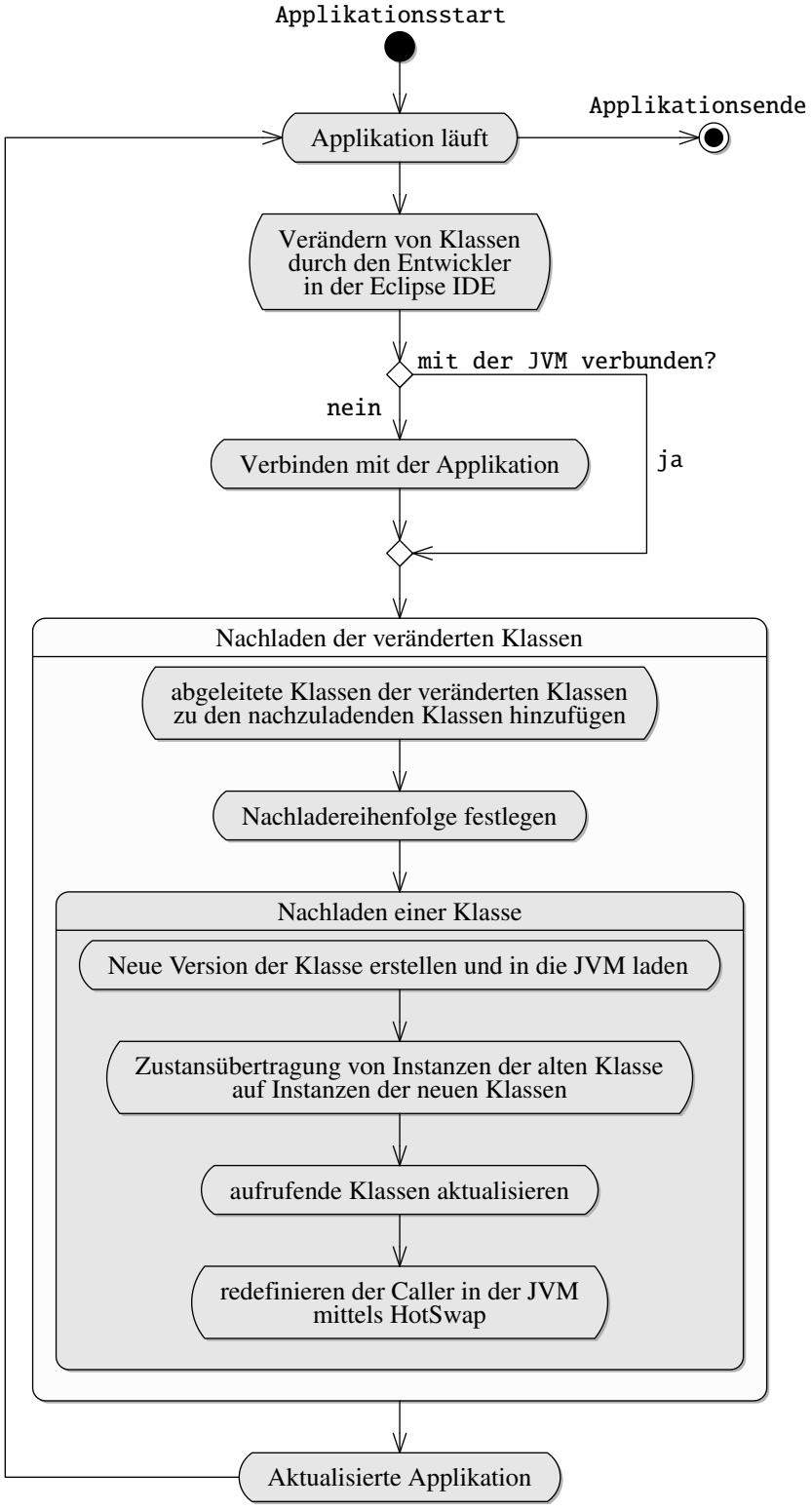

Abbildung 5.1: Programmablauf von JavAdapor.

Nachdem die Klasse mit neuem Referenztypen in die Applikation geladen wurde, müssen die Zustände existierender Instanzen paarweise auf die Instanzen der neuen Klassenversion übertragen werden. Dies muss für alle nicht statischen Felder geschehen, die die Instanz besitzt. Außerdem werden die Zustände statischer Felder auf die neue Klassenversion übertragen, wenn das Feld nicht die Modifier `final` und `static` besitzt. Worauf bei der Zustandsübertragung geachtet werden muss, wird in Abschnitt 5.4 auf Seite 37 genauer beschrieben.

Ist die nachzuladende Klasse in die JVM geladen worden, müssen ihre aufrufenden Klassen dahingehend verändert werden, dass sie die neue Version der nachgeladenen Klasse nutzen. Wird die nachzuladende Klasse global in einer Klasse verwendet, so werden die nötigen Veränderungen mittels `Expression Editor` eingebaut. Die dabei nötigen Änderungen werden anhand eines Expression Editors, der Feldzugriffe behandelt beschrieben (Abschnitt 5.5 auf Seite 39).

Damit die in den aufrufenden Klassen durchgeführten Änderungen in die JVM übernommen werden, werden die Klassen mittels HotSwap redefiniert. Danach ist der Aktualisierungsprozess abgeschlossen und die geänderten Programmteile können ausgeführt werden.

5.2 Subklassen zu den nachzuladenden Klassen hinzufügen

Gab es eine schemabeeinflussende Änderung in einer Klasse, so müssen alle direkten und indirekten Subklassen dieser Klasse mit neuem Namen in die Applikation geladen werden. Das Gleiche gilt auch für Klassen, die ein Interface implementieren, dessen Schema verändert wurde.

Damit dies gewährleistet werden kann, wird in der internen **JavAdaptor** Repräsentation einer Klasse, einem `ClassObject`, beim Applikationsstart gespeichert, welche Klassen von der Klasse abgeleitet sind. Handelt es sich bei der Klasse um eine Interface, werden zusätzlich noch alle Klassen gespeichert, die das Interface implementieren. Gibt es zur Laufzeit der Applikation vererbungshierarchiebeeinflussende Änderungen, werden diese Informationen aktualisiert.

Wurde eine Klasse durch den Entwickler verändert, so wird ihr repräsentierendes `ClassObject` in eine Liste eingefügt, in der die Repräsentationen aller durch den Entwickler veränderten Klassen abgespeichert sind. In Abbildung 5.2 auf der nächsten Seite ist ein Diagramm gezeigt, in dem die Schritte beschrieben sind, die durchgeführt werden müssen, um alle indirekt durch den Entwickler veränderten Klassen zu identifizieren.

Dazu wird in der Methode eine Ergebnisliste erstellt, die am Anfang noch leer ist. Anschließend werden nacheinander alle Elemente, die in der übergebenen Liste, in der am Anfang alle durch den Entwickler veränderten Klassen enthalten sind, aus dieser entfernt und bearbeitet. Nachdem das `ClassObject` einer Klasse aus der Startliste entfernt wurde, wird es in die Ergebnisliste eingefügt.

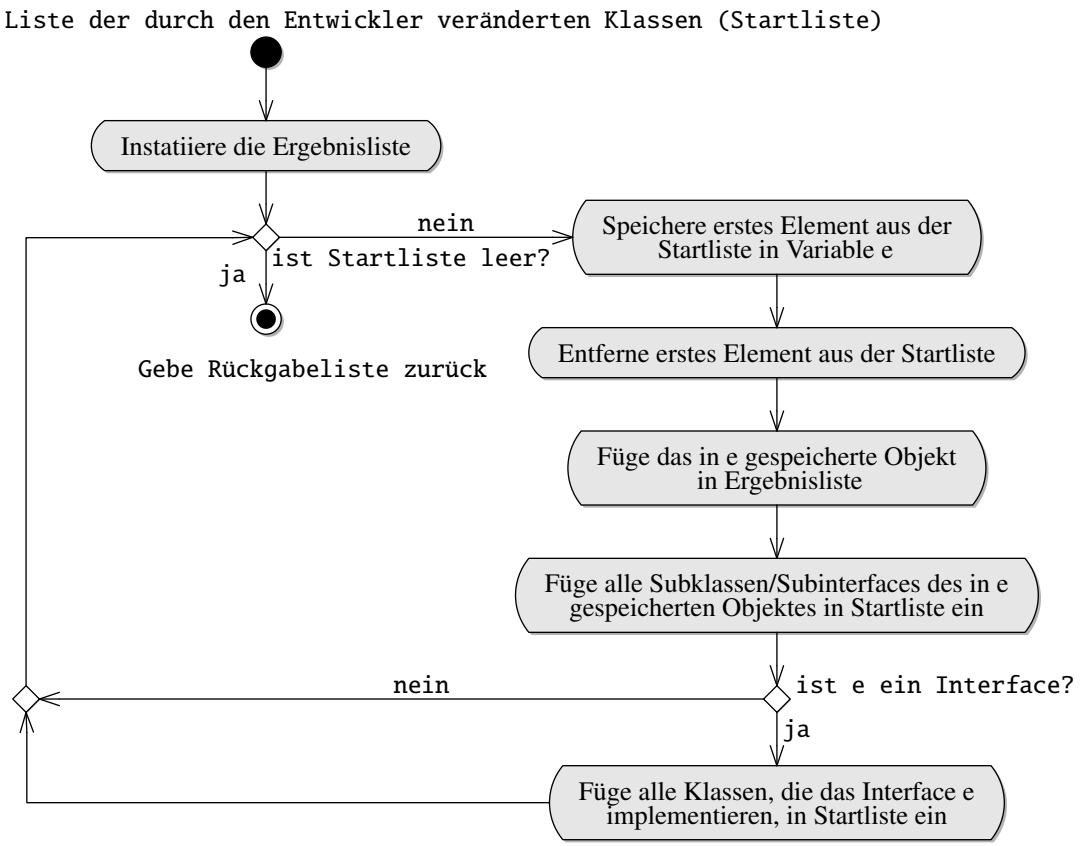

Abbildung 5.2: Abgeleitete Klassen zur Liste der nachzuladenden Klassen hinzufügen.

Danach werden die repräsentierenden Objekte der direkten Subklassen beziehungsweise der direkten Subinterfaces in die Startliste eingefügt. Repräsentiert das `ClassObject` ein Interface, so werden außerdem alle internen Repräsentationen der Klassen, die das Interface implementieren, in die übergebene Liste eingefügt. Dies geschieht jedoch nur, wenn die Objekte noch nicht in der Startliste oder der Ergebnisliste enthalten sind.

In einem späteren Durchlauf der Methode werden dann die in die Startliste eingefügten Elemente bearbeitet. Dadurch wird sichergestellt, dass alle direkten und indirekten Subklassen auch nachgeladen werden, wenn an einer ihrer Superklassen Änderungen vorgenommen wurden. Das Gleiche gilt auch für alle Klassen, die ein verändertes Interface implementieren.

Ausschnitte des dazugehörigen Quellcodes sind in Listing A.1 auf Seite 57 abgebildet.

5.3 Nachladen einer Klasse

Nachdem alle zu aktualisierenden Klassen identifiziert wurden, muss eine Reihenfolge festgelegt werden, in der sie in die JVM geladen werden. Beim Festlegen dieser Reihenfolge wird darauf geachtet, dass die Superklassen vor ihren abgeleiteten Klassen nachgeladen werden müssen.

Sind die Klassen sortiert worden, so können die geänderten Klassen geladen werden. Die dazu verwendete Vorgehensweise ist in dem in Abbildung 5.3 auf der nächsten Seite

abgebildeten Diagramm skizziert. Sollen die Änderungen, die an einer Klasse durchgeführt wurden, in die JVM übernommen werden, versucht JavAdaptor zunächst, die Klasse mittels HotSwap zu redefinieren. Bevor dies geschieht, werden alle standardmäßigen Änderungen an der Klasse vorgenommen. Wie in Abschnitt 2.2.2 auf Seite 12 beschrieben, handelt es sich dabei um das Hinzufügen des FieldContainer Feldes und um das Entfernen des `final` Modifier sowohl aus der Klassendefinition, als auch aus den Parameterndefinitionen der Methoden.

Hat der Entwickler Schemaänderungen an der Klasse durchgeführt, so schlägt der Versuch des Redefinierens der Klasse mittels HotSwap fehl und die Klasse muss unter neuem Namen in die Applikation geladen werden. Ist dem so, müssen alle Subklassen dieser Klasse unter neuem Namen in die JVM geladen werden.

Konnte eine Klasse nicht mittels HotSwap redefiniert werden, so muss für sie eine neue Version erstellt werden, die dann in die Applikation geladen wird. Nachdem dies geschehen ist, müssen die Zustände existierender Instanzen von Instanzen der neuen Klassenversion übernommen werden. Außerdem müssen alle aufrufenden Klassen der ausgetauschten Klasse dahingehend angepasst werden, dass sie die neue Version der Klasse verwenden. Dabei muss darauf geachtet werden, dass es zu keiner Schemaänderung in den aufrufenden Klassen kommt.

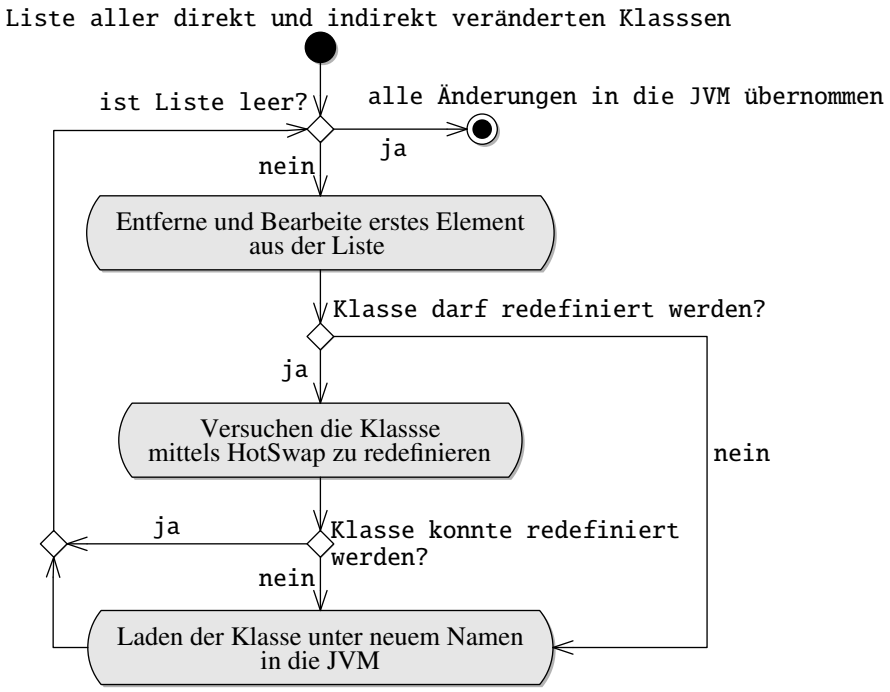

Abbildung 5.3: Redefinieren mittels HotSwap.

5.4 Zustandsübernahme

Bevor eine Zustandsübertragung für existierende Instanzen vollzogen werden kann, müssen die Klassen ermittelt werden, die nach wie vor Bestandteil der Hierarchie der Klasse

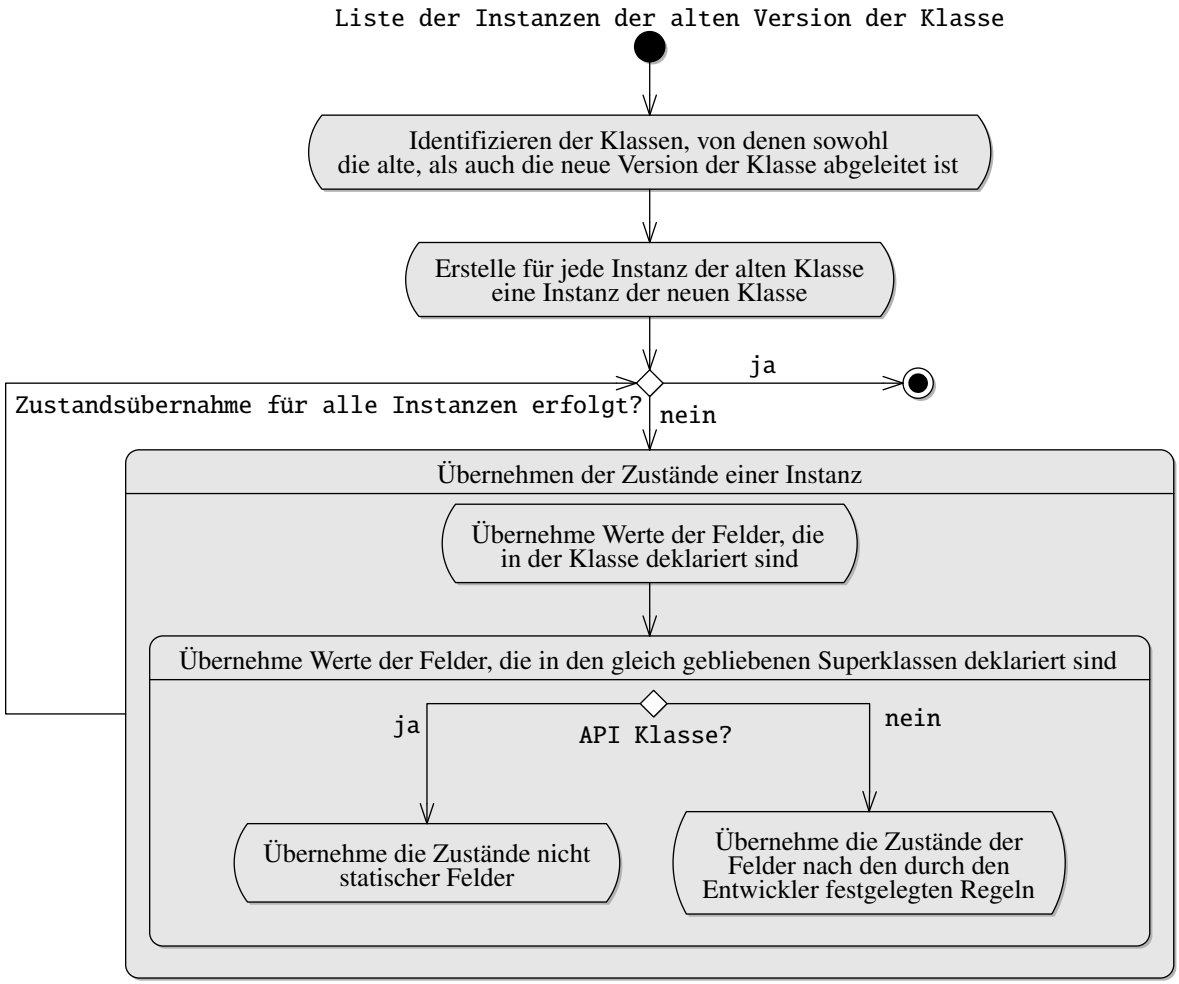

Abbildung 5.4: Zustandsübernahme

sind. Dabei handelt es sich um alle Klassen, von denen sowohl die alte als auch die neue Version der Klasse abgeleitet sind.

Nachdem diese Klassen identifiziert wurden, wird für jede Instanz der alten Klassenversion eine Instanz der neuen Klassenversion erstellt. Ist dies geschehen, kann die Zustandsübernahme für diese Instanzen vollzogen werden. Dabei werden zuerst die Zustände der nicht statischen Felder, die in der nachgeladenen Klasse selbst deklariert werden, auf eine Instanz der neuen Klasse übertragen. Gibt es Änderungen in diesen Feldern, so muss der Entwickler Regeln festlegen, wie diese Änderungen bei der Zustandsübertragung berücksichtigt werden sollen. Nachdem dies geschehen ist, werden die Zustände der Felder, die in den gleich gebliebenen Superklassen deklariert sind, übertragen. Dabei muss unterschieden werden, ob es sich bei der Superklasse um eine API Klasse oder um eine Klasse, die in der Applikation definiert ist, handelt. Der Grund für die Unterscheidung liegt in der Tatsache, dass eine API Klasse nicht verändert werden kann und somit die Zustände eins zu eins übertragen werden müssen. Im Gegensatz dazu, können sich Klassen, die in der Applikation definiert werden, geändert haben. Gab es Änderungen in den Feldern, die in einer gleich gebliebenen Superklasse

deklariert werden, so hat der Entwickler zu dem Zeitpunkt, als die neue Version der Klasse in die JVM geladen wurde, festgelegt, welche Felder der alten und der neuen Version der Klasse miteinander in Beziehung stehen und wie diese Zustandsübertragung erfolgen soll. Da diese Beziehungen auch für Instanzen der Subklasse gelten, müssen sie beim Übertragen der Zustände ebenfalls angewendet werden.

5.5 Anpassung der Aufrufenden Klassen mittels Expression Editor

Nachdem eine Klasse mit neuer Version in die Applikation geladen wurde, müssen ihre aufrufenden Klassen dahingehend verändert werden, dass sie die neue Version der Klasse benutzen. Müssen Feldzugriffe über eine FieldContainer Klasse erfolgen oder müssen Proxy Instanzen in einer Methode verwendet werden, so wird diese Anpassung unter Zuhilfenahme der Klasse `ExprEditor` der Klassenbibliothek **Javassist** vollzogen.

5.5.1 Zugriffe auf Felder vom Typ der Nachgeladenen Klasse

Anders als in Abschnitt 2.2.2 auf Seite 8, in dem durch den FieldContainer einer Klasse nur auf Felder, die in der Klasse definiert wurden, verwiesen wird, kann es durch Vererbungsbeziehungen in den aufrufenden Klassen einer nachgeladenen Klasse auch vorkommen, dass, wie schon in Abschnitt 2.4 auf Seite 15 beschrieben, auf Felder zugegriffen wird, die in einer der Superklassen definiert wurden.

In Abbildung 5.5 auf der nächsten Seite (Zeile 20 -37) ist ein Ausschnitt der Methode des Expression Editors gezeigt, der die Veränderung der Feldzugriffe durchführt. Zuvor wird ein Überblick über die in der Implementierung verwendeten Variablen und durch `Javassist` definierten Konstrukte gegeben:

Verwendete Variablen:

- declaringClass = Der Name der Klasse, in der das originale Feld deklariert wurde.
- cast = Der Name der Klassen, in dem die neue Version des Feldes deklariert ist.
- feldName = Der Name des Feldes, auf dem der Zugriff passiert.
- feldCont = Der Name des `FieldContainer` Feldes.

Konstrukte in `Javassist`:

- $0 = Das Objekt, über das der Aufruf auf das Feld erfolgt. Dabei muss es sich nicht unbedingt um die this Referenz handeln.
- $1 = Der Wert, der durch den schreibenden Zugriff in dem Feld gespeichert werden soll.
- $_ = Der Wert des zu lesenden Feldes.

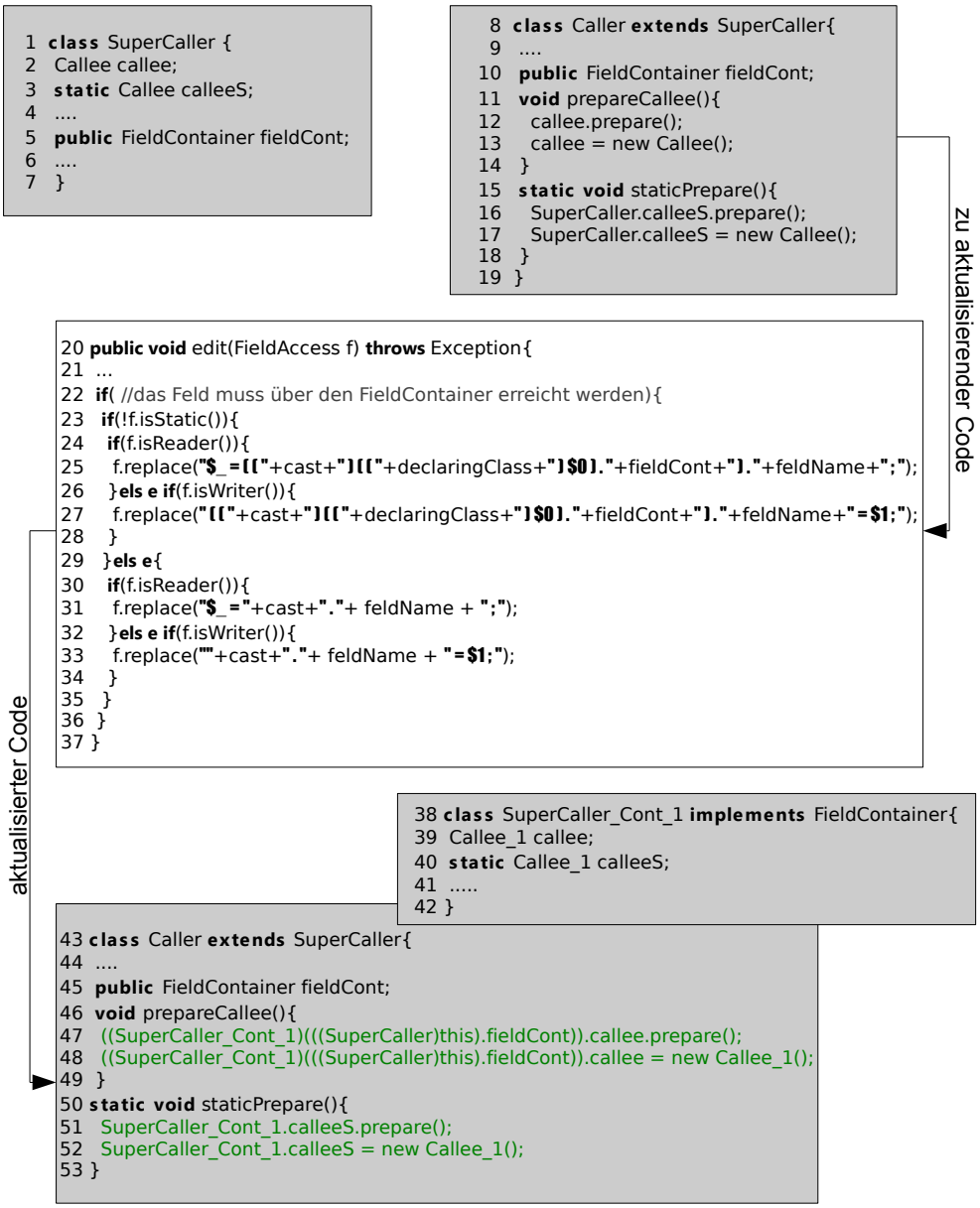

Abbildung 5.5: Änderung der Feldzugriffe mittels Expression Editor.

Zum Verständnis der Bytecode-Änderungen, die durch den in Zeile 20 - 37 gezeigten Expression Editor vorgenommen werden, wird in Abbildung 5.5 in den Zeilen 1 bis 19 und 38 bis 53 ein Beispiel gegeben. In ihm werden einzelne Feldzugriffe in der Klasse `Caller` verändert, um auf Felder vom Typ der nachgeladenen Klasse (`Callee`) auch nach dem Nachladen der Klasse und dem Auslagern der Felder in einen FieldContainer zugreifen zu können.

Bei den Veränderungen muss unterschieden werden, ob es sich um einen Aufruf auf ein statisches oder ein nicht statisches Feld handelt, sowie ob es ein lesender oder schreibender Zugriff ist.

Zugriff auf nicht statische Felder. Handelt es sich um Zugriffe auf nicht statische Felder wie in Zeile 12 und 13 der Abbildung 5.5 auf der vorherigen Seite, so wird bei einem lesenden Zugriff die Anweisung aus Zeile 25 ausgeführt und bei einem schreibenden Zugriff die Anweisung aus Zeile 27.

Der Zugriff auf nicht statische Felder muss dabei über das **FieldContainer** Feld der deklarierenden Klasse erfolgen. Um bei Überlagerung von Feldern auf dem richtigen Feld, dessen Aufruf angepasst werden soll, zu arbeiten, wird das Objekt, welches den Feldaufruf ausführen soll, auf die Klasse gecastet, in der das Feld deklariert wird. Dies geschieht in Abbildung 5.5 auf der vorherigen Seite in Zeile 25 und 27 durch einen cast auf die Klasse, deren Name in der Variable `declaringClass` gespeichert ist. In Abbildung 5.5 auf der vorherigen Seite handelt es sich dabei um die Klasse `SuperCaller`. Danach kann auf den FieldContainer zugegriffen werden, in dem die neue Version des aufgerufenen Feldes deklariert ist. Um dann in dem FieldContainer auf das Feld zuzugreifen, wird die Fieldcontainer Instanz auf die Fieldcontainer Klasse, in der das neue Feld deklariert ist, gecastet. In Abbildung 5.5 auf der vorherigen Seite handelt es sich um die Klasse `SuperCaller_Cont_1`. Nachdem dies gemacht wurde, kann auf dem gecasteten Objekt der Feldzugriff durchgeführt werden (Zeile 47,48).

Zugriff auf statische Felder. Wird ein statisches Feld aufgerufen, dass in eine FieldContainer Klasse ausgelagert wurde, so reicht es aus, den Zugriff über die `FieldContainer` Klasse, in der die neue Version des Feldes deklariert wird, umzuleiten. Für das in Abbildung 5.5 auf der vorherigen Seite gezeigte Beispiel bedeutet das, dass die Zugriffe auf das Feld `calleeS` dahingehend verändert werden, dass ihr Aufruf über die Klasse `SuperCaller_Cont_1` erfolgt. In Zeile 51 und 52 ist zu sehen, wie diese Veränderung für einen lesenden beziehungsweise für einen schreibenden Zugriff aussieht.

6. Evaluierung

Im vorhergehenden Kapitel wurden Teile der Implementierung gezeigt, durch die es möglich ist, mittels JavAdaptor an Applikationen vererbungshierarchiebeeinflussende Änderungen während der Laufzeit durchzuführen. Zur Evaluierung dieser Implementierung werden zwei Testfälle herangezogen, in denen eine Veränderung der Vererbungshierarchie einer Klasse und eines Interfaces zur Laufzeit der Applikation durchgeführt wird. Außerdem wird in zwei anderen Fällen getestet, ob das Nachladen von Klassen korrekt durchgeführt werden kann, wenn eine Vererbungshierarchie in den aufrufenden Klassen beziehungsweise den aufrufenden Interfaces existiert. Des Weiteren ist es noch notwendig zu testen, ob Proxies und FieldContainer für Interfaces korrekt erstellt und dann anschließend verwendet werden.

6.1 Aufbau der Testfälle

Damit diese Testszenarien durchgeführt werden können, muss eine Testumgebung entwickelt werden, an die bestimmte Anforderungen gestellt sind. Bei ihr muss es sich um eine lang laufende Applikation handeln, in der außerdem noch die Möglichkeit besteht, bestimmte Codesegmente wiederholt auszuführen. Da diese Anforderungen von grafischen Applikationen erfüllt werden, wurde eine grafische Testumgebung entwickelt, deren Implementierung in Kapitel A auf Seite 57 zu sehen ist.

Während der Laufzeit der Applikation wird von der Benutzeroberfläche eine Instanz der Klasse `CallerCaller` gehalten. Die Implementierung dieser Klasse ist in Abbildung 6.1 auf der nächsten Seite (Zeile 1 - 8) gezeigt. Sie besitzt eine Methode `bar()` (Zeile 4-7) und ein Feld vom Typ `Caller` (Zeile 2). Bei der Klasse `Caller` handelt es sich in allen Testfällen um eine aufrufende Klasse der nachzuladenden Klasse oder des nachzuladenden Interfaces. Die Methode `do()` der Klasse `Caller`, die von der Methode `bar()` aufgerufen wird, ruft die Funktionalität auf, die zur Laufzeit verändert werden soll. Durch diesen Aufbau kann überprüft werden, ob die veränderte Funktionalität korrekt in die JVM übernommen wurde und auch korrekt verwendet wird.

```
1 class CallerCaller{
2   Caller caller;
3   ...
4   void bar(){
5     caller.do();
6     ...
7   }
8 }
```

Abbildung 6.1: Aufbau der Testfälle - CallerCaller.

Bei der Auswertung der Testfälle wird jeweils ein statischer und ein dynamischer Test durchgeführt. Im statischen Test wird untersucht, ob der Bytecode der aufrufenden Klassen an den notwendigen Stellen korrekt verändert wurde. Bei dem dynamischen Test wird überprüft, ob eine Zustandsübertragung von alten Instanzen auf die neuen Instanzen der nachgeladenen Klasse korrekt vollzogen wurde. Damit dieser Test durchgeführt werden kann, besitzt die Klasse `Callee`, bei der es sich jeweils um eine der nachzuladenden Klassen handelt, in jedem Testfall ein `instIdent` Feld vom Typ `int`, dem beim Instantiieren ein zufällig generierter Wert zugewiesen wird. Der dynamische Test besteht darin, ob eine Gleichheit der Werte vor und nach dem Nachladen der Klasse in der existierenden Instanz vorliegt.

Zur besseren Übersichtlichkeit wurde in den nachfolgenden Abbildungen der Szenarien darauf verzichtet, das obligatorische FieldContainer Feld der Klassen mit anzugeben, wenn es bei dem beschriebenen Szenario nicht benötigt wird. Das gleiche gilt auch für die Konstruktoren der Klassen.

6.2 Vererbungshierarchieänderung an einer Klasse

In diesem Testszenario wird überprüft, ob das Verändern der Vererbungshierarchie einer Klasse durch den Austausch ihrer Superklasse zur Laufzeit ordnungsgemäß durchgeführt wurde. Außerdem wird untersucht, ob die geerbten Methoden der neuen Superklasse korrekt verwendet werden.

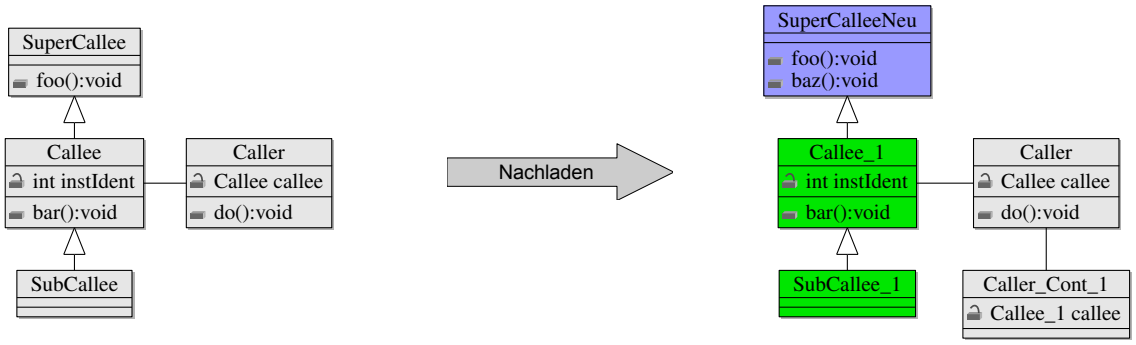

Abbildung 6.2: Vererbungshierarchieänderung einer Klasse.

Testszenario. Bei der Klasse, deren Vererbungshierarchie verändert werden soll, handelt es sich um die Klasse `Callee` aus Abbildung 6.2 auf der vorherigen Seite. Vor dieser Veränderung erbt sie von der Klasse `SuperCallee`. Diese Klasse wird dann zur Laufzeit der Applikation durch die Klasse `SuperCalleeNeu` ersetzt. Das bedeutet, dass die Klasse `Callee` nicht mehr von der Klasse `SuperCallee`, sondern von der Klasse `SuperCalleeNeu` erben muss. Wird diese Änderung an der Klasse `Callee` vorgenommen, so muss sie, aufgrund der daraus resultierenden Schemaänderung, unter neuem Namen nachgeladen werden. Außerdem muss die Klasse `SubCallee` nachgeladen werden, da sie von `Callee` abgeleitet wurde, die aktuelle Version der `Callee` Klasse jedoch die Klasse `Callee_1` ist.

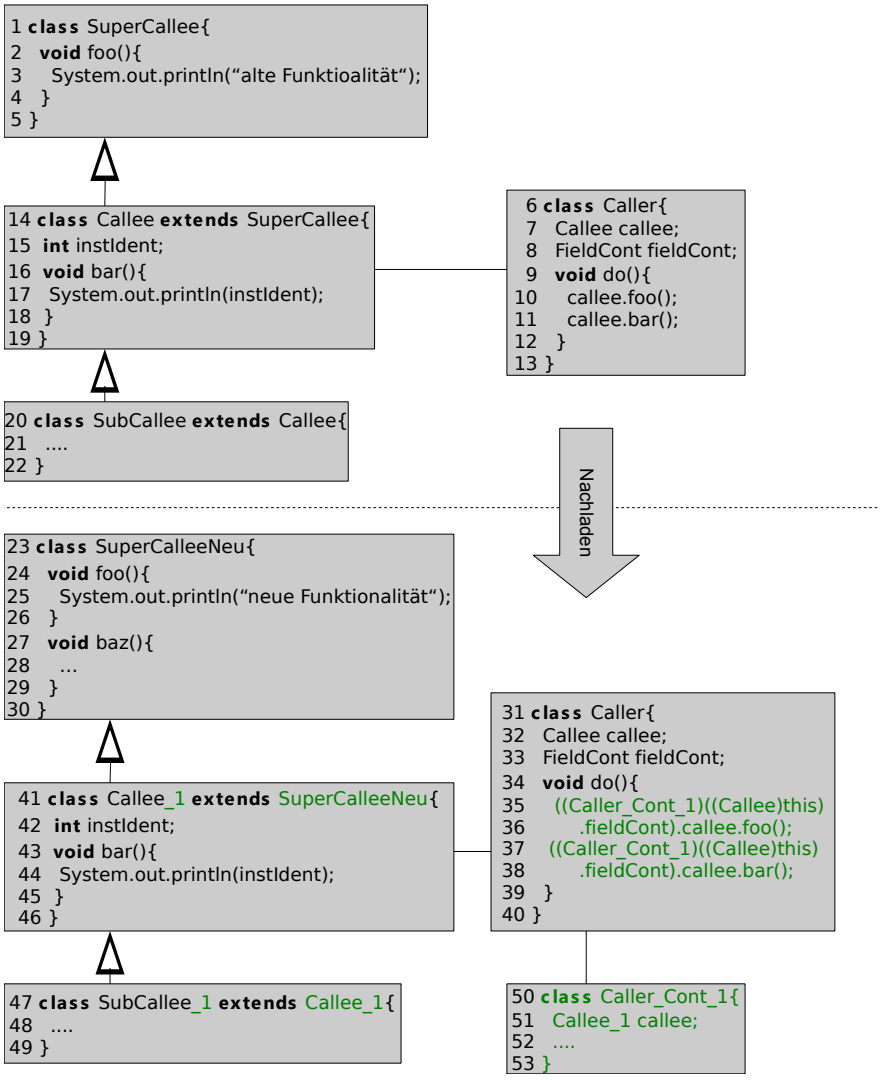

Abbildung 6.3: Quelltextänderungen - Vererbungshierarchieänderung einer Klasse.

Als aufrufende Klasse der Klasse `Callee` fungiert die Klasse `Caller` (Abbildung 6.2 auf der vorherigen Seite). Da sie ein Feld vom Typ `Callee` besitzt, muss für sie eine **FieldContainer** Klasse erstellt werden. In diese Klasse wird dann das `callee` Feld der

Klasse `Caller` ausgelagert. Bei der neu erstellten Klasse handelt es sich um die Klasse `Caller_Cont_1`.

Die dazu nötigen Veränderungen, die in den Klassen durchgeführt werden, sind in Abbildung 6.3 auf der vorherigen Seite gezeigt. In der Abbildung ist der Quelltext vor dem Nachladen der Klasse `Callee` zu sehen. Ihm ist der durch JavAdaptor veränderte Bytecode gegenübergestellt. Zur besseren Lesbarkeit und Vergleichbarkeit ist der Bytecode in äquivalenten Quellcode überführt worden.

Auswertung. Der veränderte Bytecode wurde dahingehen analysiert, ob die notwendigen Änderungen korrekt durchgeführt wurden. Da der Bytecode korrekt verändert wurde und auch bei einem Aufruf der Methode `do()` (Zeile 34 - 39) der Klasse `Caller`, die Funktionalität der `foo()` Methode der neuen Superklasse der nachgeladenen Klasse ausgeführt wird, sind alle nötigen Änderungen korrekt durchgeführt worden.

Wird der dynamische Test durch einen Aufruf der Methode `bar()` der nachgeladenen Klasse ausgeführt, wird von der `Callee_1` Instanz, die vom `Callee_1` Feld des Field-Containers (Zeile 51) gehalten wird, der gleiche Wert zurückgegeben, der auch vor dem Nachladen der Klasse von der existierenden `Callee` Instanz zurückgegeben wurde. Damit wurde auch die Zustandsübernahme ordnungsgemäß durchgeführt.

6.3 Vererbungshierarchieänderung an einem Interface

In diesem Testfall wird die Vererbungshierarchie eines Interfaces zur Laufzeit der Applikation insoweit verändert, dass ein zweites Interface hinzugefügt wird, von dem das betrachtete Interface erbt. Des Weiteren wird noch überprüft, ob Proxy Klassen für Interfaces korrekt erstellt werden.

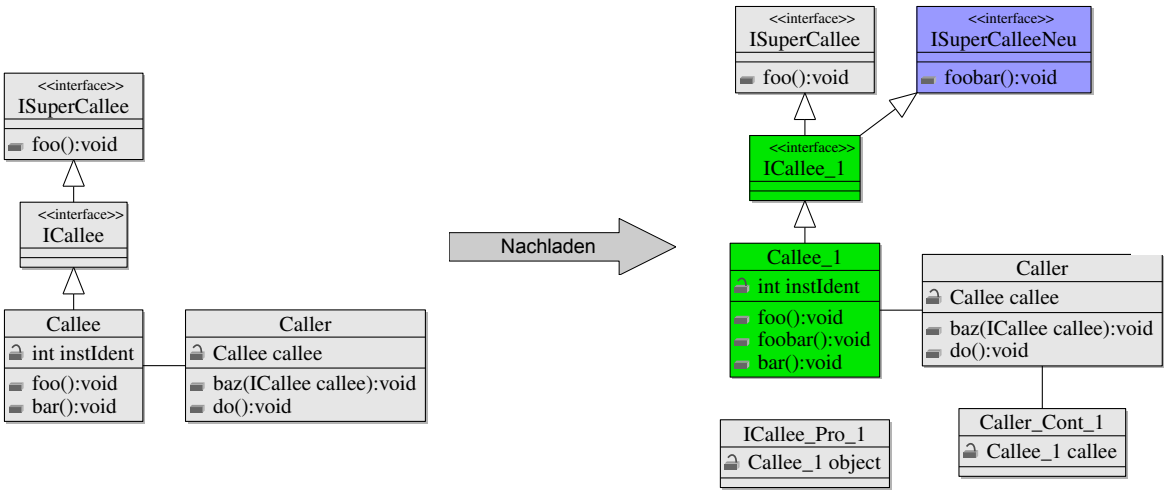

Abbildung 6.4: Vererbungshierarchieänderung eines Interfaces.

Testszenario. Bei dem Interface, dessen Vererbungshierarchie verändert werden soll, handelt es sich um das Interface `ICallee` aus Abbildung 6.4, das vor der Veränderung der Vererbungshierarchie vom Interface `ISuperCallee` abgeleitet wurde. Außerdem wird das Interface von der Klasse `Callee` implementiert. Als aufrufende Klasse

Abbildung 6.5: Quelltextänderungen - Vererbungshierarchieänderung eines Interfaces.

des Interface `ICallee` dient die Klasse `Caller`. Soll das Interface `ICallee` während der Laufzeit von zwei Interfaces erben, muss es unter neuem Namen nachgeladen werden. Damit die Klasse `Callee` auch von der neuen Version ihres Superinterfaces erbt, ist es nötig, auch diese unter neuem Namen nachzuladen.

Damit das Interfaces `ICallee_1` und die Klasse `Callee_1` in der Klasse `Caller` benutzt werden können, ohne eine Schemaänderung an der `Caller` Klasse durchführen zu müssen, wird eine FieldContainer Klasse für die Klasse `Caller` erstellt. Außerdem muss eine Proxy Klasse für das Interface `ICallee` erstellt werden. In Abbildung 6.5 sind die Veränderungen gezeigt, die mittels JavAdaptor gemacht wurden, um die beiden Klassen

unter neuem Namen in die Applikation zu laden und dann auch während der Laufzeit zu verwenden. Zum Vergleich ist der unveränderte originale Quellcode dem Bytecode, der durch den Entwickler und durch JavAdaptor verändert wurde, gegenübergestellt. Dieser Bytecode wurde vorher in äquivalenten Quellcode überführt.

Auswertung. Zunächst wurde der veränderte Bytecode darauf geprüft, ob alle erforderlichen Änderungen an ihm durchgeführt worden sind. Dabei wurden alle gewünschten Anpassungen lokalisiert. Da dem so ist, wird bei einem Aufruf der Methode `do()` (Zeile 37 - 41), nachdem das Interface `ICallee` und die Klasse `Callee` nachgeladen wurden und die Klasse `Caller` redefiniert wurde, die hinzugefügte Methode `foobar()` aufgerufen.

Wird die Methode `bar()` nach dem Nachladen der Klassen aufgerufen, so gibt sie den gleichen Wert wie vor dem Nachladen der Klassen `ICallee` und `Callee` zurück. Dadurch wurde gezeigt, dass auch die Zustandsübernahme korrekt vollzogen wurde.

6.4 Existierende Vererbungshierarchie in den aufrufenden Klassen

Nachdem Veränderungen der Vererbungshierarchie einer Klasse und eines Interfaces betrachtet wurden, folgt in diesem Abschnitt der Anwendungsfall, dass eine Klasse nachgeladen wird, in deren aufrufenden Klassen eine Vererbung existiert.

Testszenario. Ein Beispiel dafür ist in Abbildung 6.6 gegeben. Bei den aufrufenden Klassen handelt es sich um die Klassen `SuperCaller` und `Caller`, wobei die Klasse `Caller` von der Klasse `SuperCaller` abgeleitet ist. Wird zu der Klasse `Callee` eine neue Methode hinzugefügt, so muss sie aufgrund der daraus resultierenden Schemaänderung unter neuem Namen nachgeladen werden. Da die Klasse `SuperCaller` durch das Feld `callee` eine globale Referenz der Klasse `Callee` besitzt, muss eine **FieldContainer** Klasse erstellt werden. Dabei handelt es sich in Abbildung 6.6 um die Klasse `SuperCaller_Cont_1`.

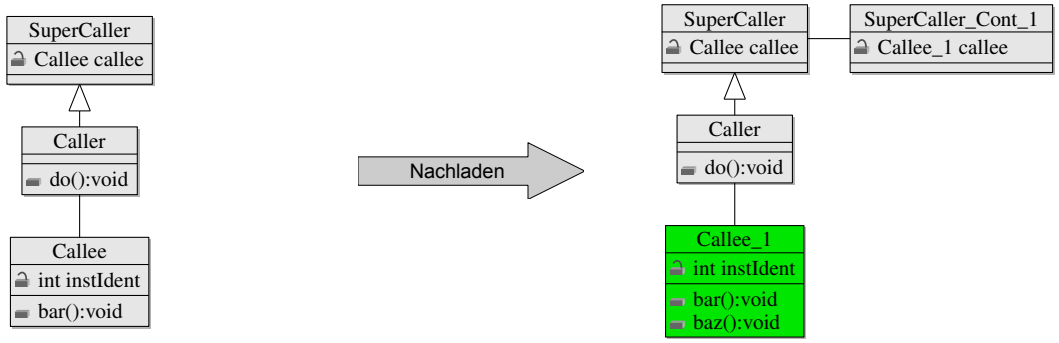

Abbildung 6.6: Vererbungshierarchie in den aufrufenden Klassen.

In Abbildung 6.7 auf der nächsten Seite ist der dazu gehörige Quellcode gezeigt. In der oberen Hälfte der Abbildung ist der Quelltext vor dem Nachladen der Klasse `Callee`

zu sehen. Ihm ist der Bytecode gegenübergestellt, der nach dem Nachladen der Klasse verwendet wird. Zur besseren Lesbarkeit und um den Code besser vergleichen zu können, wurde der veränderte Bytecode in äquivalenten Quellcode überführt.

Auswertung. Bei einer Überprüfung, ob die durch JavAdaptor an der Applikation durchgeführt Änderungen korrekt durchgeführt wurden, sind sowohl ein statischer als auch ein dynamischer Test vollzogen worden. Wird der veränderte Bytecode mit den angestrebten Änderungen verglichen, so wurden alle nötigen Änderungen an dem Bytecode vorgenommen. Aus diesem Grund wird bei einem Aufruf der Methode `do()` (Zeile 19 - 21) der Klasse `Caller` mit einer Instanz der Klasse `Callee_1` gearbeitet. Der dynamische Test wird durch einen Aufruf der Methode `bar()` (Zeile 25 - 27) durchgeführt. Da von der Methode vor und nach dem Nachladen der Klasse `Callee` der gleiche Wert zurückgegeben wird, wurde die Zustandsübernahme korrekt durchgeführt.

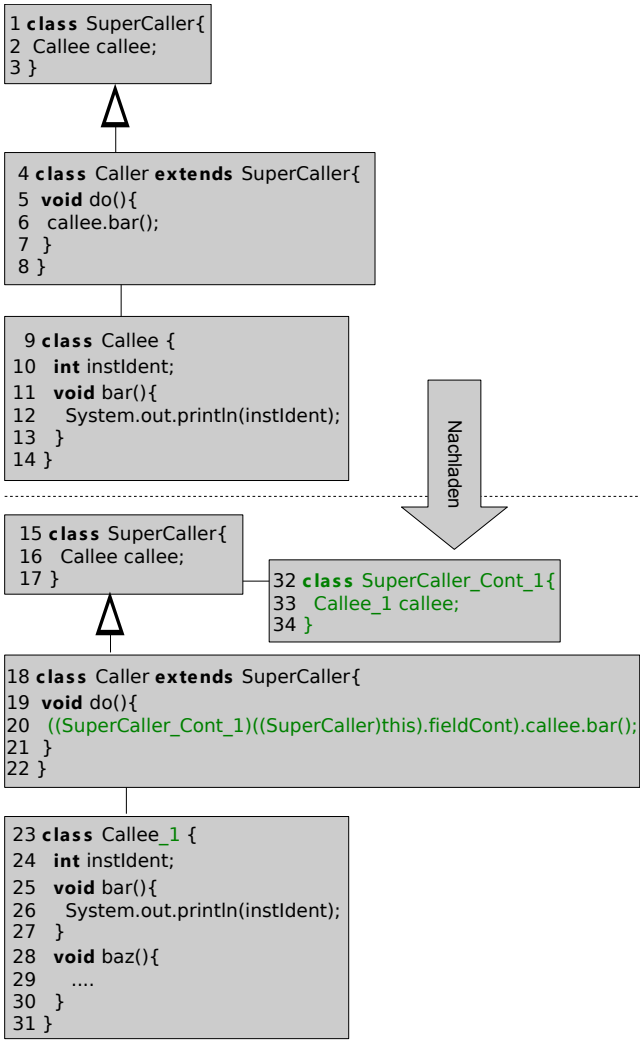

Abbildung 6.7: Quelltextänderungen - Vererbungshierarchie in den aufrufenden Klassen.

6.5 Existierende Vererbungshierarchie im aufrufenden Interface

In diesem Testszenario wird der Fall betrachtet, dass eine nachzuladende Klassen sowohl von Klassen, als auch von Interfaces genutzt wird. Damit eine Vererbungshierarchie in den aufrufenden Klassen zu existiert, implementiert die aufrufende Klasse das aufrufende Interface.

Testszenario. Ein Beispiel dafür ist in Abbildung 6.8 zu sehen. Bei der nachgeladenen Klasse handelt es sich um die Klasse `Callee`, zu der eine Methode hinzugefügt wurde. Die aufrufenden Klassen sind dabei das Interface `ISuperCaller` und die Klasse `Caller`. Da in dem Interface `ISuperCaller` ein Feld vom Typ der nachgeladenen Klasse deklariert wurde, muss eine FieldContainer Klasse erstellt werden. Außerdem muss für die `Callee` Klasse eine Proxy Klasse erstellt werden. Durch sie ist es dann möglich, an die `foo(Callee callee)` Methode der `Caller` Klasse Instanzen der neuen Klassenversion zu übergeben.

Die Änderungen, die durch JavAdaptor am Bytecode durchgeführt werden, sind als Quellcode dem originalen Quellcode in Abbildung 6.9 auf der nächsten Seite gegenübergestellt. Zur Lesbarkeit des veränderten Codes wurde er in äquivalenten Quellcode überführt.

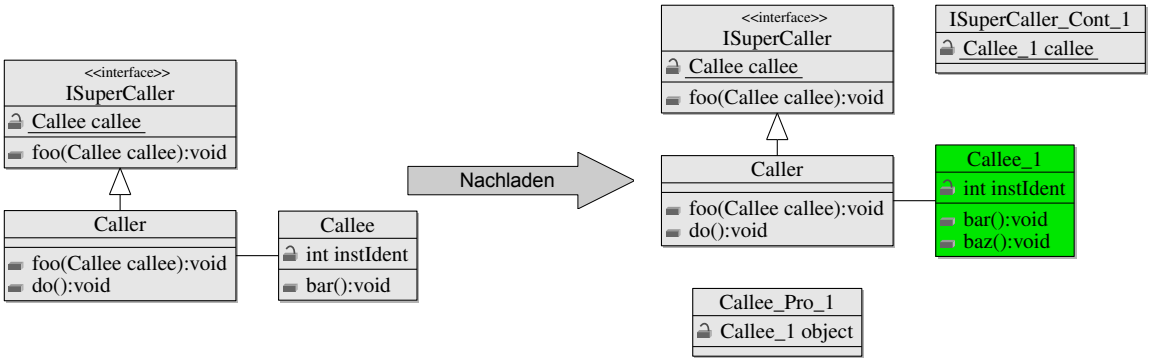

Abbildung 6.8: Vererbungshierarchie im aufrufenden Interfaces.

Auswertung. Die gewünschten Änderungen wurden im Bytecode der Klassen korrekt durchgeführt. Da bei einem Aufruf der `bar()` Methode des im FieldContainer gespeicherten Objektes der gleiche Wert wie vor dem Nachladen der Klasse `Callee` zurückgegeben wird, wurde eine Zustandsübernahme ordnungsgemäß vollzogen. Damit wird gezeigt, dass eine Zustandsübernahme auch bei Feldern, die in Interfaces deklariert wurden, ordnungsgemäß durchgeführt wird.

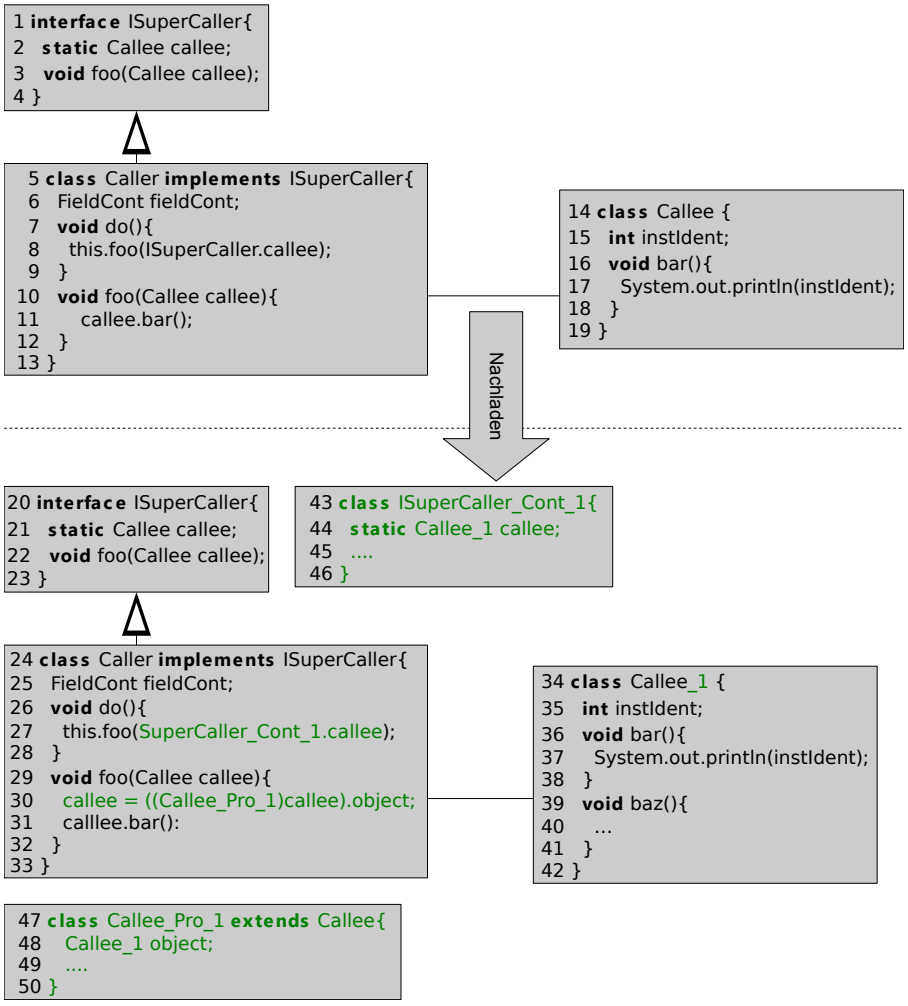

Abbildung 6.9: Quelltextänderungen - Vererbungshierarchie im aufrufenden Interfaces.

7. Zusammenfassung und Ausblick

Im Zuge dieser Arbeit wurde das Plug-In JavAdaptor erweitert. Es befindet sich zum gegenwärtigen Zeitpunkt noch in der Entwicklungsphase. Durch JavAdaptor ist es möglich, umfassende Funktionsänderungen an Java Applikationen während der Laufzeit durchzuführen. Dabei handelt es sich nicht nur um Änderungen, die durch HotSwap unterstützt werden, sondern auch um das Hinzufügen oder Entfernen von Methoden und Feldern einer Klasse. Es ist damit mittels JavAdaptor möglich, an Klassen schemabeeinflussende Änderungen während der Laufzeit durchzuführen. Wird das Schema einer Klasse verändert, so wird sie unter neuem Namen in die Applikation geladen. In diesem Zusammenhang werden ihre aufrufenden Klassen dahingehend verändert, dass sie die neue Version der Klasse verwenden, ohne dabei eine Schemaänderung an der Klasse durchzuführen. Dies wird unter Zuhilfenahme des Proxy und des FieldContainer Konzeptes, welche Teile von JavAdaptor sind, erreicht. Das Nachladen einer Klasse und das Verändern in den aufrufenden Klassen funktioniert jedoch nur, wenn die nachgeladene Klasse nicht von anderen Klassen abgeleitet wird und auch keine Vererbung in den aufrufenden Klassen untereinander existiert.

Gegenstand dieser Arbeit ist die Erweiterung von JavAdaptor zur Unterstützung von vererbungshierarchiebeeinflussenden Änderungen zur Laufzeit. Da es sich bei Java um eine objektorientierte Sprache handelt, ist Vererbung eines der wichtigsten Sprachkonzepte dieser Programmiersprache. Somit ist eine Unterstützung von Änderungen, die an diesem Konzept zur Laufzeit durchgeführt werden, unabdingbar.

Die dazu entwickelte Erweiterung unterstützt alle Veränderungen der Vererbungshierarchie, die mittels Änderungen von Superklassen oder zu implementierenden Interfaces möglich sind. Dadurch werden alle möglichen Vererbungshierarchieänderungen, die an Klassen und Interfaces durchgeführt werden können, unterstützt. Wird die Vererbungshierarchie eines Interfaces oder einer Klasse verändert, so kommt es automatisch zu einer Änderung des Klassenschemas. Da Klassen mit verändertem Schema von JavAdaptor unter neuem Namen in die Applikation geladen werden, wird dies auch mit Klassen gemacht, deren Vererbungshierachie verändert wurde.

Durch dieses Umbenennen von Klassen taucht im Zusammenhang mit Vererbung ein generelles Problem auf. Muss eine Klasse aufgrund einer Schemaänderung unter neuem Namen nachgeladen werden, so muss dies auch mit ihren abgeleiteten Klassen geschehen, auch wenn in ihnen keine Änderungen vollzogen wurden. Handelt es sich bei der veränderten Klasse um ein Interface, so müssen zusätzlich zu den existierenden Subinterfaces noch alle Klassen nachgeladen werden, die eines der nachzuladenden Interfaces implementieren.
Ein Grund dafür liegt in der Tatsache, dass die Änderungen, die an Klassen durchgeführt wurden, auch für ihre abgeleiteten Klassen durchgeführt werden müssen. Ein weiterer Grund ist der in Java existierende Polymorphismus. So müssen die aktuellsten Versionen von abgeleiteten Klassen immer von der aktuellsten Version ihrer Superklasse erben, um mit ihr typkompatibel zu sein. Deshalb darf beim Nachladen einer Klasse nicht darauf verzichtet werden, auch ihre Subklassen nachzuladen. Dies kann beim Nachladen von Klassen, die von sehr vielen Klassen abgeleitet werden, zu erheblichen Zeitaufwand führen. Wird in einer Applikation ein Großteil der Klassen von einer abgeleitet, so sollte man prüfen, ob es nicht besser ist, die Applikation umzudesignen und mehr Klassenkompositionen anstelle von Vererbung zu verwenden [GHJV94].

Eine weitere Problematik, mit der sich in dieser Arbeit auseinandergesetzt wurde, sind Vererbungshierarchien in den aufrufenden Klassen einer nachgeladenen Klasse. So können Instanzen einer Klasse auch auf Felder zugreifen, die in einer ihrer Superklassen deklariert sind und von ihr oder einer anderen Superklasse überschrieben wurden. Damit ein Zugriff auf alle Felder vom Typ der nachgeladenen Klasse auch nach dem Nachladen möglich ist, wurde das in JavAdaptor existierende Konzept der FieldContainer erweitert. Es wird nicht nur auf dem FieldContainer Feld, das in der Klasse selbst deklariert wird, gearbeitet, sondern auch mit den FieldContainer Feldern, die die Klasse der Instanz durch ihre Vererbungshierarchie besitzt.

Es existieren allerdings noch weitere Sprachkonzepte, die bisher durch JavAdaptor nicht unterstützt werden. Dabei handelt es unter anderem um das Unterstützen von Aufrufen, die mittels Reflexion durchgeführt werden. Gibt es textbasierte reflexive Aufrufe auf eine Klasse, die zur Laufzeit der Applikation schon unter neuem Namen nachgeladen wurde, so wird die originale Version der Klasse aufgerufen. Der Grund dafür liegt in der Umbenennung der Klassen. Wird eine Klasse nachgeladen, so müssten auch alle string-basierten reflexiven Aufrufe auf ihre Klassenrepräsentation dahingehend verändert werden, dass die aktuelle Version der Klasse aufgerufen wird.

Ein weiterer wichtiger Bereich, der durch JavAdaptor nicht unterstützt wird, sind konsistenz-erhaltende Nachladeprozesse. Ein Problem, was in diesem Zusammenhang gelöst werden muss, ist die Wahl eines geeigneten Nachladezeitpunkt für eine Klasse. Bei der Wahl diese Zeitpunktes müssen einige Seiteneffekte, die durch das Nachladen einer Klasse auftauchen können, beachtet werden. Wird die Funktionalität einer Methode verändert, während in einer Schleife auf Objekten der veränderten Klasse gearbeitet wird, so wurde mit einem Teil der Objekte die alte Funktionalität ausgeführt und mit den anderen Objekten die neue. Dadurch konnte eine Inkonsistenz in den Objekten der Klasse erzeugt werden. Ein anderes Problem entsteht bei der Zustandsübernahme der Instanzen der alten Klassenversion. So muss die Zustandsübernahme und das Redefinie-

ren der aufrufenden Klassen atomar geschehen. Ist dem nicht so, kann es passieren, dass nach der Zustandsübernahme und vor dem Redefinieren der aufrufenden Klassen noch auf Objekten der alten Klassenversion gearbeitet wird. Wird dabei der Zustand eines Felder des Instanzen verändert, sind in den Instanzen der neuen Klassenversion veraltete Werte. Da es sich bei diesen Problematiken nur um kleine Teilprobleme handelt, sind sehr hohe Anforderungen an einen Nachladeprozess gesetzt, wenn der Zustand die Applikation konsistent bleiben soll. Obwohl nie sichergestellt werden kann, ob alle möglichen Nachladeprozesse konsistent verlaufen können, besteht in diesem Bereich noch weiterer Forschungsbedarf, um das Nachladen von Klassen so konsistent wie möglich zu vollziehen.

A. Anhang

```
1  ...
2  allChangedClassesAndSubClasses = new ArrayList<ClassObject>();
3  while(allChangedClasses.size()>0){
4   ClassObject changedClass = allChangedClasses.remove(0);
5   if(!allChangedClassesAndSubClasses.contains(changedClass)){
6    allChangedClassesAndSubClasses.add(changedClass);
7   }
8   for(int i = 0; i < changedClass.getSubClasses().size();i++){
9    ClassObject currSubClass = changedClass.getSubClasses().get(i);
10     if(!allChangedClasses.contains(currSubClass) && !
           allChangedClassesAndSubClasses.contains(currSubClass)){
11      allChangedClasses.add(currSubClass);
12   }
13  }
14  for(int i = 0; i < changedClass.getImplementingClasses().size();i++){
15   ClassObject curImplementingClass = changedClass.getImplementingClasses
        ().get(i);
16     if(!allChangedClasses.contains(curImplementingClass) && !
           allChangedClassesAndSubClasses.contains(curImplementingClass)){
17      allChangedClasses.add(curImplementingClass);
18   }
19  }
20 }
21 ...
```

Listing A.1: Abgeleitete Klassen zur List der nachzuladenden Klassen hinzufügen.

Testumgebung

Zur Durchführung der Testszenarien, die im Kapitel 6 auf Seite 43 gezeigt werden, ist es nötig, eine Testumgebung zu entwickeln. Dabei muss es sich um eine lang laufende Applikation handeln, in der die Möglichkeit besteht, gleiche Codesegmente zu verschiedenen Zeitpunkten auszuführen. Aufgrund der Tatsache, dass diese Anforderungen von

grafischen Applikationen unterstützt werden, wurde eine solche Applikation entwickelt. In Abbildung A.1 ist ein Bild dieser Umgebung gegeben. Der dazugehörige Quellcode der Umgebung, in der die Testfälle ausgeführt wurden, ist in Listing A.2 gezeigt.

Abbildung A.1: Oberfläche der Testumgebung.

Da der grafische Teil der Applikation keinen Einfluss auf die durchgeführten Testszenarien hat, wurde er minimal gehalten. Bei ihm handelt es sich um die Klasse `MyFrame`, die von der Klasse `java.awt.Frame` abgeleitet ist. Um mit ihr die Testszenarien durchführen zu können, besitzt sie ein Feld vom Typ `CallerCaller` (Zeile 4). Wird der `Button` (Zeile 3) des Frames gedrückt, so wird in der Methode `buttonPushed(...)` (Zeile 20 - 22) das jeweilige Testszenario ausgeführt.

```
1  public class MyFrame extends Frame{
2
3      public Button button;
4      public CallerCaller  caller = new CallerCaller();
5
6      public MyFrame( ){
7              addWindowListener(new java.awt.event.WindowAdapter(){
                   public void windowClosing (java.awt.event.
                   WindowEvent evt) {System.exit(0);}});
8          setSize(100,100);
9          setVisible(true);
10         setResizable(true);
11         setLayout(new BorderLayout());
12         setBackground(Color.lightGray);
13
14         button = new Button("show");
15         add(button);
16          button.addActionListener(new ActionListener(){
17             public void actionPerformed(ActionEvent evt){
                   buttonPushed(evt);};});
18     }
19
20     public void buttonPushed(ActionEvent e){
21        caller.bar();
22     }
23
24     public static void main(String[]args){
25         MyFrame frame = new MyFrame( );
26     }
27 }
```

Listing A.2: Testumgebung - Quellcode der Klasse MyFrame.

Literaturverzeichnis

[Caz04] Walter Cazzola. Smartreflection: Efficient introspection in java. *Journal of Object Technology*, page 2004, 2004. (zitiert auf Seite 5)

[Chi00] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336, London, UK, 2000. Springer-Verlag. (zitiert auf Seite 13)

[Chi10] Shigeru Chiba. Javassist. Website, August 2010. Available online at http://www.csg.is.titech.ac.jp/~chiba/javassist/; visited on August 12th, 2010. (zitiert auf Seite 15)

[CN03] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc. (zitiert auf Seite 13)

[Dmi01] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *In Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, 2001. (zitiert auf Seite 4)

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994. (zitiert auf Seite 9 and 54)

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. (zitiert auf Seite 3)

[Hug97] Jim Hugunin. Python and java: The best of both worlds. In *in Proceedings of the 6th International Python Conference. CNRI*, pages 1997–10, 1997. (zitiert auf Seite 1)

[jdk08] JDK 6 java platform debugger architecture (JPDA)-related APIs & developer guides – from sun microsystems, June 2008. (zitiert auf Seite 13)

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *In Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98), volume 33, number 10 of ACM SIGPLAN Notices*, pages 36–44. ACM Press, 1998. (zitiert auf Seite 3)

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. (zitiert auf Seite 3)

[PKG+09] Mario Pukall, Christian Kästner, Sebastian Götz, Walter Cazzola, and Gunter Saake. Flexible runtime program adaptations in java - a comparison. Technical report, Fakultät für Informatik, Universität Magdeburg, 2009. (zitiert auf Seite 3 and 5)

[TIO10] Tiobe software. Website, October 2010. Available online at http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html; visited on October 7th, 2010. (zitiert auf Seite 1)

[Ull09] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Computing, eighth edition, 2009. (zitiert auf Seite 16)

[Ven99] B. Venners. *Inside the Java 2 Virtual Machine*. McGraw-Hill, 2nd. edition, 1999. (zitiert auf Seite 3)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 21.10.2010