Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik

Eine Bibliothek für Selbstmodifikationen zur Laufzeit in Java

Bachelorarbeit
Lehrstuhl Prof. Dr. Andreas Zeller

vorgelegt von

Christian Felix Bürckert

am 20. März 2012

Angefertigt und betreut unter der Leitung von
Dr. Frank Padberg

Begutachtet von
Dr. Frank Padberg
Prof. Dr. Andreas Zeller

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ...

(Datum / Date) (Unterschrift/ Signature)

Kurzzusammenfassung

Selbstmodifizierender Code (SMOC) - Programmcode der seine Instruktionen zur Laufzeit ändert - wurde bereits in der Assemblerzeit erfunden. Smalltalk machte den Schritt, die dafür notwendigen Operatoren - wie z.B. become: - zu konkretisieren. Die modernen Programmiersprachen Java und C#, bieten jedoch nur noch die Introspektion - die SMOC-Fähigkeit die Programmkonstrukte zur Laufzeit zu analysieren. Selbstadaptive Systeme tauschen zur Laufzeit ganze Komponenten aus, um Modifikationen zu ermöglichen. Diese Arbeit stellt einen Ansatz vor, die Granularität solcher Modifikationen zu verfeinern. Dazu werden die SMOC-Operatoren ermittelt, analysiert, konkretisiert und schließlich in einer Bibliothek (jSMOC) implementiert, um in Java Klassenmodifikationen zur Laufzeit durchführen zu können. Das bedeutet, Methoden und Felder können zur Laufzeit hinzugefügt und entfernt werden, die Klasse eines Objekts kann geändert werden kann und es ist möglich, alle Referenzen zu einem Objekt auf ein anderes Objekt umzulenken. Am Ende werden kleinere Programme vorgestellt, die zeigen wie man z.B. dynamische Softwareupdates zur Laufzeit mit jSMOC implementieren könnte.

Inhaltsverzeichnis

1 Einführung

Selbstmodifizierender Code (SMOC) ist ein Programmcode, der seine Instruktionen *zur Laufzeit* verändert [3]. SMOC ist bereits in Lisp und Smalltalk vorhanden. Weitere Quellen für Selbstmodifikationen findet man unter Linux[1], bei Viren [1] und aus der Zeit, in der Assembler programmiert wurde[2]. Introspektion - eine SMOC-Fähigkeit, die es ermöglicht, dass ein Programm seine Strukturen zur Laufzeit analysiert [3], tritt wieder in den neuen Programmiersprachen Java und C# in Erscheinung. Jedoch fehlen meist konkrete Funktionen, wie z.B. addMethod, um diese Strukturen zu verändern. SMOC könnte durch solche konkreten Funktionen helfen, dynamische und selbstadaptive Software zu schreiben. Im Moment basiert selbstadaptive und dynamische Software auf dem Austauschen ganzer Klassen und Komponenten.

Durch die Vielzahl aktueller Projekte wie JavAdaptor [17] und ObjectEvolution [7], die Bibliotheken zur Bytecode-Manipulation wie Javassist [6], BCEL [14] und ASM [4], aber auch durch die nicht implementierten Funktionen zur Klassenmodifikation im JVM-TI[3], erkennt man den Bedarf, zur Laufzeit mehr in das Geschehen eingreifen zu können. Jedoch macht keines der genannten Projekte den Sprung zurück zu der Dynamik und Funktionalität, Klassen über Metaklassen zur Laufzeit zu ändern, wie man es aus Smalltalk kennt. Ebenso möchte man nicht mehr in Smalltalk programmieren, da diese alte Sprache nicht mehr gelehrt wird und moderne Projekte in Java, C# oder C++ umgesetzt werden. Die logische Konsequenz dieses Dilemmas, ist es eine moderne Programmiersprache mit den aus Smalltalk bekannten Fähigkeiten auszustatten. Dies sollte natürlich selektiv geschehen, unter der Abwägung der Vor- und Nachteiele dieser Operationen. Das Implementieren diverser Selbstmodifikationsoperatoren in Java, die *einfach* zu verwenden sind, ist das Kernziel dieser Arbeit.

Eine der möglichen Gefahren selbstmodifizierender Software, ist der Umgang mit dem Löschen von Methoden. Dies kann zum Absturz der Software führen. Erlaubt man jedoch nur die Erweiterung von Klassen, so führt dies in langlebigen Programmen mit jeder Änderung zu unübersichtlicheren größeren Klassenkonstrukten. Genau diese Gefahren gilt es herauszustellen und abzuwägen, was ebenso einen Teil dieser Arbeit ausmacht.

Mit Hilfe der in dieser Arbeit entwickelten Funktionen können Forschungsergebnisse und Erfahrungen gesammelt werden, um selbstadaptive und dynamische Software zukünftig intuitiver zu gestalten. Dies könnte somit auch positive Auswirkungen auf deren Robustheit haben.

Ziele dieser Bachelorarbeit

- **Die Auswirkungen der SMOC-Operatoren analysieren.**

  SMOC-Operatoren können eine gravierende Auswirkung auf die Klassenstruktur haben, wenn zum Beispiel eine Methode einer Klasse entfernt wird. Wie können die anderen Klassen dies kompensieren? Diese Auswirkungen müssen für alle Operatoren analysiert werden. Probleme und die möglicherweise daraus entstehende Instabilität von Programmen muss gut verstanden sein, um zu entscheiden, ob die Umsetzung eines Operators überhaupt sinnvoll ist. Diese Entscheidung soll ein Teil der Analyse werden und den

[1] Webseite http://asm.sourceforge.net/articles/smc.html von Karsten Scheibler, Stand: Oktober 2011
[2] Webseite `http://www.hlasm.com/english/sampsnip.htm#self_modify` 13. Oktober 2011
[3] JVM - Tool Interface, eine Schnittstelle, um laufende Programme auf einer Java Virtual Maschine zu inspizieren. Wird z.B. von Debuggern genutzt.

später geplanten Ansatz der jSMOC-Bibliothek berücksichten. Am Ende soll als Ergebnis eine Liste sinnvoller, umsetzbarer SMOC-Operatoren stehen, die eine Entwicklung modifizierbarer Javaprogramme ermöglicht.

- **Die Umsetzung der SMOC-Operatoren in Java als Bibliothek.**

  Nachdem man eine Liste aller sinvollen Operatoren aufgestellt hat, gilt es diese unter einem Ansatz zu vereinen. Auftretende Seiteneffekte der Operatoren in diesem Ansatz werden herausgearbeitet und dargestellt. Beispielhafte Seiteneffekte sind, wie bei JavAdaptor [17], durch Codetransformation erzeugte Hilfsfelder und Hilfsklassen, sowie unerwartete Ergebnisse bei der Verwendung der javaeigenen Funktionen, wie z.B. die Introspektion.

  Die herausgearbeiteten Operatoren sollen in einer Bibliothek - jSMOC - bereitgestellt werden, mit der man einfach selbstmodifizierende Software schreiben kann. Die Erstellung einer solchen Bibliothek ist das Kernziel dieser Arbeit. Dazu soll jSMOC, ähnlich wie in Smalltalk, eine Metaebene in Form von Reflexionsklassen zur Verfügung stellen, welche die bereits im Paket java.lang.reflect unter Java vorhandene Introspektion widerspiegelt und diese durch eine Liste von Operatoren der Kategorien Programmreflexion, Klassenreflexion und Objektreflexion vervollständigt.

Abgrenzung

Das naheliegende Ziel, die *Java Virtual Machine* (JVM) anzupassen, um die genannten SMOC-Operatoren umzusetzen, wird in dieser Arbeit nicht verfolgt. Dies hat den Grund, dass die JVM zwar Open-Source ist, jedoch deren Implementierung oder Anpassung den Umfang einer Bachelorarbeit überschreitet. Bei der vorliegenden Arbeit, handelt sich also eher um eine konzeptionelle Vorarbeit, welche die Entwicklung oder Anpassung einer VM unterstützen kann.

Ebenso wird der Programmierer einige Richtlinien bei der Verwendung von jSMOC einhalten müssen, da der volle Umfang der Programmiersprache Java mit dieser Bibliothek nicht kompatibel ist. So wird unter anderem, wie im Kapitel 9.1 ausführlich beschrieben, auf folgende Sprachanteile der Programmiersprache Java verzichtet:

- Nebenläufigkeit: Die Verwendung von Caches und Locks - im Zusammenhang mit Manipulationen der Klassenstrukturen innerhalb der virtuellen Maschine - sind unklar. Nebenläufigkeitsprobleme sind schwer reproduzierbar und treten sporadisch auf. Nebenläufigkeit würde somit die Entwicklung maßgeblich erschweren. Ebenso könnte es sogar möglich sein, dass auftretende Nebenläufigkeitsprobleme aus der VM selbst stammen, da diese nicht für Selbstmodifikationen ausgelegt ist. Bei der Entwicklung von jSMOC wurde daher auf Synchronisationsmechanismen verzichtet. An geeigneter Stelle wird jedoch auf die kritischen Stellen hingewiesen, um die Nebenläufigkeit durch weiterführende Arbeiten wieder zu ermöglichen.

- Innere Klassen sind ein nicht zur Programmierung notwendiges Konstrukt. Vor allem nicht statische innere Klassen - also diejenigen, die für jede Instanz erzeugt werden - machen im Ansatz von jSMOC Probleme. Da bei Änderungen neue Klassen erzeugt werden und somit die innere Klasse kopiert würde, ist es später schwer zu sagen, auf welche innere Klasse eine Instanz referenziert. Wird die innere Klasse vom Programmierer ausgelagert, so treten diese Probleme nicht auf.

- Klassen, die modifizierbar sein sollen, werden durch die Implementierung einer leeren Schnittstelle als solche markiert. Als modifizierbar markierte Klassen dürfen jedoch nicht generisch sein. Generische Typen werden lediglich zur Compilerzeit geprüft und werden zur Laufzeit auf die allgemeine Oberklasse Object reduziert. Eine Modifikation generischer Typen macht zur Laufzeit also keinen Sinn. Theoretisch dürften diese generischen Typen auch keinen Einfluss auf die jSMOC-Bibliothek haben. Es hat sich jedoch herausgestellt, dass die Java Virtual Machine mit kritischen Fehlermeldungen an unerwarteten Stellen abstürzt, sobald generische Typen in modifizierbaren Klassen verwendet werden. Da der verursachende Fehler nicht lokalisiert werden konnte und möglicherweise ein VM-Fehler ist, muss auf generische modifizierbare Klassen verzichtet werden.

2 Stand der Technik

Selbstmodifizierender Code hat, wie in der Einleitung bereits erwähnt, eine hohe nicht konkretisierte Präsenz in aktuellen Programmen. Dieses Kapitel soll diese aufzeigen sowie die für die Entwicklung von jSMOC notwendigen oder mitwirkenden Bestandteile kurz einführen.

2.1 Programmiersprachen mit SMOC-Präsenz

SMOC ist in den bekannten Programmiersprachen nur teilweise, versteckt oder gar nicht vorhanden. Intercession ist die Fähigkeit, die Semantik der zugrundeliegenden Programmiersprache zur Laufzeit zu modifizieren [15]. Die Intercession ist für diese Arbeit nicht notwendig, gehört aber zu den SMOC-Fähigkeiten und vervollständigt somit den folgenden Überblick:

Sprache	Introspektion	Selbstmodifikation	Intercession
Assembler	nicht konkret	nicht konkret	nein
C/C++	nicht konkret	nicht konkret	nein
Delphi	nicht konkret	nicht konkret	nein
Smalltalk	ja	ja	nein
Lisp	ja	ja	ja
C#	ja	nein	nein
Java	ja	nein	nein

Legende

- ja: Die Programmiersprache beinhaltet konkrete Operatoren oder Methoden diese Fähigkeit umzusetzen.
- nein: Die Programmiersprache hat keine Möglichkeit, die Fähigkeit umzusetzen.
- nicht konkret: Es gibt keine Funktionen, die Fähigkeit kann jedoch über Tricks, wie zum Beispiel eine Manipulation des Arbeitsspeichers, umgesetzt werden.

2.1.1 Lisp

Lisp ist die einzige bekannte Sprache, die alle SMOC-Funktionen zur Verfügung stellt. Intercession wird über eine Schnittstelle zur Sprache erreicht - das Meta-Object Protokoll [11].

2.1.2 Assembler

Im Assembler unterscheidet man zur Laufzeit nicht zwischen Daten und Quellcode. Eine Modifikation des Programmablaufs kann genau so durchgeführt werden, wie die Modifikation eines Datensegments. Häufig werden Instruktionsmodifikationen aus Effizienzgründen vorgenommen.

Beispiel 1: [4]

```
INIT      BC     X'00',INITDONE
OI     INIT+1,X'F0'
-- Programmcode --
INITDONE EQU    *
```

Listing 1: SMOC in Assembler

In Beispiel 1 kommentiert sich ein Programmteil selbst aus. Dies wird umgesetzt, indem die Bedingung eines Sprungs von **false** auf **true** gesetzt wird. Stellt man den Sachverhalt, wenn dies nicht über eine Instruktionsmodifikation geschieht, in einer verständlichen Anweisungssprache dar, in der jede Zeile einer ausgeführten CPU-Anweisung entspricht, so erkennt man den Effizienzgewinn:

```
MAKE SPACE FOR BOOLEAN A          //nicht noetig in SMOC
LOAD BOOLEAN A TO REGISTER 1      //nicht noetig in SMOC
IF (REGISTER 1 == 0)              // IF (TRUE)
    STORE 1 in A                  //    MODIFY LINE 03: IF (FALSE)
-- Programmcode --
```

Listing 2: Vergleich der Anweisungen mit und ohne SMOC

In Listing 2, wurde in den Kommentaren die gleiche Funktionalität durch Instruktionsmodifikation dargestellt. Durch SMOC können Zeile eins und Zeile zwei gespart werden, Zeile drei benötigt eine andere Instruktion und Zeile vier enthält die Modifikationsinstruktion. Diese Modifikationsinstruktion macht die Verwendung einer Variablen unnötig.

Beispiel 2: [5]

```
; codeEnd  starting address
mov ecx, codeEnd-codeStart
; length of encrypted block

; now decrypt the code,
;starting from the last byte
decryptLoop:
xor byte [edi], al      ; decrypt byte
dec edi                 ; move to the next byte
loop decryptLoop

codeStart:
; put encrypted code here
codeEnd:
```

Listing 3: Selbstentschlüsselung mit Assembler

In Beispiel (Listing 3), wird das eigentliche Programm mit dem Hexadezimalwert $0x12$ verschlüsselt am Ende eingebunden. Zur Laufzeit wird der Programmcode dann entschlüsselt, eingebunden und dann gestartet. Mit diesem Verfahren können sich z.B. Viren vor der Erkennung schützen. Man kann aber auch Software vor nicht authorisierter Ausführung schützen, indem man den Key zum Entschlüsseln erst eingeben lässt. Da dann der eigentliche Quellcode

[4] Webseite `http://www.hlasm.com/english/sampsnip.htm#self_modify` 13. Oktober 2011
[5] Webseite `http://migeel.sk/blog/2007/08/02/advanced-self-modifying-code/`

verschlüsselt vorliegt und der Schlüssel nicht ausgelesen werden kann, ist das Authentifizierungsverfahren auch nicht durch einen Disassembler oder Hexeditor umgehbar.

In beiden Beispielen erfolgt die Modifikation jedoch nicht über dafür vorgesehene Instruktionen, sondern indem die Instruktionsbytes im Speicher manipuliert werden. Daher ist SMOC in Assembler nicht konkret. Will ein Assemblerprogramm sich inspizieren, so müssen die entsprechenden Instruktionsbytes im Speicher interpretiert werden.

2.1.3 C/C++ und Delphi

In C / C++, Delphi und anderen zur Maschinensprache übersetzbaren Hochsprachen findet sich SMOC vor allem in Bufferoverflows und JIT-Compilern.

Bufferoverflows werden dazu genutzt, um sich die Rechte eines Programms als Hacker anzueignen. Meistens sollen Administrationsrechte erlangt werden, damit sich Viren installieren können oder damit man Sicherheitsmechanismen deaktivieren kann. Bufferoverflows führen zu einer Modifikation des laufenden Programms. Durch eine unerwartete Eingabe werden Maschinensprachenbefehle in einen Buffer geladen. Durch fehlende Überlaufprüfung der Buffergrenzen werden unerwartete Speicherbereiche überschrieben. Damit können Sprungaddressen so manipuliert werden, dass die in den Buffer geladenen Anweisungen ausgeführt werden. Da das Programm die Eingaben selbst verarbeitet und dadurch seinen Programmfluss ändert, kann man hier von Selbstmodifikation sprechen. Ferner könnte ein Programm so auch Selbstmodifikationsoperatoren implementieren. Aus Komplexitätsgründen ist davon jedoch abzuraten.

Listing 4 soll diese Komplexität veranschaulichen. Es handelt sich um ein Programm, welches ein bestimmtes Zielprogramm mit root-Rechten ausnutzt, um einen Administrationsterminal zu öffnen. Die Modifikation muss auf dem Level der Maschinensprache ausgedrückt werden und findet sich in der Variablen shellcode. Das Programm wurde als Lösung zu einer Aufgabenstellung der Security-Vorlesung der Universität des Saarlandes eingereicht.

Beispiel:

```
1  static char shellcode[] =
2  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
3  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
4  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
5  //machine language: "execute /bin/sh"
6
7  int main(void)
8  {
9  char *args[3];
10 char *env[1];
11
12 //call arguments
13 args[0] = TARGET; //target application with root rights
14 args[1] = "";     //will be set to buffer later
15 args[2] = "";     //program wants 2 arguments :)
16
17
18 char buffer[200]; //generate too large buffer
19 //[TARGET expects 127 bytes]
20
21 //fill shellcode into buffer
```

```
22 int j=0;
23 int i=0;
24 for (; i<127; i++){
25     if (i < sizeof(shellcode)-1){
26         buffer[j++] = shellcode[i];
27     } else {
28         //fill the buffer with something different than 0x00
29         buffer[j++] = 'A';
30     }
31 }
32 //insert new jump address 0xbf.ff.fd.78
33 //at the correct position (bufferstart+127)
34 //this will overwrite the return address on
35 //the stack to the beginning of the buffer
36 buffer[j++] = (char)0x78;
37 buffer[j++] = (char)0xfd;
38 buffer[j++] = (char)0xff;
39 buffer[j++] = (char)0xbf;
40
41 //call the application with the too large input
42 args[1] = buffer;
43 env[0] = NULL;
44 execve(TARGET, args, env);
45 return 0;
46 }
```

Listing 4: Bufferoverflow-Angriff auf eine Anwendung mit root-Rechten

JIT-Compiler werden genutzt, um die Ausführungsgeschwindigkeit von interpretierten Sprachen zu beschleunigen. Sie kompilieren diese interpretierten Sprachen in Maschinensprache und adaptieren so die in der interpretierten Sprache implementierte Funktionalität. JIT-Compiler laden dazu die Maschinensprachinstruktionen in Buffer und springen gezielt in diese Buffer zur Ausführung der dann darin enthaltenen Instruktionen. Probleme bekommen JIT-Compiler jedoch mit Bufferoverflow-Schutzmechanismen, welche die Ausführung von Instruktionen in Datensegmenten verbieten. Da der JIT-Compiler Funktionalität zur Laufzeit adaptiert und ausführt, kann man von Selbstmodifikation sprechen.

Die kompilierten Hochsprachen wie C/C++ und Delphi bieten jedoch keine elementaren Funktionen an, um Modifikationen durchzuführen. Man muss den Speicher manuell interpretieren und verändern. Eine mögliche Ursache, warum auf solche Modifikationsoperatoren verzichtet wird, ist dass diese in der Maschinensprache ebenfalls nicht existieren. Objektorienterte Strukturen werden vom Compiler fast völlig zerstört. Eine Introspektion oder Modifikation ist zur Laufzeit daher nicht mehr möglich. Introspektion zu ermöglichen, bedeutet einen Disassembler zu entwickeln, der zur Laufzeit zurück in die Hochsprache übersetzt. Alternativ könnte man zur Introspektion auch die Klassenstrukturen zusätzlich in geeigneter Form abspeichern.

2.1.4 Smalltalk

Smalltalk bietet als objektorientierte Sprache Funktionalität zur Selbstmodifikation. Hierfür werden neben become: auch andere Funktionen bereitgestellt [9]. So besitzt in Smalltalk jede Klasse eine Metaklasse, über die man die Methoden der Klasse verändern kann. Smalltalk nutzt become:, um den Typ von Objekten zu ändern. Somit kann z.B. ein volles Array zu einer Liste werden, wenn ein weiteres Element hinzugefügt wird [12].

2.2 Das JVM Tool Interface

Das Java Virtual Machine Tool Interface (JVM-TI) ist der Zusammenschluss des JVM Debugger Interface (JVM-DI) und JVM Profiler Interface (JVM-PI). Das JVM-TI ist die allgemeine Schnittstelle, welche Debugger und andere Laufzeitanalysetools nutzen müssen, um Informationen zu laufenden Programmen zu bekommen oder auf deren Ausführung einzuwirken[6]. Die Verbindung zu dieser Schnittstelle erfolgt über Sockets oder SharedMemory.

Das JVM-TI ermöglicht die komplette Introspektion laufender Programme. Die Javaimplementierung dieser Schnittstelle verwendet jedoch eigene Introspektionsklassen und nicht jene des java.lang.reflect-Pakets. Das JVM-TI ermöglicht es auch, alle Instanzen einer Klasse zu inspizieren, eine Fähigkeit, die dem java.lang.reflect-Paket zur Introspektion zur Laufzeit fehlt. Das java.lang.reflect-Paket hat jedoch die Möglichkeit, auch den Inhalt von konstant deklarierten Feldern zu verändern. Das JVM-TI beachtet diesen Zugriffsmodifikator. Andere Zugriffsmodifkatoren, wie z.B. private, beachten weder JVM-TI, noch das java.lang.reflect-Paket.

Das JVM-TI erlaubt es Klassen neu zu definieren. Hierfür wird die Funktion redefineClass angeboten. Die Mächtigkeit zur Klassenmodifikation dieser Methode gliedert sich dabei in drei sich jeweils erweiternde Stufen:

1. Stufe: Verändern von Klassen, solange sich die Klassensignatur nicht ändert. Das bedeutet, es können nur Methodenrümpfe ausgetauscht werden.
2. Stufe: Verändern von Klassen, solange nur Methoden hinzugefügt werden.
3. Stufe: Beliebiges Verändern von Klassen.

Um zu prüfen, welche dieser Stufen eine Virtual Machine bereit stellt, gibt es in der JVM-TI drei Funktionen: canRedefine, canAddMethod und canUnrestrictedlyRedefineClasses. Man merkt, dass die Entwickler des JVM-TI die Probleme der Modifikation in drei Stufen eingeteilt haben und bereits auf nicht umgesetzte Stufen eingehen. Bis jetzt (20. Oktober 2011) ist nämlich keine Virtual Machine zu finden, die bei canAddMethod oder canUnrestrictedlyRedefineClasses true zurück liefert. Die Methode canRedefine liefert zumindest in der OpenJDK und Sun VM true.

Zur einfachen Verwendung von redefineClass bietet Javassist [5] die HotSwapper Implementierung[7] an. Beim HotSwapper baut das Programm eine Verbindung zur ausführenden JVM über das JVM-TI auf und ermöglicht es, Klassen auszutauschen.

Das JVM-TI erlaubt auch die Reflexion des Stacks. Die Modifikation eines Stacks ist jedoch nur erlaubt, wenn der zugehörige Thread angehalten ist. Da es möglich ist, sich aus dem Programm heraus zur ausführenden JVM zu verbinden, kann ein Programm den eigenen Stack verändern. Dazu muss diese Manipulation jedoch in einen seperaten Thread ausgelagert werden.

Die JVM muss mit zusätzlichen Parametern gestartet werden, damit die Verbindung zum JVM-TI aufgebaut werden kann[8].

[6] `http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/#0` (22. Februar 2012)

[7] `http://www.csg.is.titech.ac.jp/~chiba/javassist/html/javassist/util/HotSwapper.html` (22. Februar 2012)

[8] `http://www.techrepublic.com/article/debug-your-java-code-with-ease-using-jpda/6139512` Parameterdefinitionen

2.3 Bytecode-Manipulation

Der Java Bytecode (JBC) ist eine Zwischensprache, in die nicht nur Java, sondern unter anderem auch Ruby und Jasmin[9] übersetzt werden. Der JBC wird von einer virtuellen Maschine - einer Stackmaschine - ausgeführt. Er beinhaltet im Gegensatz zur Maschinensprache die Klassenstrukturen. Lediglich die Methodenrümpfe haben eine lineare maschinensprachliche Form. Der JBC wird in Klassendateien gespeichert, wobei jede Klasse - außer den inneren Klassen - ihre eigene Datei zugewiesen bekommt. Die Bytecode-Manipulation analysiert diese Dateien und bietet Wege an, die beinhalteten Klassenstrukturen zu verändern, bevor diese an den ClassLoader weitergegeben werden. Dies ermöglicht es, die Klassen zur Laufzeit vor der ersten Verwendung beliebig anzupassen. Ebenso können neue Klassendateien und somit auch Klassen zur Laufzeit erzeugt und geladen werden.

Neben den drei großen Bibliotheken zur Bytecode-Manipulation ASM [4], Javassist [6] und BCEL [14] gibt es auch noch JMangler [13] und DataScript [2]. Laut Chiba und Nishizawa [6], den Entwicklern von Javassist, ist der wesentliche Vorteil von Bytecode-Manipulation gegenüber Quellcode-Manipulation die Fähigkeit, Programme und Bibliotheken, die ohne Quellcode ausgeliefert werden, zu verändern. Nachteil ist, dass man dazu weitreichende JBC-Erfahrungen braucht.

Obwohl alle Bibliotheken zur Bytecode-Manipulation gleich mächtig sind, haben sie unterschiedliche Vor- und Nachteile in der Verwendung, und es bietet sich in der vorliegenden Arbeit an, zwei dieser Bibliotheken - ASM und Javassist - zu benutzen, um die Funktionalität möglichst einfach zu implementieren.

2.3.1 ASM

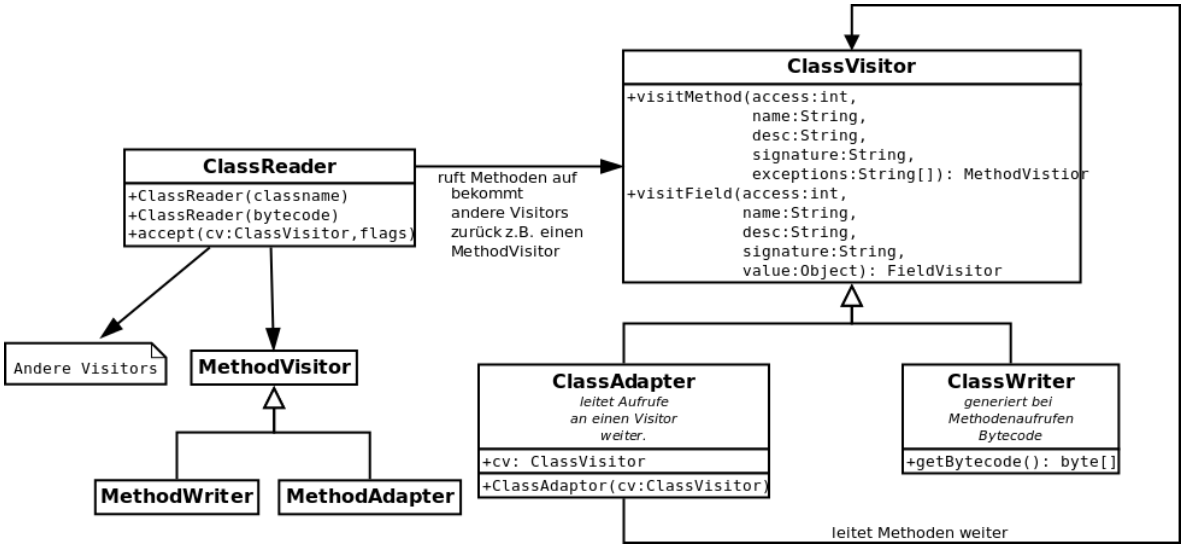


Abbildung 1: ASM Besuchermusterkonzept

ASM ist die schnellste Bibliothek zur Bytecode-Manipulation[10]. Sie liefert einen ClassRea-

[9] Eine assemblerartige Sprache

[10] `http://blog.ubigrate.de/2008/04/18/java-bytecode-engineering/` Nicht die Bohne kompliziert: Java Bytecode-Engineering (25. September 2011)

der, der über den JBC iteriert, und ruft an jeder Stelle über das Besuchermuster (engl. VisitorPattern[11]) eine entsprechende Methode auf. Diese wird dann an einen ClassWriter weitergeleitet, der die Klasse neu schreibt. Zwischen ClassReader und ClassWriter können beliebige selbst implementierte ClassAdapter eingehängt werden, die beim Iterieren den JBC direkt verändern (vgl. Abb. 15 - S. 76). ASM eignet sich hervorragend, um Strukturen, die eine eigene Methode im Besuchermuster haben, auszutauschen oder zu verändern. Durch zusätzliche Aufrufe oder nicht Weiterleiten an den ClassWriter kann man auch Klassen sehr gut um solche Strukturen erweitern oder reduzieren. ASM setzt ab der Verwendung der Methodenbesucher tiefreichende Kenntnisse über den JBC voraus. ASM eignet sich außerdem hervorragend, um Bytecode zu analysieren. Dazu hängt man anstatt der Adapter eigene Visitors ein und sammelt Informationen (vgl. Abb. 16 - S. 77).

Abbildung 1 zeigt stark vereinfacht das Besuchermusterkonzept von ASM beschränkt auf die Methoden visitMethod und visitField. Listing 5 zeigt, wie man die Klasse Otto mit Hilfe des ASM TraceClassVisitors in lesbaren Bytecode übersetzt. Listing 6 zeigt die gekürzte Ausgabe des TraceClassVisitors. Der TraceClassVisitors ist ein von ASM mitgeliefertes Analysewerkzeug um den Bytecode sichtbar zu machen. Die Implementierung des TraceClassVisitors macht für jeden Aufruf einer der Besuchermustermethoden eine entsprechende textuelle Ausgabe auf dem übergebenen PrintWriter.

```
1 ClassReader  cr = new ClassReader("Otto"); //Creates a reader
2 //for class Otto
3 PrintWriter  pw = new PrintWriter(new File("Otto.bytecode.txt")); //output
4 ClassVisitor cv = new TraceClassVisitor(pw);
5
6 cr.accept(cv, ClassReader.EXPAND_FRAMES);
```

Listing 5: Ausgabe einer lesbaren Version eines Bytecodes mit ASM

```
1
2 public class Otto { // access flags 0x21
3 // access flags 0x1
4 public <init>()V
5 //... (removed)
6
7 // access flags 0x1
8 public sayHello()V
9 L0
10 LINENUMBER 11 L0
11 GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
12 LDC "Otto: Hello"
13 INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
14 L1
15 LINENUMBER 12 L1
16 RETURN
17 L2
18 LOCALVARIABLE this LOtto; L0 L2 0
19 MAXSTACK = 2
20 MAXLOCALS = 1
21 }
```

Listing 6: Inhalt von Otto.bytecode.txt (gekürzt)

[11] http://de.wikipedia.org/wiki/Visitor

Führt man eine Modifikation durch, so übergibt man an ClassReader.accept() einen vorgefertigten oder selbst entwickelten ClassAdapter. Der ClassAdapter leitet alle Aufrufe an einen übergebenen ClassVisitor weiter. Nimmt man als ClassVisitor einen ClassWriter, so erzeugt dieser aus den Aufrufen Bytecode. Da ClassAdapter auch ClassVisitor sind, kann man diese beliebig verketten. Durch das Ableiten eines ClassAdapters hat man die Möglichkeit, Aufrufe weiterzuleiten, verändert weiterzuleiten, nicht weiterzuleiten oder zusätzliche Aufrufe zu generieren. Exemplarisch zeigen Listing 7 und Abbildung 2 die Änderung eines Methodennamens in ASM. Dabei wird eine Kette von ClassReader, ClassAdapter und ClassWriter gewählt.

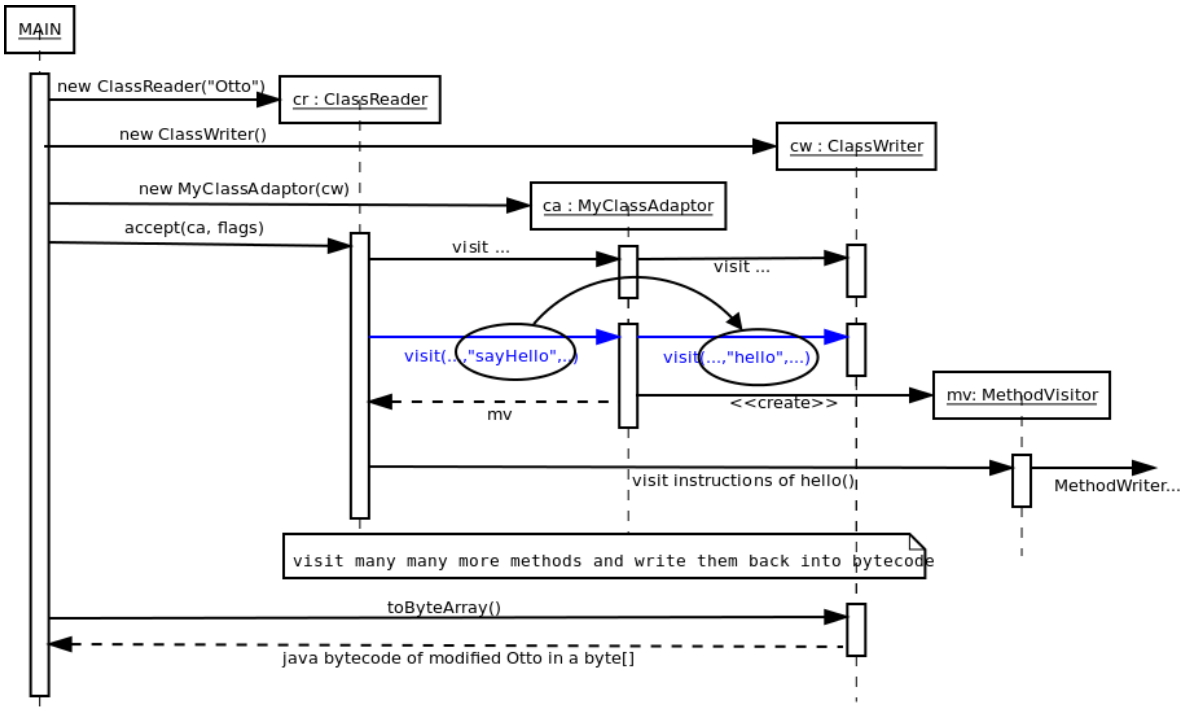


Abbildung 2: ASM-Manipulation sayHello() –> hello() als Sequenzdiagramm

```
ClassReader cr = new ClassReader("Otto");
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES
    | ClassWriter.COMPUTE_MAXS);
ClassVisitor ca = new ClassAdapter(cw) {
    public MethodVisitor visitMethod(int access, String name,
                    String desc, String signature, String[] exceptions) {
        if (name.equals("sayHello")) { //method found
            //change name into "hello"
            return super.visitMethod(access, "hello", desc, signature,
                            exceptions);
        } else {//ignore other methods
            return super.visitMethod(access, name, desc, signature,
                            exceptions);
        }
    }
};
cr.accept(ca, ClassReader.EXPAND_FRAMES);
byte[] bytecode = cw.toByteArray(); // return the manipulated bytecode
```

Listing 7: Ändern eines Methodennamens in der Klasse Otto von sayHello() in hello()

ASM wurde im Jahr 2000 durch Eric Bruneton entwickelt und diente anfangs lediglich dazu, dynamisch Proxyklassen zur Laufzeit zu erzeugen. Später wurde das ClassReader- und Besuchermusterkonzept hinzugefügt, um allgemein Klassen zu verändern oder zu erzeugen. 2002 wurde ASM ein OpenSource-Projekt und wurde durch Eugene Kuleshov (2003), Andrei Loskutov (2004) und Rémi Forax (2008) weiterentwickelt. ASM wird bis heute regelmäßig aktualisiert[12].

2.3.2 Javassist

Shigeru Chiba motiviert die Entwicklung von Javassist aus der Idee, Bytecode-Manipulation ohne JBC-Erfahrungen durchführen zu können. Nach der Analyse einer Klassendatei bietet die Bibliothek eine Metaklasse an, über welche die repräsentierte Klasse durch einfache Methodenaufrufe analysiert und verändert werden kann. Die Veränderungen selbst werden als Quellcodestrings angegeben. Diese verarbeitet Javassist unter der Verwendung von jinline [18] und OpenJava [19] und kann somit ohne JBC-Wissen auskommen.

Javassist eignet sich hervorragend dazu, Änderungen durchzuführen, die sich über der Anweisungsebene befinden; also das Verändern von Methoden, Feldern und Interfaces, jedoch nicht das Austauschen von z.B. Methodenaufrufen oder Feldzugriffen. Das Hinzufügen von Anweisungen am Anfang oder am Ende von Methodenrümpfen ist ebenfalls durch einen Methodenaufruf einfach erreichbar.

Javassist ist langsamer als ASM[13]. Die Idee von Javassist entstand auf einem Workshop für Reflektive Programmierung in C++ und Java 1998 [5]. In den folgenden Jahren bis 2003 stellte Shigeru Chiba die Anwendungsbereiche von Javassist in der aspekt-orientierten Programmierung vor. Inzwischen ist Javassist ein Teil des Apache JBoss Anwendungsservers und die aktuelle Version wurde neulich (am 8. Juli 2011) unter MPL, LGPL und Apache Licence veröffentlicht[14]. Javassist wird unter anderem für GluonJ[15], ein einfaches AOP-Framework, benutzt.

Abbildung 17 (Seite 78) zeigt, wie man eine Methode zu einer Klasse hinzufügt und sie unter neuem Namen zur Laufzeit kompiliert und einbindet.

2.4 ObjectEvolution

ObjectEvolution [7] von Cohen und Gil (Google Israel) beschäftigt sich damit, Objekte einer Klasse zur Laufzeit in andere Klassen zu überführen. Cohen und Gil geben dafür drei theoretische Ansätze an: die I-Evolution, die M-Evolution und die S-Evolution. Die I-Evolution (I für „Inheritence“) zum Beispiel ermöglicht es, Objekte v eines statischen Typs C in jede Unterklasse von C zu evolvieren. Ist C ein Interface, so darf v in jede implementierende Klasse evolvieren. Als Beispiel stellen Cohen und Gil eine Liste vor, die nach dem Hinzufügen einiger Elemente für Änderungen gesperrt wird, indem man sie in eine blockierte Liste evolviert.

Die M-Evolution und S-Evolution benötigen für eine kurze Einführung die umfangreichen Erklärungen zu „Mixins“ und „Shakeins“. Daher wird an dieser Stelle nur eine Idee vermittelt: So geht es zum Beispiel darum, Objekte transaktionssicher zu machen, indem man sie in ein Transactional shakein evolviert.

[12] `http://asm.ow2.org/history.html` History and Changes on the official ASM Website (29. Oktober 2011)
[13] `http://blog.ubigrate.de/2008/04/18/java-bytecode-engineering/`
[14] `http://www.csg.is.titech.ac.jp/~chiba/javassist/` Projekthomepage
[15] http://www.csg.is.titech.ac.jp/projects/gluonj/ Projekthomepage

Cohen und Gil führen in ihrem Artikel zu ObjectEvolution [7] drei Vorschläge zur praktischen Umsetzung ein; sehr kurz und ohne weitere Details. Die erste vorgestellte Variante ist die Gilgul VM [8] - eine virtuelle Maschine für Java, die mit Hilfe eines #=-Operators alle Referenzen zu einer Instanz auf eine andere Instanz umsetzen kann. Die zweite Möglichkeit sehen Cohen und Gil darin, Objekte während einer RAM-Defragmentierung zu ersetzen. Die dritte und letzte Möglichkeit haben Cohen und Gil benutzt, um die ObjectEvolution zu implementieren[16]. Es handelt sich um eine Variante, bei der jede Klasse ein newRef Feld via Codetransformation erhält. Mit Codetransformation werden dann ebenfalls alle Methodenaufrufe an newRef weitergeleitet, sofern dieses nicht null ist. Ansonsten wird die reguläre Implementierung aufgerufen.

Laut der Projekthomepage[17] ist die Implementierung thread-sicher und gibt an, wie diese in nebenläufigen Prozessen zu verwenden ist. Man könnte SMOC auf eine ähnliche Weise programmieren. Nachteilig ist jedoch, dass die Java Reflexion dann unerwartete Ergebnisse liefert, da sie eben nicht auf newRef achtet. Ebenso sind öffentliche Felder schwierig auf newRef umzuleiten. Das Löschen von Methoden ist auf diese Weise auch nicht umsetzbar.

2.5 Die Gilgul VM

Die Gilgul VM [8][18] ist eine Virtual Machine für die Programmiersprache Java und unterstützt JDK 1.3. Gilgul basiert auf der Kaffe OpenVM[19] und erweitert Java um den #=-Operator. Dieser Operator ermöglicht es, alle Referenzen eines Objekts zu tauschen (Abbildung 3). Damit dies effizient funktioniert, erhalten alle Referenzen eine zusätzliche Indirektion. Wie man in Abbildung 3 sieht, zeigen die Referenzen ref1 und ref3 des Objekts o1 auf einen Speicher, der dann auf das eigentliche Objekt o1 zeigt. Durch den Befehl ref1 #= ref2 verschiebt sich der indirekte Zeiger von o1 auf o2. Das bedeutet, ref1 und ref3 zeigen danach auf o2. Costanza, der Entwickler der Gilgul VM, nennt dies eine „referent assignment expression“ [8] also einen Referenzzuweisungsausdruck.

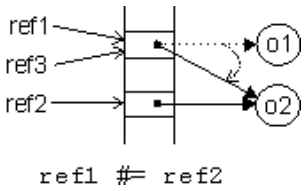


Abbildung 3: #=

Die Gilgul VM bietet den weiteren Klassenmodifikator typeless. „Ist eine Klasse oder ein Interface typeless, wird der Compiler jeden Versuch, den Namen als statischen Typ zu verwenden, zurückweisen.“[20]

```
1 typeless class MyClass {
2     void m() { ... }
3 }
4 MyClass obj = ...              // compile-time error!
5 Object obj = new MyClass();    // legal.
6 ((MyClass)obj).m();            // legal.
7 void n(MyClass mc) { ... }     // compile-time error!
```

Listing 8: Beispiel der Gilgul-Projektwebseite zu typeless-Klassen

[16] Quellcode und Implementierung sind nicht verfügbar. Die Links auf der Website zeigen auf leere Zip-Kontainer

[17] http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Object_Evolution

[18] http://javalab.cs.uni-bonn.de/research/gilgul/ Projektwebseite

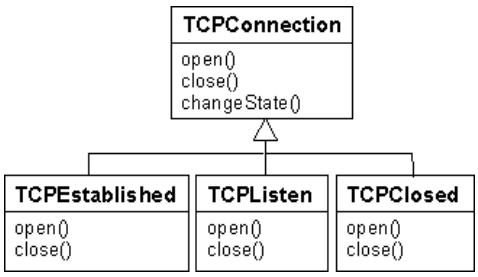
[19] http://www.kaffe.org/ Projektwebseite

[20] Übersetztes Zitat http://javalab.cs.uni-bonn.de/research/gilgul/languageDocumentation.htm/

Die Projektseite erklärt gültige Referenzzuweisungen wie folgt:[21] Ein Referenzzuweisungsausdruck lref #= rref mit Laufzeittypen L von lref und R von rref ist genau dann eine gültige Laufzeitanweisung, wenn mindestens einer der folgenden Punkte erfüllt ist:

- L und R sind gleich oder R ist eine Unterklasse von L. (additive replacement)
- L ist typeless und es gibt eine gemeinsame Oberklasse S von L und R. Jede Oberklasse von L, welche eine eigene Unterklasse von S ist, ist typeless. Die Menge der von L implementierten typless Schnittstellen ist eine Teilmenge der von R implementierten Schnittstellen. (subtractive replacement)

Als Beispiel für die Verwendung gibt die Gilgul VM Webseite die Implementierung eines Zustandsmusters an:




```
1  abstract class TCPConnection {
2  void changeState
3      (TCPConnection newState) {
4      this #= newState;
5      }
6  }
7  TCPConnection connection =
8  new TCPListen(...);
9  connection.changeState
10 (new TCPEstablished(...));
```

Die Gilgul VM ermöglicht keine Modifikationen im Sinne von SMOC. Jedoch ist der #=-Operator für die Entwicklung von jSMOC von bedeutendem Interesse. Mit seiner Hilfe kann man effizient Instanzen tauschen. In Kombination mit Bytecode-Manipulationsbibliotheken, die das Erstellen von Klassen zur Laufzeit ermöglichen, könnte man Selbstmodifkationen umsetzen. Da jedoch die Gilgul VM nur JDK 1.3 (aktuell ist JDK 1.7 vom Februar 2012[22]) unterstützt und seit der Publikation auf der AOSD 2002 nicht weiterentwickelt wurde, kann man die Gilgul VM als veraltet bezeichnen, und sie ist somit als Grundlage für die Entwicklung von jSMOC nicht geeignet.

Die Idee, Instanzen durch Indirektion schnell austauschen zu können, fließt in die Entwicklung der jSMOC-Bibliothek über die SMOCInstanceWrapper ein.

[21] Übersetztes Zitat http://javalab.cs.uni-bonn.de/research/gilgul/languageDocumentation.htm/
[22] Oracle Webseite http://www.oracle.com/technetwork/java/javase/7u3-relnotes-1481928.html

2.6 JavAdaptor

JavAdaptor ist ein Projekt, welches es ermöglichen soll, laufende Anwendungen weiter zu programmieren, ohne diese neu zu starten. Dazu gibt es ein Eclipse-Plugin und ein Rahmenwerk. Eclipse verbindet sich über das JVM-TI zu dem im Programm laufenden Rahmenwerk und spielt neue Klassenversionen ein. Da geladene Klassen in Java nicht ersetzt werden dürfen[23], werden die Klassen unter einem neuen Namen geladen. Dazu wird einfach die Version an den Namen angehängt (Name, Name_V1, Name_V2, ...). Die Instanzen der alten Klassen werden mit Hilfe des JVM-TI gesucht und in neue Instanzen der neuen Klassenversion transferiert. Dazu wird die Feldbelegung anhand des Feldnamens in die neuen Instanzen kopiert, sofern der Typ noch kompatibel und das Feld noch vorhanden ist. Um diese neuen Instanzen in das Programm zu integrieren, erzeugt JavAdaptor Proxy- und Adapterklassen. Die Referenzen zu diesen Hilfsklassen werden durch eine Codetransformation vorbereitet, indem alle Klassen für jedes ihrer Felder ein Hilfsfeld eingeschleust bekommen, welches dann die Proxy- und Adapterklassen aufnehmen kann. Die genaue Vorgehensweise ist in einem technischen Paper [17] und einer Publikation [16] mit einigen korrigierbaren technischen Fehlern[24] beschrieben. Laut den Entwicklern hat JavAdaptor keine Einschränkungen in Fexibilität, Performanz und Architektur. Sie demonstrieren die Funktionalität in einem Video[25], indem sie das Spiel „Snake“ zur Laufzeit umprogrammieren.

Nachteilig ist, dass weder der Quellcode noch das Plugin verfügbar sind. Ebenso ist das Proxy- und Adapterklassenkonzept sehr kompliziert und führt unter der Verwendung von Java Reflexion zu unerwarteten Ergebnissen. Die Aussage, die Architektur sei uneingeschränkt, ist somit falsch. Auf Reflexion aufbauende Architekturen sind nur eingeschränkt nutzbar, da der Programmierer weder mit den neuen Klassenfeldern noch den neuen Klassenversionen rechnet.

JavAdaptor ist somit für eine intuitive SMOC-Implementierung nicht geeignet. Die uneingeschränkte Reflexion ist für die Selbstmodifikation von entscheidender Bedeutung und somit unverzichtbar. Außerdem steuert JavAdaptor viele Modifikationen aus dem externen Plugin heraus. Es ist unklar, ob diese Algorithmen in das laufende Programm selbst übertragen werden können. Trotzdem verwendet JavAdaptor ähnliche Techniken in seinen Ansätzen, wie zum Beispiel das Kopieren der Feldbelegung einer Instanz anhand der Namen in eine andere neue Instanz.

[23] Das JVM-TI erlaubt nur das Ersetzen von Methodenrümpfen einer Klasse. Die Signatur der Klasse darf sich nicht verändern

[24] Ein Diagramm z.B. zeigt den Zugriff auf ein Feld in einer Schnittstelle. Dieser Fehler ist durch einen hinzugefügten Cast zu vermeiden.

[25] `http://www.youtube.com/watch?v=jZm0hvlhC-E`

3 Analyse der SMOC-Operatoren

Die Verwendung von SMOC in Programmen, Bibliotheken und Programmiersprachen findet meist unkonkret statt. Es gibt keine elementaren Operatoren oder Funktionen zur Modifikation. Viele Systeme bieten auch keine elementare Introspektion. Dieses Kaptiel soll die in der Einführung vorgestellten und in Funktionen gekapselten Operatoren der Kategorien Programmreflexion, Klassenreflexion und Objektreflexion ermitteln und konkret als solche aufstellen. Dies geschieht exemplarisch an der Programmiersprache Java und unter der Berücksichtigung des angestrebten Ansatzes zur Entwicklung der jSMOC-Bibliothek. Es wird jedoch versucht, weitestgehend allgemein auf objektorientierte Sprachen einzugehen. Ziel ist es, eine konkrete Liste von Modifikationsoperatoren herauszuarbeiten und deren Problematik zu erläutern. Dabei ergibt sich eine Einteilung in drei Hauptkategorieren: Programm, Klasse und Objekt sowie die Nebenkategorien: Methode, Feld, Parameter und Codeblock. Man bezeichnet diese Kategorien als Nebenkategorien, da ihre Umsetzung in eigene Operatoren sich nicht als sinnvoll herausgestellt hat. Den Typ eines Methodenparameters zu ändern, ergibt zum Beispiel nur einen Sinn, wenn der Methodenrumpf mit geändert wird. Es reicht also, einen Operator anzubieten, der Methoden löscht, und einen, der Methoden hinzufügt.

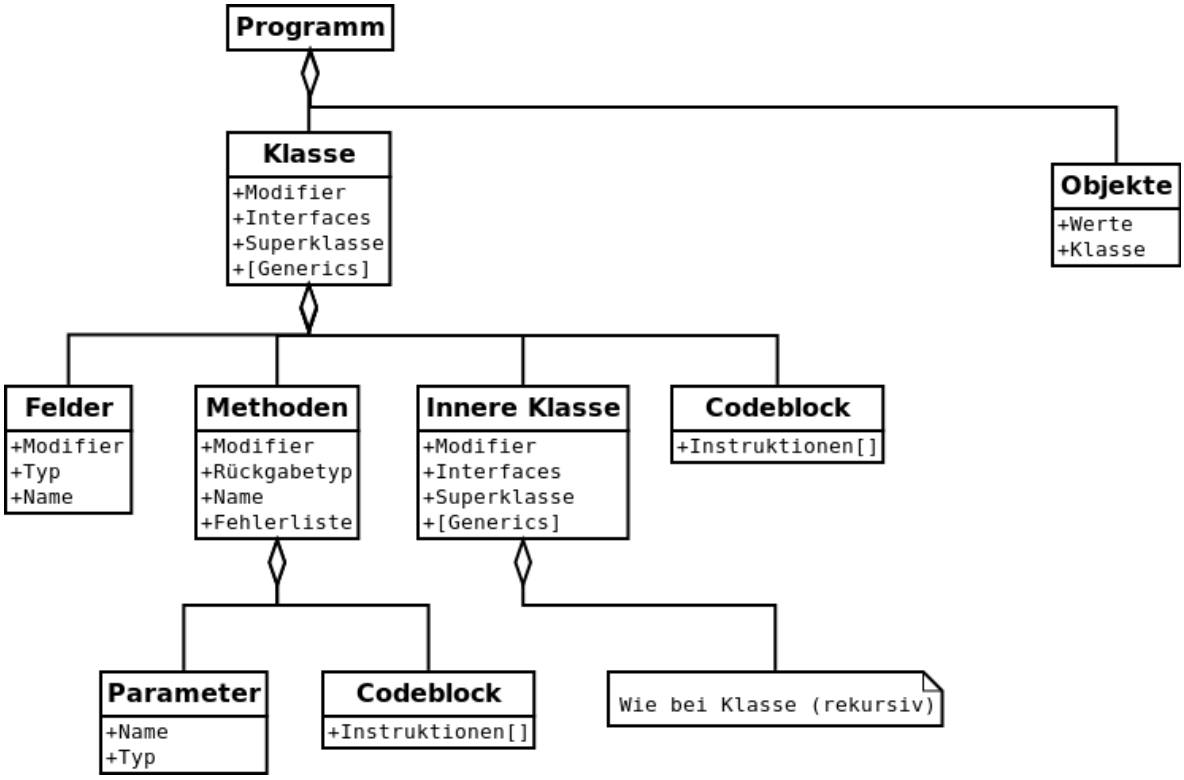


Abbildung 4: Programmaufbau in Java

Der Aufbau eines Programms sollte dabei als Grundlage zur Strukturierung der Reflexion gewählt werden. Der Programmaufbau unter Java wird in Abbildung 4 dargestellt. An dieser Abbildung orientiert sich die Analyse. Die konkreten SMOC-Operatoren stellen die Liste der Operatoren dar, die notwendig sind, um alle Elemente dieser Grafik zu reflektieren. Das heißt, die Introspektion und Modifikation der Elemente soll in konkrete Operatoren gekapselt

werden, sofern sie sinnvoll sind. Ein Operator ist dann sinnvoll, wenn er allein ausgeführt ein fehlerfreies Programm ergeben kann, das eine andere Semantik besitzt als vor der Ausführung des Operators.

3.1 Kategorie Programm

Die Kategorie des Programms wird in Java vernachlässigt. Es gibt keine Möglichkeit, alle Klassen anzuzeigen oder alle Instanzen einer Klasse zurückzugeben. Die Reflexion - und speziell die Introspektion - fängt in Java bei den Klassen an. Eine konkrete Überordnung des Programmkontextes findet nicht statt. Daher muss der Programmkontext konkretisiert werden. Wie man dem Programmaufbau entnehmen kann, gehört zu dieser Kategorie das Ermitteln, Hinzufügen, Löschen und Tauschen von Klassen und Objekten sowie die Rückgabe von deren Reflexionsschnittstellen.

3.1.1 Klassen ermitteln

Es gehört zu einer der grundlegenden Introspektionsfähigkeiten alle geladenen Klassen zu ermitteln. In Java sollte diese Fähigkeit dem ClassLoader zugeschrieben werden, da dieser das Laden übernimmt. Allerdings bietet der ClassLoader keine Methode, um alle von ihm geladenen Klassen zu ermitteln. Dies könnte an der fehlenden Konkretisierung des Programmkontextes liegen. Von diesem Operator gehen keine Gefahren aus. Er wird in jSMOC daher im Programmkontext als konkreter Operator angeboten.

3.1.2 Klassen hinzufügen

Java lädt seine Klassen dynamisch bei der ersten Verwendung. Dies geschieht über den ClassLoader, der eine kompilierte Klassendatei nutzt, um eine entsprechende Klasse zu definieren. Die Klassendatei muss dazu nicht unbedingt als Datei, sondern als Bytearray vorliegen. Somit ist das Laden neuer Klassen bereits ein elementarer Bestandteil der Programmiersprache Java. Möchte man Klassen zur Laufzeit generieren, muss man lediglich ein Bytearray mit korrektem Javabytecode füllen und über den ClassLoader als Klasse definieren. Die Bytecode-Manipulationsbibliotheken bieten zum Generieren von Klassen geeignete Methoden an.

Klassen zur Laufzeit zu generieren und einzubinden ist ein SMOC-Operator und sollte als solcher konkret definiert werden.

Kritik und Gefahren:

- Eine neue Klasse, die keine zur Kompilierzeit bekannte Schnittstelle implementiert, muss umständlich über Reflexion angesprochen werden, da der Compiler die Methodenzusicherung nicht prüfen kann.
- Lädt man ungültigen Bytecode, so führt dies zu Laufzeitfehlern.
- Ist ein Klassenname bereits vorhanden, kann er nicht erneut geladen werden. Dabei ist anzumerken, dass ein Klassenname sich aus Paket und Klassenbezeichnung zusammensetzt. Gleiche Namen in verschiedenen Paketen sind also möglich.
- Anstatt den Umweg über einen ClassLoader zu wählen, sollten konkrete Operatoren zum Klassenladen, Klassendefinieren und Klassengenerieren eingeführt werden. Ebenso

Operatoren, die prüfen, ob Klassen bereits geladen sind. Ein Operator zum Erzeugen von Klassen sollte etwa den gleichen Stellenwert haben wie der new-Operator zum Erzeugen neuer Objekte.

3.1.3 Klassen entfernen

In Java werden die Klassen der definierenden ClassLoader-Instanz in einer Art Namensraum zugeordnet. Wird die ClassLoader-Instanz vom GarbageCollector entfernt, so werden auch die entsprechenden Klassen entfernt. Die ClassLoader sind hierarchisch aufgebaut. Den obersten ClassLoader - auch BootstrapClassLoader genannt - kann man weder entfernen noch verändern. Das gezielte Löschen von Klassen ist somit nicht möglich, ohne die JVM zu verändern. Somit kann jSMOC keine Klassen löschen, obwohl dies ein nützlicher Operator wäre. Es gibt verschiedene Möglichkeiten, wie andere Klassen eine gelöschte Klasse kompensieren können. Man unterscheidet dabei den Umgang mit den verbleibenden Instanzen und den Umgang mit erbenden Klassen.

Alternativen für den Umgang mit erbenden Klassen

1. Löscht man eine Klasse, so werden alle erbenden Klassen ebenfalls gelöscht.
2. Löscht man eine Klasse, die noch erbende Klassen besitzt, führt dies zu einem Laufzeitfehler.
3. Löscht man eine Klasse, ändert sich die Oberklasse der erbenden Klassen auf den Obertyp der gelöschten Klasse.

Alternativen für den Umgang mit verbleibenden Instanzen

1. Löscht man eine Klasse, so werden alle Instanzen auf den nächsten verfügbaren Obertyp umgesetzt. Zugriffe auf nicht mehr vorhandene Methoden und Felder führen zu Laufzeitfehlern.
2. Löscht man eine Klasse, so werden die statischen Deklarationen dritter Instanzen auf den nächsten verfügbaren Obertyp gesetzt und mit null initialisiert.
3. Löscht man eine Klasse, werden alle Methodenaufrufe, Felder und ähnliches mit entfernt. Dies würde Automatismen erfordern, die ein nahezu menschliches Verständnis des Programms haben müssen, und ist somit vermutlich nicht umsetzbar.
4. Löscht man eine Klasse, die noch Instanzen besitzt, so erhält man einen Laufzeitfehler.

Zur Implementierung eines Operators, der Klassen löscht, sind alle Zweierkombinationen der beiden Listen denkbar. Die Löschung der Klassen Object und Thread sowie anderer elementarer Klassen sollte verhindert werden. Es würde sich ein Zugriffsmodifikator für Klassen anbieten, der das Löschen derselben verhindert. Mit diesem Zugriffsmodifikator könnten für das Programm essentielle Klassen geschützt werden.

Die Gefahren dieses Operators hängen von der Implementierung ab. Löscht man eine Klasse, so muss der Programmierer sichergestellt haben, dass diese nicht mehr benötigt wird. Geschieht eine Verwendung dennoch, so kann dies ähnlich einem Nullpointer-Fehler in einem Laufzeitfehler enden. Werden die Instanzen einer Klasse gelöscht, so kann dies auf die Fehler, die das Löschen von Objekten verursachen zurückgeführt werden. Dies wird im Kapitel 3.1.8 geklärt.

3.1.4 Klassen reflektieren

Klassen zu reflektieren bedeutet, eine Schnittstelle anzubieten, die eine Klassenintrospektion sowie eine Klassenmodifkation anbietet. Java ermöglicht über die java.lang.reflect.Class-Schnittstelle bereits eine vollständige Introspektion einer geladenen Klasse. Diese Schnittstelle sollte mit den Möglichkeiten der Modifikation erweitert werden. Ziel dieser SMOC-Operatoren muss der einfache Lese- und Schreibzugriff auf die Strukturen einer Klasse in der Form einer Metaklasse sein. Ob diese Metaklassen als Klassen oder als nicht reflektierbare neue Struktur eingezogen werden sollten, bleibt offen. Ein konkreter Nutzen der Reflexion der Metastruktur ist nicht zu erkennen. Java bietet die Introspektion der Metastruktur an, da diese selbst als Klassen umgesetzt ist.

Ein Operator zur Klassenreflexion birgt lediglich die Gefahren, die er durch die Methoden zur Modifikation der Reflexionsmetaklasse erbt (siehe Kapitel 3.2 - S. 27). Die introspektiven Anteile bergen die Gefahr der einfachen Extraktion von Passwörtern und Schlüsseln aus dem Arbeitsspeicher. Diese Gefahr besteht jedoch durch RAM-Editoren und Emulatoren auch ohne die introspektiven Anteile und ist somit anderweitig zu sichern.

3.1.5 Klassen tauschen

Eine Klasse zu tauschen bedeutet, sie zur Laufzeit durch z.B. eine neue Version zu ersetzen. Dies ist ein elementarer SMOC-Operator. Durch seine korrekte Implementierung könnten alle Klassenmodifikationen durchgeführt werden. Die Implementierung eines solchen Operators wird aus der Sicht des JVM-TI bereits vorbereitet (Vergleiche Kapitel 2.2 - S. 14). Auch wenn das Ersetzen bisher nur möglich ist, wenn sich die Signaturen der Klasse und ihrer Methoden nicht ändern, so ist der Schritt, einen Operator zum freien Austauschen zu stellen, bereits geschehen. Jedoch ist dieser Operator nicht elementar integriert, sondern soll über einen Zugriff auf die ausführende JVM über das JVM-TI geschehen. Dies ist aus SMOC-Sicht kritisch zu beäugen, da es nicht dem Zwecke der Selbstmodifikation dient, sondern dazu gedacht ist, Programme zur Laufzeit oder im Debugging von außen zu verändern. Es wäre erstrebenswert, auch für das Klassentauschen eine Bytecode-Instruktion zur Verfügung zu stellen.

Im Moment ist das freie Tauschen von Klassen noch nicht möglich. Da es jedoch ein essentieller Bestandteil von jSMOC ist, wird jSMOC eine vergleichbare Semantik implementieren, bei der Klassen durch Klassen mit neuem Namen ersetzt werden. Auf diesem Operator aufbauend werden alle Modifikationen der Klassen realisiert. Das Tauschen einer Klasse bedeutet, den Typ aller Objekte dieser Klasse zu verändern. Die Gefahren, die daraus resultieren können, sind im nächsten Unterkaptiel 3.1.6 beschrieben.

3.1.6 Objekte tauschen

In der Gilgul VM (vgl. Kapitel 2.5 - S. 19) wird der #=-Operator eingeführt, um Objekte effizient zu tauschen. Die Verwendung ist jedoch auf die Typsicherheit eingeschränkt und sollte erweitert werden, so dass Objekte völlig unabhängig getauscht werden können.

Das freie Tauschen von Objekten birgt folgende Gefahren:

1. Es kann zu NoSuchMethod-Laufzeitfehlern kommen, wenn ein Objekt durch eines ersetzt wird, dessen Klasse eine benötigte Methode nicht mehr besitzt.

2. Es kann zu NoSuchField-Laufzeitfehlern kommen, wenn ein Objekt durch eines ersetzt wird, dessen Klasse ein benötigtes Feld nicht mehr besitzt.

3. Es kann zu Nullpointer-Laufzeitfehlern kommen, wenn ein Objekt durch eines ersetzt wird, welches in einem Feld an unerwarteter Stelle null beinhaltet, oder ein Objekt durch null selbst ersetzt wird, um es zu löschen.

Das freie Tauschen von Objekten wird im später vorgestellten Ansatz benötigt, um das Verändern der Klasse eines Objekts zu ermöglichen. Da dieser Operator Modifikationen ermöglicht, ist dessen Implementierung umzusetzen. Zum Tauschen von Objekten wird in diesem Ansatz das Kopieren der Feldbelegung angestrebt (vgl. Kapitel 5.1 - S. 43).

3.1.7 Objekte hinzufügen

Dieser Operator ist bereits vorhanden. Das Erstellen neuer Objekte geschieht über den NEW-Operator, der zuerst ein leeres Objekt einer Klasse erstellt und anschließend die Konstruktormethode aufruft. Das Erstellen eines Leerobjekts, ohne den Konstruktor aufzurufen ist im Bytecode möglich, wird aber durch den Compiler in Java verhindert. Wie man im folgenden sieht, ist das Erzeugen und Aufrufen der Konstruktormethode im Bytecode getrennt:

```
1 NEW Address                //Erzeugen einer neuen Objektreferenz auf dem Stack
2 DUP                        //Duplizieren der Objektreferenz auf dem Stack
3 LDC "christian"            //"christian" auf den Stack legen
4 LDC "Heidenkopferdell"     //"Heidenkopferdell" auf den Stack legen
5 BIPUSH 45                  //den int 45 auf den Stack legen
6 INVOKESPECIAL Address.<init> (Ljava/lang/String;Ljava/lang/String;I)V
7 //Den Konstruktur Address(String, String, int) aufrufen.
```

Der Aufruf von DUP ist dabei notwendig, da der Konstruktor eine Referenz zum Objekt als this vom Stack nimmt. Wird die Referenz nach dem Erzeugen nicht mehr benötigt, wird vom Compiler auf DUP verzichtet.

3.1.8 Objekte entfernen

Ein Objekt zu entfernen bedeutet, es mit null zu tauschen, da dann der GarbageCollector den Speicher dieses Objekts wieder freigibt. Die einzige Gefahr dabei sind auftretende Nullpointer-Laufzeitfehler an Stellen, an denen das Objekt noch benötigt wird. Das gezielte Entfernen von Objekten kann erstrebenswert sein, wenn man sicherstellen möchte, dass Speicher freigegeben wird, oder wenn man Klassen entfernen möchte, die noch Instanzen haben. Der GarbageCollector wurde dagegen vielmehr dazu erfunden, dass der Speicher versehentlich nicht gelöschter, aber nicht mehr referenzierter Objekte freigegeben wird. Zusätzlich sollte es die Entwicklung von Programmen vereinfachen, wenn Referenzen dezentral gehalten und gelöscht werden, da in diesem Fall nicht mehr geprüft werden muss, wann der Speicher freigegeben werden kann. Die Einführung des GarbageCollectors hat dabei dazu geführt, dass das aktive Löschen von Objekten verloren gegangen ist, obwohl dies einen völlig anderen Charakter hat: So kann das aktive Löschen Fehler zur Testzeit aufdecken, die ansonsten wegen verbleibender unerwarteter Referenzen zu später auftretenden Fehlern führen. Diese später auftretenden Fehler erklären meist nicht die Ursache „hier war noch eine benötigte Referenz“, sondern erzeugen semantische Fehler, die zwar häufig keinen Absturz des Programms verursachen, aber zu unerwarteten Ergebnissen führen.

Ein Operator zur Löschung von Objekten sollte realisiert werden. Dabei ist die Existenz des GarbageCollectors aufrecht zu erhalten.

3.1.9 Objekte reflektieren

Objekte zu reflektieren bedeutet, eine Introspektionsschnittstelle für Objekte anzubieten, die es ermöglicht, die Wertebelegung sowie die zugehörige Klasse zu inspizieren. Zusätzlich muss eine Modifikationsschnittstelle angeboten werden, die das Verändern der Feldbelegung sowie das Ändern der zugehörigen Klasse erlaubt. Das Ändern der zugehörigen Klasse ist dabei mit dem Tausch eines Objekts dieser Klasse vergleichbar. Die Feldbelegung sollte übertragen werden (vgl. Kapitel 5.1 - S. 43).

Java bietet den versteckten Operator zur Introspektion der Klasse eines Objekts über die getClass()-Methode, die jedes Objekt besitzt. Über diese Klassenschnittselle kann dann auch die Feldbelegung inspiziert und manipuliert werden. Ein Operator, der das Setzen der Klasse ermöglicht, ist nicht vorhanden.

Eine konkretere Schnittstelle zur Reflexion von Objekten sollte umgesetzt werden. Objekte müssen unabhängig von der Klassereflexion inspiziert und modifiziert werden können.

3.1.10 Objekte ermitteln

Java selbst bietet keine Möglichkeit, die vorhandenen Instanzen (oder die einer Klasse) zu ermitteln. Über das JVM-TI ist das jedoch möglich. Da dieser Operator keine Gefahren birgt, jedoch einen sehr hohen Nutzen hat, ist es verwunderlich, dass Java keine Methode dafür anbietet. Besonders hilfreich, wäre das Ermitteln aller Instanzen einer Klasse. Sehr häufig aggregiert man in Java eine Liste der erzeugten Instanzen einer Klasse. Das Problem daran ist, dass der GarbageCollector dann immer eine Referenz zu einer Instanz findet und diese somit nicht frei gibt. Umständlich muss diese Freigabe über das Löschen aus dieser Liste erfolgen, was bei dezentralen Programmen schwierig ist. Da die JVM eine solche Liste hält, könnte man diese genau so gut anfragen.

In jSMOC kann ein solcher Operator jedoch nicht umgesetzt werden. In jSMOC werden diverse Instanzen gekapselt. Dieser Operator würde diese Kapselung zerstören oder für den Programmierer unerwartete Resultate verursachen. In einem anderen Ansatz sollte dieser Operator nach Möglichkeit umgesetzt werden. Da auch jSMOC die Liste aller Instanzen benötigt, wird dies über das JVM-TI im Hintergrund realisiert.

3.2 Kategorie Klasse

Java ermöglicht eine komplette Introspektion der Klasse zur Laufzeit. Diese wird in einer Metaklasse java.lang.reflect.Class angeboten. Diese Metaklasse bietet jedoch keine Möglichkeit der Modifikation. In diesem Kapitel sollen die möglichen Modifikationen und deren Gefahren ausgearbeitet werden.

3.2.1 Oberklasse aktualisieren

Wenn wir eine Klasse K modifizieren, so kann dies mit folgenden Operationen durchgeführt werden:

```
1 function modify(Class C, Modification m, boolean propagate){
2     Class N = Copy Class C
```

```
3     Apply Modification m N
4     forall Instanzes i of C
5         set type of i to N
6     if (propagate){
7         forall Subclasses c of C
8             modify(c, replace supertyp with N, true)
9     }
10 }
```

Die Entscheidung, Änderungen an die Unterklassen zu propagieren, ist eine Entscheidung, die auf die Stabilität zurückgeführt werden kann. Ändert sich eine Klasse, ohne die erbenden Klassen anzupassen, so sind diese unter der alten Klassenversion stabil. Propagiert man die Änderungen an die Unterklassen, so kann es zu Fehlern kommen, weil zum Beispiel neue Felder oder neue Methoden in der Unterklasse aus semantischen Gründen nicht eingeführt werden können.

Propagiert man Änderungen jedoch nicht, so erhält man ein für den Programmierer unlogisches Klassenkonstrukt, da plötzlich erbende Klassen Methoden der Oberklasse nicht besitzten, da die tatsächliche Oberklasse von der erwarteten abweicht. Man hat also eigentlich keine Änderung der Klasse erzeugt, sondern eine neue eingeführt, die völlig unabhängig von der Klassenhierarchie ist.

Ein denkbarer Mittelweg wäre es, dem Programmierer das Propagieren zu den Unterklassen zu überlassen. Das hätte dann jedoch vielmehr den Charakter, die Oberklasse zu aktualisieren. Dazu könnte ein eigener Operator updateSuperClass() eingeführt werden, der die Oberklasse auf die aktuelle Version setzt. Durch diese Variante kann der Programmierer flexibel entscheiden, welcher Weg für die aktuelle Änderung sinnvoll ist. Ebenso können Unterklassen vor der Anpassung der Oberklasse darauf vorbereitet werden. Dieser Weg wird in jSMOC gewählt, da er dem Programmierer die größtmögliche Freiheit gibt und man Erfahrungen zur Programmstabilität für alle beschriebenen Möglichkeiten generieren kann. Es gilt daher zu klären, welche Gefahren nun beim Aktualisieren der Oberklasse entstehen können:

1. Ein neues public oder protected Feld oder Methode ist durch die Änderung der Oberklasse bereits vorhanden.

2. Ein gelöschtes Feld oder Methode, das jedoch benötigt wird, ist nicht mehr vorhanden.

3. Die Änderung kann aus einem resultierenden Compilerfehler nicht durchgeführt werden. Wenn zum Beispiel eine Klasse durch eine Modifikation auf final gesetzt wird, führt das Aktualisieren der Unterklassen zu einem solchen Compilerfehler. Das Setzen des final-Zugriffsmodifikators bedeutet in Java, dass keine Klasse von dieser Klasse erben darf.

4. Man erzeugt eine Methode mit gleicher Signatur, die jedoch nicht überschrieben werden kann. Zum Beispiel kann eine int getValue()-Methode nicht mit einer Object getValue()-Methode überschrieben werden, obwohl sie die gleiche Signatur haben. In Java gehört der Rückgabetyp nicht zur Signatur.

3.2.2 Felder entfernen

Das Entfernen von Feldern ist eine der elementaren Veränderungen, die an einer Klasse durchgeführt werden können. Möchte man Felder löschen, die noch verwendet werden, so muss die

Verwendung zuerst entfernt werden. Geschieht dies nicht, erhält man Compilerfehler zur Laufzeit.

Auf folgende Gefahren sollte geachtet werden:

1. Das Löschen von pubilc- und protected-Feldern darf nur geschehen, wenn keine Unterklasse diese Feldnamen bereits deklariert. Sonst schlägt die Aktualisierung der Oberklasse in der Unterklasse fehl.

2. Das Löschen von Feldern darf nur geschehen, wenn das Feld auch in Unterklassen nicht mehr benutzt wird.

3. Das Löschen von Feldern sollte immer in der deklarierenden Klasse geschehen, da sie der Eigentümer des Felds ist.

3.2.3 Felder hinzufügen

Das Hinzufügen von Feldern ist eine elementare Veränderung, die an einer Klasse durchgeführt werden kann.

Das Aktualisieren der Oberklasse einer Unterklasse führt zu weiteren Problemen, da eingeführte Felder nicht überschrieben werden können. Besitzt also eine Unterklasse ein Feld mit gleichem Namen, so führt dies beim Aktualisieren der Oberklasse dieser Unterklasse zu einem Compilerfehler.

Das Hinzufügen von Feldern in Blätterklassen - Klassen die keine Unterklassen besitzen - ist stabil. Das Hinzufügen von private-Feldern ist ebenfalls unproblematisch, da diese in erbenden Klassen nicht sichtbar sind.

Man sollte also folgende Richtlinien beachten:

1. Hinzufügen von private-Feldern ist jederzeit möglich.

2. Hinzufügen von public-Feldern und protected-Feldern sollte nur geschehen, wenn sichergestellt ist, dass keine Unterklasse ein Feld mit gleichem Namen deklariert. Ansonsten sollten diese vor dem Propagieren entfernt oder umbenannt werden.

3. Hinzufügen von Feldern in Blätterklassen ist unproblematisch.

3.2.4 Felder reflektieren

Java erlaubt die Introspektion von Feldern. Es ist jedoch fraglich, ob eine Modifikation sinnvoll ist. Verändert man den Typ eines Felds, so ist dies gleichzusetzen mit dem Löschen des Felds und Hinzufügen eines Felds mit gleichem Namen und neuem Typ. Es ist daher kein eigener Operator notwendig. Gleiches gilt für den Namen und den Zugriffsmodifikator. Jedoch ist beim Verändern des Namens zu überlegen, ob alle Zugriffe auf dieses Feld bei der Änderung mitverändert werden sollten. Dies ist jedoch nicht so einfach möglich, da sich die Zugriffe auf mehrere Klassen verteilen können und umständlich gesucht werden müssten.

Wegen des geringen Nutzens und der möglichen Umsetzung durch das Löschen und Hinzufügen reicht es, die Introspektion eines Felds anzubieten.

3.2.5 Methoden hinzufügen

Eine Methode einer Klasse hinzuzufügen ist meistens unproblematisch. Bereits das JVM-TI erkennt, dass dies nur problematisch wird, wenn eine Methode eingefügt wird, die an späterer Stelle wegen ihrer Signatur nicht überschrieben werden kann, oder eine Oberklasse diese Methode bereits auf final gesetzt hat. Daher bietet das JVM-TI in seiner Schnittstelle bereits die Prüfung canAddMethod() an (vgl. Kapitel 2.2 - S. 14).

Die erste Problematik rührt daher, dass der Rückgabetyp einer Methode nicht Teil der Signatur ist. Deshalb kann eine Methode int getHausnummer() in einer Unterklasse nicht mit long getHausnummer() überschrieben werden.

In der Implementierung von jSMOC führen diese beiden Probleme zu einem Laufzeitfehler. Bei der Signaturproblematik tritt dieser jedoch erst beim Aktualisieren der Oberklasse über updateSuperclass() auf.

Neu hinzugefügte Methoden können nicht über den Punktoperator aufgerufen werden, da dieser nur compilersicher funktioniert.

```
1 Addresse a = new Addresse("Christian", "Heidenkopferdell", 45);
2 Addresse.addMethod(Addresse, "print() {...}");
3 a.print(); //---> Compilerfehler
4 //Reflexionsaufruf (Laufzeitfehler, falls print nicht existiert)
5 a.getClass().getDeclaredMethod("print").invoke();
```

Erstrebenswert wäre ein Operator, der den komplizierten Aufruf über Reflexion vereinfacht und nicht zu einem Compilerfehler führt. Dazu könnte man einen zusätzlichen Operator einführen, der statt eines Compilerfehlers zu einem Laufzeitfehler führt (hier der ?-Operator):

```
1 Addresse a = new Addresse("Christian", "Heidenkopferdell", 45);
2 Addresse.addMethod(Addresse, "print() {...}");
3 a?print(); //Laufzeitfehler, falls print() nicht existiert.
```

In jSMOC wird die JVM nicht verändert. Die Einführung eines solchen Operators ist daher nicht möglich. Der Reflexionsaufruf soll jedoch vereinfacht werden. Ein Operator, der eine Methode hinzufügt, sollte umgesetzt werden. Die Problematik mit der Signatur ist vom Programmierer zu beachten.

3.2.6 Methoden entfernen

Das Löschen einer Methode sollte ebenfalls als Operator umgesetzt werden. Fügt ein Programm nur Methoden hinzu, so kann ein Klassenkonstrukt mit der Zeit unübersichtlich werden. Das Entfernen einer Methode ist vergleichbar schwierig, wie das Entfernen eines public-Felds. An Stellen, an denen diese Methode nach dem Löschen noch aufgerufen wird, tritt eine NoSuchMethod-Laufzeitfehlermeldung auf.

Weitere Hinweise:

- Methoden, die über Schnittstellen oder abstrakte Oberklassen zugesichert werden, sollten nicht entfernt werden. Das Entfernen sollte zuerst in der abstrakten Klasse oder Schnittstelle geschehen, bevor diese in den umsetzenden Klassen gelöscht werden.

- Wird eine Methode gelöscht, ohne deren Verwendung zu beseitigen, führt das bei einem Aufruf zu Laufzeitfehlern.
- Beim Löschen der Methoden spielt es eine Rolle, ob dies in der Klassenhierarchie aufwärts oder abwärts geschieht. Abwärts ist die Methode direkt in dieser Klasse nicht mehr verfügbar. Aufwärts existiert sie noch durch die Vererbungsstrukturen.

3.2.7 Methoden reflektieren

Die Introspektion einer Methode ist in Java bereits möglich. Das Modifizieren einer Methode muss nicht angeboten werden. Das Ändern der Elemente einer Methode kann auf das Löschen und Hinzufügen zurückgeführt werden und erbt somit die Probleme dieser beiden Operatoren.

Die OpenJDK bietet das Austauschen von Methodenrümpfen zur Laufzeit über das JVM-TI (siehe Kapitel 2.2 - S. 14).

Hinweise und Gefahren:

- Methodennamen zu ändern ist nur sinnvoll, wenn deren Verwendung über das ganze Programm mit verändert wird.
- Methodenparametertypänderungen können nur sinnvoll durchgeführt werden, wenn die Methodenrümpfe mit geändert werden.
- Methodenparameternamensänderungen können nur sinnvoll durchgeführt werden, wenn die Methodenrümpfe mit geändert werden.
- Verändern des Rückgabetyps kann ebenfalls nur sinnvoll durchgeführt werden, wenn der Methodenrumpf angepasst wird. Ferner müssen alle Methodenaufrufe im kompletten Programm mit modifiziert werden.
- Der Methodenrumpf kann problemlos ersetzt werden.

Es ergibt sich also, dass die Änderung an den Elementen einer Methode (mit Ausnahme des Rumpfs) alleine nicht sinnvoll ist. Daher bietet jSMOC keine Veränderung der Elemente einer Methode. Eine Methode muss komplett ersetzt werden. Dies geschieht über Löschen und erneutes Hinzufügen und besitzt keinen eigenen Operator.

3.2.8 Innere Klassen

Innere Klassen hinzufügen birgt keine Gefahren, sofern deren Namen nicht belegt sind. Durch den Ansatz von jSMOC, bei dem Klassen durch neue Klassenversionen ersetzen werden, wird jedoch vom Hinzufügen innerer Klassen abgesehen. Die Problematik wurde bereits in der Einleitung erörtert (vgl. Kapitel 1 - S. 8).

Das Entfernen innerer Klassen und die daraus entstehenden Probleme sowie die Reflexion dieser ist vergleichbar mit der Reflexion und den Problemen einer normalen Klasse und muss hier nicht weiter ausgeführt werden.

3.2.9 Statische Codeblöcke

Statische Codeblöcke werden beim Laden der Klasse ausgeführt. Eine Änderung zur Laufzeit einer bereits geladenen Klasse ist wegen der fehlenden Auswirkung nicht sinnvoll.

Im jSMOC-Ansatz werden Klassen kopiert. Daher wird davon abgeraten, statische Codeblöcke in modifizierbaren Klassen zu verwenden. Tut man dies dennoch, so muss man sich im Klaren darüber sein, dass diese bei jeder Klassenänderung erneut ausgeführt werden. jSMOC bietet keine direkten Modifikationsoperatoren für statische Codeblöcke. Jedoch kann über die allgemeine Methode modify(Modification m) eine Modifikation der statischen Codeblöcke erfolgen. Würde man die Veränderungen einer Klasse über die VM realisieren, so sind statische Codeblöcke für Veränderungen nicht mehr relevant, da sie nicht erneut ausgeführt werden.

3.3 Kategorie Objekte

Objekte spielen in der objektorientierten Sprache eine elementare Rolle. Jedoch werden sie in Java in der Reflexion vernachlässigt. So kann lediglich die zugehörige Klasse zur Laufzeit ermittelt werden. Umständlich über diese kann man dann prüfen, ob gewisse Methoden oder Felder vorhanden sind.

Das Ändern der Feldbelegung in Objekten ist der Kernaspekt der objektorientierten Programmierung. Will man Sprachen um SMOC-Elemente erweitern, so muss man verstehen, dass dieser Kernaspekt zwar noch elementar ist, jedoch nicht mehr die zentrale Rolle einnimmt. Das Verändern der Feldbelegung wird durch den Compiler geschützt und deren Existenz zur Compilerzeit zugesichert. Verändern sich die Felder und Methoden einer Klasse, so können viele dieser geschützten Zugriffe nicht mehr vom Compiler zugesichert werden. Der Zugriff auf Objekte muss daher erweitert werden.

Um mit Objekten zu interagieren, gibt es im Wesentlichen den Punktoperator. Dieser Operator ermöglicht den Zugriff auf die Feldbelegung und Methoden der Klasse eines Objekts. Genau diese beiden Zugriffe werden zur Compilerzeit geprüft. Es wird sichergestellt, dass die Felder und Methoden existieren. In einer SMOC-erweiterten Programmiersprache, kann dies nicht mehr garantiert werden. Ein Feld oder eine Methode die zur Compilerzeit existierte, kann zum Zeitpunkt des Zugriffs bereits entfernt sein. Des Weiteren möchte man Methoden aufrufen, die zur Compilerzeit noch nicht existieren.

Da die Prüfung des Compilers sinnvoll ist, und auch das Schreiben von Programmen in Entwicklungsumgebungen durch Autovervollständigung vereinfacht, sollte die Compilerpflicht des Punktoperators nicht verändert werden. Er sollte zusichern, dass alle Aufrufe über ihn zur Compilerzeit geprüft sind. Vielmehr ist der versteckte Operator, den die Introspektion anbietet, zu konkretisieren. Die Semantik eines solchen Operators würde in Java nicht abweichen, lediglich der Compiler müsste auf die Prüfung verzichten. Es geht also darum einen Operator einzuführen, der die gleiche Semantik wie der Punktoperator besitzt, jedoch ohne dessen Compilerzusicherung.

In Abschnitt 3.2.5 über das Hinzufügen von Methoden wurde ein solcher Operator bereits als ?-Operator kurz vorgestellt:

```
1 Addresse a = new Addresse("Christian", "Heidenkopferdell", 45);
2
3 a?print(); //Laufzeitfehler, falls print() nicht existiert.
4 a?hausnummer = 34;  //Laufzeitfehler, wenn hausnummer nicht existiert
5
6 a.print(); //Compilerfehler, falls print() nicht existiert.
```

```
7 //Laufzeitfehler, falls print() zur Laufzeit geloescht ist
8 a.hausnummer = 35; //Compilerfehler falls hausnummer nicht existiert
9 //Laufzeitfehler, falls hausnummer geloescht ist.
```

Eine Einführung eines solchen Operators führt lediglich zu Problemen, wenn die Programmierer unvorsichtig damit umgehen. In Skriptsprachen entspricht der Punktoperator bereits dem ?-Operator, trotzdem werden dort nicht unbedingt mehr Fehler gemacht.

Beispiel (JavaScript):

```
1 var otto = new Object();
2 otto.hallo();  //Laufzeitfehler: Methode hallo existiert nicht
3 otto.hallo = function (){ alert("hallo"); };
4 otto.hallo();  //--> Dialog wird angezeigt
5 otto.hallo = null;
6 otto.hallo();  //Laufzeitfehler: Methode hallo existiert nicht
```

In Java besitzen alle Objekte eine introspektive getClass()-Methode. Der Modifikationsoperator setClass(class) ist jedoch nicht existent. Das Setzen einer Klasse und die damit auftretenden Probleme sind bereits auf Programmebene geklärt worden. Ein Operator zum Setzen der Objektklasse sollte - wie in der Programmebene bereits begründet - umgesetzt werden.

3.4 Kategorie Methode

Mögliche Operatoren, um auf Methoden zuzugreifen, sind:

- Zugriffsmodifikator verändern
- Rückgabetyp verändern
- Name verändern
- Fehlerliste verändern
- Parameter reflektieren
- Parameter hinzufügen
- Parameter entfernen
- Codeblock reflektieren
- Codeblock verändern

Im Wesentlichen verbergen sich hier bereits geklärte Operatoren und deren Gefahren. Diese Operatoren sind auf das Löschen und Hinzufügen einer Methode zurückzuführen. Die Elemente einer Methode, bis auf den Codeblock, können nicht sinnvoll alleine modifiziert werden. Daher werden in jSMOC keine Operatoren zur Modifikation von Methodenelementen angeboten. Dies wurde bereits im Kapitel 3.2.7 angedeutet.

3.5 Kategorie Feld

Mögliche Operatoren, um auf Felder zuzugreifen, sind:

1. Zugriffsmodifikator verändern
2. Typ verändern
3. Name verändern

Im Wesentlichen verbergen sich hier bereits geklärte Operatoren und deren Gefahren. Diese Operatoren sind auf das Löschen und Hinzufügen eines Felds zurückzuführen.

3.6 Kategorie Parameter

Mögliche Operatoren, um auf Parameter einer Methode zuzugreifen, sind:

1. Name verändern
2. Typ verändern

Im Wesentlichen verbergen sich hier bereits geklärte Operatoren und deren Gefahren. Diese Operatoren sind auf das Löschen und Hinzufügen einer Methode samt Parameter zurückzuführen. Den Parameter zu verändern ist ein Spezialfall der Operatoren in Kapitel 3.4.

3.7 Kategorie Codeblöcke

Codeblöcke können durch das JVM-TI getauscht werden. Sie bestehen aus einer linearen, assemblerartigen Operationsliste und können in Java bereits durch die Bytecode-Manipulationsbibliotheken ausreichend inspiziert und geändert werden.

Die jSMOC-Bibliothek sieht davon ab, Codeblöcke gesondert zu betrachten. Sie können ebenfalls praktisch durch das Löschen und erneute Hinzufügen von Methoden geändert werden.

3.8 Operatoren als Methoden

Java verbirgt viele introspektive Operatoren hinter *nativen* Methoden. Unter *nativen* Methoden versteht man Methoden, deren Implementierung in der VM zu finden ist. Java kann auf diese Methoden über die Bytecode-Instruktion invokenative zugreifen. Üblicherweise sind solche *nativen* Methoden dafür gedacht die Geschwindigkeit von C/C++ in elementaren Funktionen, wie das Kopieren eines Arrays oder Stringoperationen, an Java weiter zu geben. Ebenso wird auf diese Weise eine Hardwareschnittstelle angeboten. Mit speziellen Programmen können Treiber in die VM geladen werden und *native* Methoden aus C++-Programmen extrahiert werden. Danach können diese in Java eingebunden und verwendet werden[26].

Man stellt fest, dass es in Java eine unsaubere Trennung zwischen Operatoren und *nativ* eingefügten Methoden gibt. Native Methoden wurden häufig verwendet, um Operatoren umzusetzen, anstatt diese in Bytecode-Instruktionen zu verlagern. Beispiele hierfür sind:

- getClass()

[26] Beispiel ist die libusb (`http://www.libusb.org/`) für den Zugriff auf USB-Geräte aus Java heraus

- intern() [in Strings]
- class
- sowie alle Methoden und Funktionen des java.lang.reflect-Pakets.

Dieses unsaubere Verhalten wird für jSMOC übernommen. Operatoren werden als Methoden eingeführt. Der ?-Operator wird als vereinfachter Introspektionsaufruf implementiert. Ebenso der setClass()-Operator. Der Hauptgrund hierfür wurde bereits in der Einleitung erwähnt: Eine Bachelorarbeit ermöglicht es zeitlich nicht, einen Compiler und eine neue VM zu entwickeln. Ebenso ist dies für die ersten Schritte mit selbstmodifizierender Software nicht notwendig und würde eine weitere Sprache auf den Markt bringen, die erst die Anerkennung der Programmierer finden müsste. Erweist sich SMOC als sinnvoll, so sollte dieser Weg, eine VM zu schreiben, in Betracht gezogen werden und mit ihm eine sauber definierte Trennung von Operator und Methode vollzogen werden. In dieser Arbeit soll es daher ausreichen, dass die unsaubere Trennung zwischen Methoden und Operatoren festgestellt und kritisch festgehalten wurde sowie dass die Operatoren auch in der jSMOC-Bibliothek als Methoden umgesetzt sind.

3.9 Konkrete Liste der jSMOC-Operatoren

Als Ergebnis dieser Analyse ergibt sich die folgende Liste, der in jSMOC umzusetzenden konkreten Operatoren zur Modifikation von Javaprogrammen, eingeteilt in Programmreflexion, Klassenreflexion und Objektreflexion:

- **Programmreflexion**:
  - SMOCReflectProgram.getClasses() liefert die Reflexionsschnittstellen aller geladenen Klassen zurück.
  - SMOCReflectProgram.newClass(String name) erzeugt eine neue modifizierbare Klasse.
  - SMOCReflectProgram.reflectObject(SMOCObject o) liefert eine Schnittstelle, um Objekte zu reflektieren.
  - SMOCReflectProgram.deleteObject(SMOCObject o) löscht ein Objekt und setzt alle Referenzen auf null.
  - SMOCReflectProgram.assignObjectReference(SMOCObject object, SMOCObject newObject) entspricht dem #=-Operator der Gilgul VM und setzt alle Referenzen zum Objekt object auf newObject um.
  - SMOCReflectProgram.copyFields(Object from, Object to) kopiert alle Feldbelegungen vom Objekt from zum Objekt to, sofern ein Feld mit gleichem Namen in to existiert und der Typ kompatibel ist.
- **Klassenreflexion**:
  - SMOCReflectProgram.reflectClass(class) liefert eine Reflexionsschnittstelle für Klassen.
  - SMOCReflectClass.addMethod(String src) fügt eine Methode mit dem Quellcode src zu der Klasse hinzu.

- SMOCReflectClass.delMethod(String name, String desc) löscht die Methode der Klasse mit dem Namen name und der Signaturbeschreibung desc.
- SMOCReflectClass.addField(String src) fügt ein Feld zu einer Klasse hinzu.
- SMOCReflectClass.delField(String name) löscht das Feld mit Namen name.
- SMOCReflectClass.setSuperClass(class) ändert die Superklasse der Klasse.
- SMOCReflectClass.addInterface(i) fügt die Schnittstelle i hinzu.
- SMOCReflectClass.remInterface(i) entfernt die Schnittstelle i
- SMOCReflectClass.modify(Modification m) führt eine beliebige Modifikation durch, die mit Javassist als Bytecode-Manipulation angegeben wird. Dabei können alle Modifikationen durchgeführt werden außer eine Namensänderung der Klasse. Diese letztere Operation würde ignoriert werden.
- SMOCReflectClass.create(parameter...) erzeugt eine neue Instanz der aktuellen Klasse.
- SMOCReflectClass.invoke(that, method, parameters...) Dabei wird die Methode method bestehend aus Name und Signaturbeschreibung auf der Instanz that mit den Parametern parameters aufgerufen.
- SMOCReflectClass.getValue(instance, name) liefert die Feldbelegung für das Feld name der Instanz instance zurück.
- SMOCReflectClass.setValue(instance, name, value) setzt die Feldbelegung für das Feld name der Instanz instance auf den Wert value.
- SMOCReflectClass.getCurrentBytecode() liefert den aktuellen Bytecode der Klasse zurück.
- SMOCReflectClass.getCurrentJavaClass() liefert die Javareflektionsschnittstelle der aktuellen Klasse zurück.
- SMOCReflectClass.getVersion() gibt die aktuelle Klassenversion zurück. Dabei ist 0 die kompilierte Klasse.
- SMOCReflectClass.getSuperClass() liefert die aktuelle Superklasse zurück.
- SMOCReflectClass.getInterfaces() liefert die Menge der implementierten Reflexionsschnittstellen zurück.
- SMOCReflectClass.getFields() liefert die Menge der Felderreflexionsschnittstellen zurück.
- SMOCReflectClass.getMethods(boolean declaredOnly) liefert die Menge der Methodenreflexionsschnittstellen mit, die je nach declaredOnly nur in dieser Klasse deklariert wurden oder einschließlich aller Methoden aller Oberklassen.
- SMOCReflectClass.is_a(class) liefert true, wenn class eine Oberklasse ist.
- SMOCReflectClass.existsMethod(name, signature) liefert true, wenn eine solche Methode existiert.
- SMOCReflectClass.existsMethod(name) liefert true, wenn eine Methode mit Namen name existiert. Dabei wird nur der Namen, nicht die Signatur berücksichtigt.

  - SMOCReflectClass.existsField(name) liefert true, wenn ein Feld mit Namen name existiert.
  - canBeModified() liefert true, wenn diese Klasse SMOCObject implementiert und somit modifizierbar ist. SMOCObject kann nicht zur Laufzeit hinzugefügt werden!

- **Objektreflexion**:
  - SMOCReflectObject.getObjectClass() liefert die SMOCReflectClass Schnittstelle der Objektklasse.
  - SMOCReflectObject.setObjectClass(class) setzt die Klasse eines Objekts.
  - SMOCReflectObject.getValue(field) liefert den Wert für das Feld field.
  - SMOCReflectObject.setValue(field, value) setzt den Wert für das Feld field auf value.
  - SMOCReflectObject.invoke(methodName, signature, parameter) ruft die definierte Methode auf diesem Objekt auf.

4 Verworfene Ansätze

Im Folgenden werden die verworfenen Ansätze kurz beschrieben und Probleme sowie Erkenntnisse daraus extrahiert. Ebenso wird versucht, die Auswirkungen auf den dann tatsächlich durchgeführten Ansatz vorzubereiten. Es ist nicht zu erwarten, dass die Ausführungen der Ansätze vollständig und detailiert sind. Es soll lediglich die Entwicklungsgeschichte des umgesetzen Ansatzes skizziert und auf die Probleme hingewiesen werden, damit die gewonnen Erkenntnisse nicht verloren gehen. Für das Verständnis von jSMOC ist dieses Kapitel nicht notwendig. Die Ansätze sind nach ihrer zeitlichen Entstehung angeordnet.

4.1 Klassengenerieren hinter einer Schnittstelle

Der erste Schritt, selbstmodifizierende Software in Java zu schreiben, ging über eine Schnittstelle. Die dahinter liegende Idee war es, dass der Programmierer eine solche Schnittstelle mit den gewünschten Methoden bereit stellt und zur Laufzeit Klassen erzeugt, die diese Schnittstelle implementieren.

Das Erzeugen solcher Klassen sollte über Javassist geschehen. Das Austauschen von Methoden war auf diese Weise einfach realisierbar. Das Hinzufügen von Methoden konnte ebenfalls realisiert werden, jedoch musste der Zugriff auf diese umständlich über die Java Introspektion erfolgen. Dieser Ansatz wurde verworfen, da damit das Hinzufügen neuer öffentlicher Felder sowie das Löschen der durch die Schnittstelle zugesicherten Methoden nicht umsetzbar war. Das Löschen konnte lediglich über das Werfen eines Laufzeitfehlers im Methodenrumpf dieser Methode realisiert werden. Methodensignaturen waren festgelegt und konnten nicht verändert werden.

Aus diesem Ansatz wurden folgende Erkenntnisse gewonnen:

- Die Reduzierung auf eine gemeinsame Schnittstelle SMOCObject kann übernommen werden, jedoch ohne Methodenzusicherung.
- Das Erzeugen neuer Klassen mit Javassist wird auch in jSMOC immer noch verfolgt.
- Methodenaufrufe in Java über Introspektion sind unschön und müssen vereinfacht werden.
- Mit jeder Änderung wird eine neue Klasse erzeugt. Dies beeinträchtigt die Speicherauslastung sowie die Geschwindigkeit der JVM. Eine genaue Anzahl möglicher Modifikationen, bis der Speicher überläuft, kann nicht bestimmt werden. Sie ist durch den verfügbaren Hauptspeicher festgelegt.

Beispiel:

```
1 public static void main(final String[] args) throws Exception {
2 //load std. class pool
3 final ClassPool classPool = ClassPool.getDefault();
4
5 //modify the class
6 final CtClass ballClass = classPool.get("beispiel1.Ball");
7 final CtMethod throwIt = ballClass.getDeclaredMethod("throwIt");
8 throwIt.setBody("System.out.println(\"ball2 thrown\");");
9
```

```
10 //load the class
11 final Class<?> modifiedClass = ballClass.toClass(); // exception
12
13 //create instance
14 final Ball ball = (Ball)modifiedClass.newInstance();
15 ball.throwIt();      //-------> ball2 thrown
16 }
```

Im obigen Beispielprogramm ist Ball die gemeinsame Schnittstelle. Sie bietet die Methode throwIt() an, die mit Hilfe von Javassist manipuliert wird, indem eine neue Klasse Ball implementiert und throwIt() anders definiert wird.

4.2 Umsetzung mit einer Metaklassenebene

Bei der Idee der Umsetzung über eine Metaklassenebene wurde das Erstellen von Klassen ähnlich wie in Skriptsprachen komplett auf die Laufzeit verlegt. Unter dieser Metaebene wurden verschiedene Klassenversionen vereint und eine zeitliche Entwicklung der Klassen festgehalten.

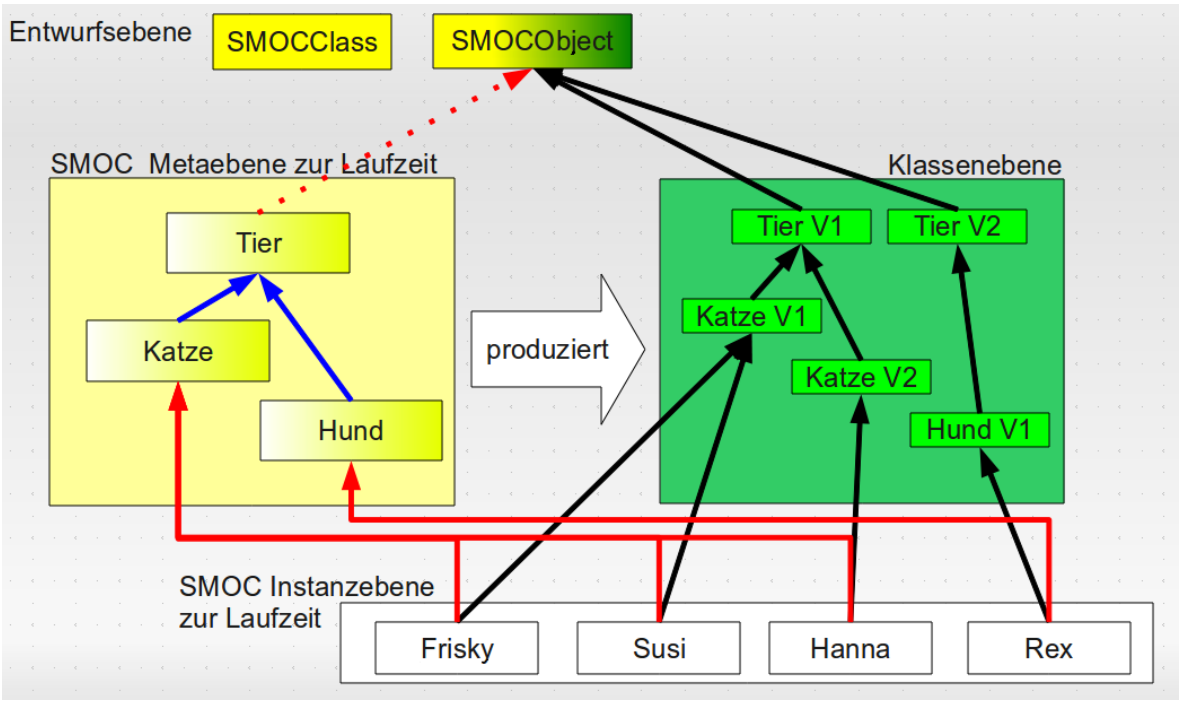


Abbildung 5: Illustration der Metaklassenidee

Wird eine Metaklasse verändert, so werden neue Klassenversionen erzeugt. Erst nach diesem Zeitpunkt erzeugte neue Instanzen werden dann unter dieser neuen Klassenversion generiert. Die alten Instanzen bleiben der alten Klassenversion zugeordnet. Metaklassen werden komplett zur Laufzeit über Methoden generiert. Die realen Klassen wurden mit Hilfe von Javassist kompiliert und geladen. Dieser Ansatz wurde frühzeitig verworfen, da der Code durch diese Technik extrem unübersichtlich und kompliziert wird. Die Hilfen von IDE und Compiler gehen

verloren. Java würde auf eine Skriptsprache mit unübersichtlicher Syntax reduziert. Trotzdem konnten so Modifikationen umgesetzt werden.

Das folgende Codebeispiel muss nicht verstanden werden und soll lediglich exemplarisch zeigen, warum von diesem Ansatz Abstand genommen wurde:

```
1  public static void main(final String[] args) throws Exception {
2  final SMOCClass Animal = new SMOCClass("Animal");
3  Animal.addField("protected final String name;");
4  Animal.addConstructor("public Animal (String name) { "//
5  + "     this.name = name; " //
6  + "}");//
7  Animal.addMethod("public void walk(String place) { "//
8  + "     System.out.println(name+\" walks to a \"+place); " //
9  + "} ");//
10
11 final SMOCClass Cat = new SMOCClass("Cat", Animal);
12 Cat.addConstructor("public Cat (String name) { super(name); }");
13 Katze.addMethod("public void miow() {  " //
14 + " walk(\"sunny place\");  " //
15 + " System.out.println(name+\": miow\");" //
16 + "} ");//
17
18 final SMOCObject frisky = Cat.newInstance("frisky");
19
20 Cat.removeMethod("miow");
21 Cat.addMethod("public void miow() {  " //
22 + " walk(\"sunny place\");  " //
23 + " System.out.println(name+\": miow version 2\");" //
24 + "} ");//
25
26 final SMOCObject susi = Cat.newInstance("susi");
27
28 Animal.removeMethod("walk");
29 Animal
30 .addMethod("public void walk(String place) {
31 System.out.println(name+\" runs to a\"+place); }");
32
33
34 final SMOCClass Dog = new SMOCClass("Dog", Animal);
35 Dog.addConstructor("public Dog(String name) {super(name); }");
36 Dog.addMethod("public void bark() { " + "   walk(\" fence\"); "
37 + " System.out.println(name+\" wooff!\"); " + "}");
38
39 final SMOCObject rex = Hund.newInstance("rex");
40 rex.invoke("bark");
41 rex.invoke("walk", "somehwere");
42 frisky.invoke("miow");
43 frisky.invoke("walk", "somehwere");
44 susi.invoke("miow");
45 susi.invoke("walk", "somehwere");
46 }
```

Folgende Erkenntnisse wurden aus diesem Ansatz gewonnen und übernommen:

- SMOC-Programme müssen übersichtlich bleiben.

- Das Generieren von Klassen zur Laufzeit wird in jSMOC ähnlich gestaltet.
- SMOC-Programme müssen Compilerunterstützung als Programmierhilfe behalten.
- jSMOC implementiert die Klassenmanipulation über Klassenversionen im Hintergrund. Die Metaebene wird also versteckt erzeugt, alte Instanzen werden neuen Klassen zugeordnet. Die Hierarchie bleibt übersichtlich.

4.3 Hotswapping

Hotswapping ist eine Technik, die ursprünglich über das JVM-TI (vergleiche Kapitel 2.2 - S. 14) ermöglicht wird. Javassist bietet hierfür eine einfache Schnittstelle an. Durch Hotswapping können Methodenrümpfe zur Laufzeit ersetzt werden. Das Hinzufügen von Methoden oder das komplette Verändern von Klassen ist darüber noch nicht möglich, da bisher keine JVM diese Fähigkeit implementiert.

Hotswapping wurde verworfen, da damit zu wenige der gewünschten SMOC-Operatoren umsetzbar gewesen wären. Das Austauschen von Methodenrümpfen ist von der Javassist HotSwapper-Implementierung ausreichend bearbeitet. Man kann also mit Javassist Methodenrümpfe einer laufenden Anwendung bereits austauschen.

Dieser Hotswapping-Ansatz hat jedoch dazu geführt, sich mit dem JVM-TI weiter auseinanderzusetzen und ist somit ein entscheidender Schritt zur Erkenntnis, dass SMOC-Programmierung zwar gewünscht, aber unzureichend verstanden und ausgearbeitet ist.

4.4 Mehrfachvererbung von SMOCObject

Wünschenswert jedoch nicht umsetzbar wäre ein Ansatz über Mehrfachvererbung gewesen, bei dem alle SMOCObjects eine gewisse Menge an Funktionen bereits vorimplementiert haben. Dies wäre in Java jedoch nur über eine abstrakte Klasse realisierbar. Diese können nur an oberster Stelle einer Klassenhierarchie eingebunden werden und würden somit das Modifizieren in Rahmenwerken unmöglich machen. Wünschenswerte Methoden für diese abstrakte Klasse wären:

- getClass(), jedoch unter anderem Namen, da diese Methode nicht überschrieben werden kann.
- setClass(class), zum Ändern der Objektklasse.
- invoke(methode, param1...), zum unsicheren Aufruf von Methoden.
- getValue(field), zum lesenden Zugriff auf Felder.
- setValue(field, value), zum schreibenden Zugriff auf Felder.

Da Mehrfachvererbung jedoch nicht möglich ist, müssen diese Methoden nun als statische Methoden eingebunden werden. Dies hat lediglich Auswirkungen auf die Intuition der Operatoren. Sofern diese wohlverstanden ist, würde man sie eigentlich dem Objekt zuordnen wollen. Da dies nicht geht, müssen sie an einer anderen Stelle statisch umgesetzt werden.

Die Idee, diese Methoden über die SMOCObject-Schnittstelle zuzusichern und über Codetransformation einzufügen, ist abzulehnen, da dies für den Programmierer unverständlich wäre. Es würde den Charakter einer Schnittstelle verkennen und beim fachkundigen Programmierer auf Unverständnis stoßen, da es nicht üblich ist, eine Schnittstelle zu implementieren

jedoch die Funktionen leer zu lassen, damit diese später durch eine Codetransformation gefüllt werden.

4.5 Implementierung durch eine VM

Nach einer kurzen Analyse des Quellcodes der OpenJDK und Kaffe VM konnte festgestellt werden, dass eine Manipulation dieser im Zeitumfang einer Bachelorarbeit nicht möglich ist. Wünschenswert wäre dies trotzdem aus folgenden Gründen:

- Saubere Implementierung der Operatoren als Bytecodeinstruktionen
- Saubere Implementierung ohne Klassenkopien durch direkte Manipulation der Klassenstrukturen
- Saubere Spezifikation und Konkretisierung der Operatoren, die mit einer solchen Implementierung einhergehen
- Kein Überlaufen des Speichers durch zu viele Modifkationen wegen zu vieler erzeugter Klassen
- Das Löschen von Klassen könnte umgesetzt werden.

5 Konzeption und Realisierung von jSMOC

Um die Modifikationen in Java zu ermöglichen, wurde der Weg über eine Bibliothek gewählt. Diese soll die in Java vorhandene Reflexions-API um die Fähigkeiten der Modifikationen erweitern. Die umgesetzten Methoden können der Liste auf Seite 35 entnommen werden.

Die Kernmethoden, auf die alle Modifikationsoperatoren schlussendlich zurückgeführt werden, ist die Fähigkeit Objektreferenzen zu verändern, Feldbelegungen zu kopieren und alle Instanzen einer Klasse aufzufinden. Es müssen also diese Methoden zuerst genauer betrachtet werden. Anschließend wird erklärt, wie diese Methoden genutzt werden können, um Modifikationen durchzuführen.

5.1 Übertragung der Feldbelegung

```
1 function copyFields(from, to){
2     FORALL Fieldnames f OF from {
3         if (exists(to.f) && type(to.f) >= type(from.f)){
4             to.f = from.f
5         }
6     }
7 }
```

Listing 9: Pseudocode zur Übertragung der Feldbelegung

Der obige Algorithmus kopiert die Feldbelegung eines beliebigen Objekts in ein anderes Objekt. Zur Auswahl wird dazu der Feldname verwendet. Die Übertragung findet statt, sofern im Zielobjekt ein Feld mit gleichem Namen und kompatiblem Typ existiert. Kompatibel bedeutet dabei, dass die Zuweisung to.f = from.f keinen Typfehler wirft. So kann ein int einem long zugewiesen werden, aber nicht umgekehrt. Ebenso kann eine Zuweisung zu einer Oberklasse erfolgen, jedoch nicht zu einer Unterklasse.

5.2 Zuweisung einer Objektreferenz

```
1 function assignObjectReference(from, to){
2     FORALL Referenz r to from {
3         r = Reference of to
4     }
5 }
```

Listing 10: Theoretischer Pseudocode zur Zuweisung einer Objektreferenz

Theoretisch müssen alle Referenzen des Objekts from gesucht werden und durch die Referenz zum Objekt to ersetzt werden. Praktisch weist dies zwei Probleme auf:

1. Alle Referenzen zu einem Objekt zu finden, erfordert es, alle Objekte, Felder und lokalen Variablen sowie den kompletten Stack jedes Threads zu inspizieren.

2. Die Zuweisung einer Referenz darf nur typsicher erfolgen. Das bedeutet, die Klasse von to muss eine Unterklasse oder identisch zu der von from sein.

Die Umsetzung des ersten Punkts ist in Java über das JVM-TI praktisch möglich. Man erhält jedoch Schwierigkeiten, da alle Threads, die analysiert werden sollen, angehalten werden müssen. Ebenso ist das vollständige Analysieren eines Programms langsam. Damit die zu

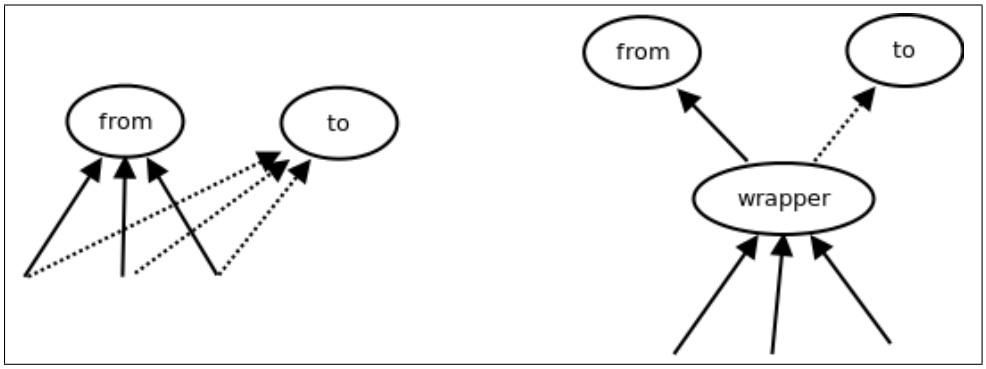


Abbildung 6: assignObjectReference(from, to) mit und ohne Wrapper

analysierenden Threads angehalten werden können, muss man diese Analyse in einen anderen Thread auslagern. Eine schnellere Alternative wäre es, alle Referenzen in einen Wrapper zu packen. Ähnlich wie in der Implementierung der Gilgul VM (siehe Kapitel 2.5 - S. 19) erhält man so eine Indirektion, die man effizient verändern kann. Dies ist auch in Java über das JVM-TI realisierbar. Der Algorithmus zum Ersetzen einer Instanz könnte dann wie folgt aussehen:

```
1 function assignInstance(Wrapper from, Object to){
2     from.reference = to;
3 }
4 function assignInstance(Wrapper from, Wrapper to){
5     from.reference = to.reference;
6 }
```

In jSMOC wird die Variante mit Wrapper verwendet, da sie zum einen effizienter ist und zum anderen keine Auslagerung in einen anderen Thread benötigt. Dies vereinfacht ebenfalls den Algorithmus, der später benötigt wird, um alle Instanzen einer Klasse auszutauschen.

Das zweite Problem vereinfacht sich mit Wrappern ebenfalls. So kann in einem Wrapper als Referenzhalter der allgemeinste Typ Object verwendet werden. Methodenaufrufe und Feldzugriffe die nun auf dem Wrapper erfolgen, müssen durch eine Codetransformation die Indirektion aufheben. Die Wrapperklasse in diesem Ansatz heißt SMOCInstanceWrapper. Es werden dabei nur die Instanzen modifizierbarer Klassen in solche Wrapper gekapselt.

5.3 Instanzen ersetzen

Möchte man alle Instanzen einer Klasse ersetzen, steht man vor dem Problem, alle Instanzen einer Klasse zu finden. Java bietet hierfür keine Möglichkeit. Speichert man alle erzeugten Instanzen zwischen, so funktioniert der GarbageCollector nicht mehr, da immer eine Referenz in dieser Instanzenliste existiert. Das JVM-TI ermöglicht es, alle Instanzen einer Klasse zu ermitteln. Jedoch erhält man diese als ObjectReferences und nicht als Instanzen der Klasse Object. Da das JVM-TI nicht die Fähigkeit bietet, die Feldbelegung von als final deklarierten Feldern zu ändern, wie es zum Kopieren der Feldbelegung notwendig wäre, muss man eine Mischform schaffen. Ein möglicher Trick, der in dieser Arbeit verwendet wird, ist es mit dem JVM-TI die ObjectReference in ein vorher festgesetztes Feld eines festgelegten Objekts zu transferieren und somit auf das Java Object zugreifen zu können. Der Algorithmus,

der dies durchführt, ist in SMOCUtils.replaceInstances(oldClass, newClass) zu finden. Er führt folgende Schritte zur Ersetzung aller Instanzen der Klasse oldClass zu Instanzen der Klasse newClass durch. Der Algorithmus geht dabei davon aus, dass alle Instanzen von oldClass in einem SMOCInstanceWrapper gekapselt sind. Ebenso erwartet der Algorithmus, dass die Klasse SMOCHolder geladen ist und exakt eine Instanz besitzt. Dies muss während der Initialisierung sichergestellt werden. SMOCHolder wird dabei als Kommunikationsbehälter zur Objektübertragung vom JVM-TI zu Java genutzt:

1. JVM-TI: Lädt die Klasse SMOCInstanceWrapper
2. JVM-TI: Lädt die Klasse SMOCHolder
3. JVM-TI: Lädt die einzige Instanz des SMOCHolder
4. Java: Lädt die einzige Instanz des SMOCHolder
5. JVM-TI: Iteriert über alle Instanzen von SMOCInstanceWrapper und filtert diese heraus, die ein Element von oldClass beinhalten:
   a) JVM-TI: Setzt SMOCHolder.oldInstance auf die Referenz im SMOCInstanceWrapper
   b) Java: Setzt SMOCHolder.newInstance auf eine neue Instanz der Klasse newClass
   c) Java: copyFields(SMOCHolder.oldInstance, SMOCHolder.newInstance)
   d) JVM-TI: Setzt die Referenz im SMOCInstanceWrapper auf SMOCHolder.newInstance

Sofern nur ein Thread diese Methode gleichzeitig aufruft, führt diese zum richtigen Ergebnis. Möchte man mehrere Threads haben, so ist sicherzustellen, dass diese Methode nicht parallel ausgeführt wird. Die Hauptursache dafür ist, dass es nur einen SMOCHolder gibt, der sonst gleichzeitig mehrfach verwendet wird. Ebenso ist unklar, was passiert, wenn die gleiche Klasse mehrfach parallel modifiziert wird. Außerdem erlaubt das JVM-TI nur eine Verbindung. Will man jSMOC mit mehreren Threads verwenden, so muss diese Funktion genauer untersucht und gesichert werden. Da diese Untersuchung den Rahmen dieser Arbeit sprengen würde, wird hier darauf verzichtet und lediglich darauf aufmerksam gemacht, dass hier eine kritische Stelle für die Nebenläufigkeit der Bibliothek ist.

Das Ersetzen aller Instanzen beinhaltet die Technik, einzelne Referenzen über assignObjectReference(..) zuzuweisen. Formal schöner wäre es gewesen, alle Instanzen über das JVM-TI in eine Liste zu laden - sprich: einen „getAllInstances“-Operator einzuführen - und danach für jedes Element dieser Liste assignObjectReference(..) aufzurufen. Durch die kombinierte Variante wurde die Laufzeit beschleunigt. Man erkennt, replaceInstances(..) benötigt die Technik und Idee des #=-Operators der Gilgul VM [8], jedoch in einer nicht typsicheren Variante.

Es kann festgehalten werden, dass das JVM-TI nur benötigt wird, um über alle Instanzen einer Klasse zu iterieren. Es wird ebenso festgestellt, dass ein Operator der alle Instanzen zurückgibt, in Java durchaus nützlich wäre, damit der Umweg über das JVM-TI wegfällt. Ferner wird darauf verzichtet, einen solchen Operator anzubieten, da der Umweg, Instanzen aus dem JVM-TI zu extrahieren, über die oben beschriebene Technik und Verwendung des SMOCHolders ein Trick ist und seine Auswirkungen auf Cacheverhalten, Nebenläufigkeit und GarbageCollector ungeklärt sind. Würde man einen solchen Operator anbieten, so muss das Zusammenspiel zwischen JVM-TI und Java besser verstanden werden. Für die Umsetzung von jSMOC reicht es, den Trick an einer Stelle zu verwenden, die ausreichend getestet wurde. In

diesen Tests konnte kein unerwartetes Verhalten festgestellt werden. Dies schließt ein solches jedoch nicht aus. Es ist zu bedenken, dass das JVM-TI nie als Schnittstelle für das laufende Programm selbst gedacht war. Vielmehr soll es externen Programmen, wie dem Debugger, Zugriff auf laufende Programme gewähren. Ferner ist anzumerken, dass ein Operator, der alle Instanzen ermittelt, die Verwendung des Wrapperansatzes gefährden würde. Man könnte über diesen Operator eine Referenz auf ein Objekt innerhalb des Wrappers bekommen und somit die Voraussetzung für die replaceInstances-Methode verletzen.

5.4 Umsetzung der Klassenmodifikation

Möchte man in Java nun eine Klassenmodifikation durchführen, so kann dies immer in den folgenden vier Schritten geschehen:

1. Kopie der zu modifizierenden Klasse erstellen
2. Modifikation an der Kopie durchführen via Bytecode-Manipulation
3. Kopieren der alten Instanzen in neue Instanzen der modifizierten Klasse
4. Umsetzen der Referenzen Dritter von den alten Instanzen zu den Neuen

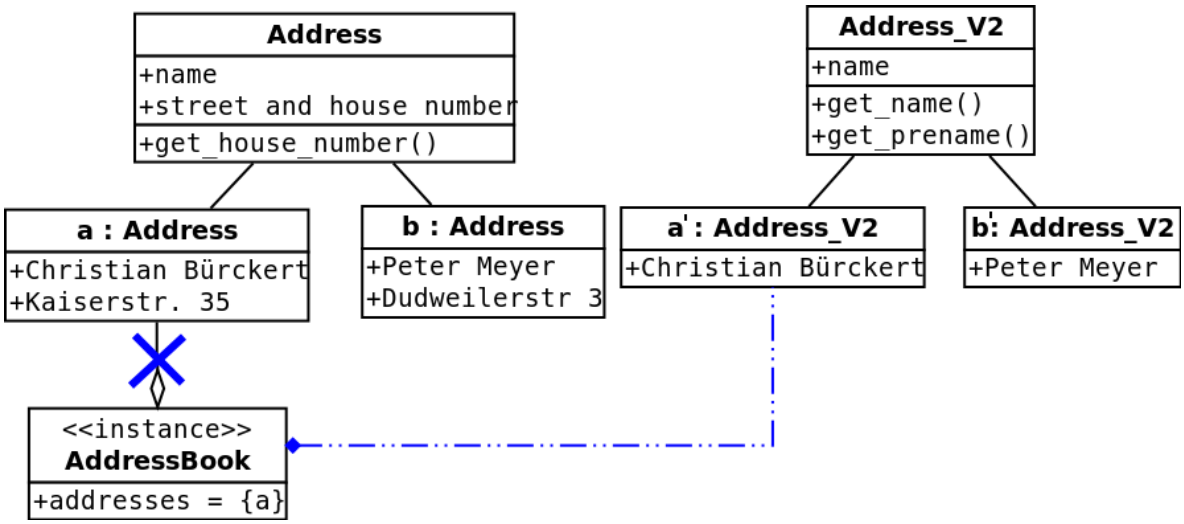


Abbildung 7: Illustration der vier Schritte einer Klassenmodifikation

Die Grafik (Abbildung 7) zeigt den oben beschriebenen Ablauf einer Modifikation, welche ein Feld und eine Methode löscht und danach zwei neue Methoden einfügt. Zuerst erstellt man eine Kopie der Klasse, die man mit einer fortlaufenden Nummer erweitert (hier Address_V2). Diese Klassenkopie kann, da sie noch nicht geladen ist, via Bytecode-Manipulation beliebig verändert werden. Diese Veränderungen werden mit Javassist durchgeführt. Die veränderte Klasse wird an einen präparierten ClassLoader weitergegeben. Damit Modifikationen überhaupt möglich sind, muss dieser präparierte ClassLoader eine Codetransformation an allen Klassen - und somit auch an der neuen Klassenkopie - durchführen. Bei dieser Codetransformation müssen die Seiteneffekte, die durch den SMOCInstanceWrapper entstehen, behoben werden.

Ist die neue Klassen dann geladen, so erzeugt man für jede Instanz der alten Klasse Address eine neue Instanz der Klasse Address_V2. So entstehen im Beispiel die Instanzen a' und b', deren Feldbelegung man von den alten Instanzen anhand des Feldnamens kopiert. Hier wird lediglich die Belegung des Felds name von a nach a' und b nach b' übernommen. Dies geschieht aber nur, wenn der Typ des Felds noch kompatibel ist. Neue oder inkompatible Felder werden mit ihrem Standardwert belegt. Zum Kopieren der Feldbelegung wird die copyFields(..)-Methode verwendet (vgl. Kapitel 5.1 - S. 43). Neue Instanzen modifizierbarer Klassen werden in SMOCInstanceWrapper gekapselt.

Anschließend werden alle Referenzen zu den alten Instanzen der Klasse Address a und b zu den neu erzeugten Instanzen der Klasse Address_V2 a' und b' umgesetzt. Im Beispiel wird nur die Referenz des AddressBook zu a auf a' umgesetzt.

Dabei wurden in der praktischen Implementierung Schritt drei und vier zur Methode replaceInstances(..) zusammengefasst.

5.5 Die Architektur von jSMOC

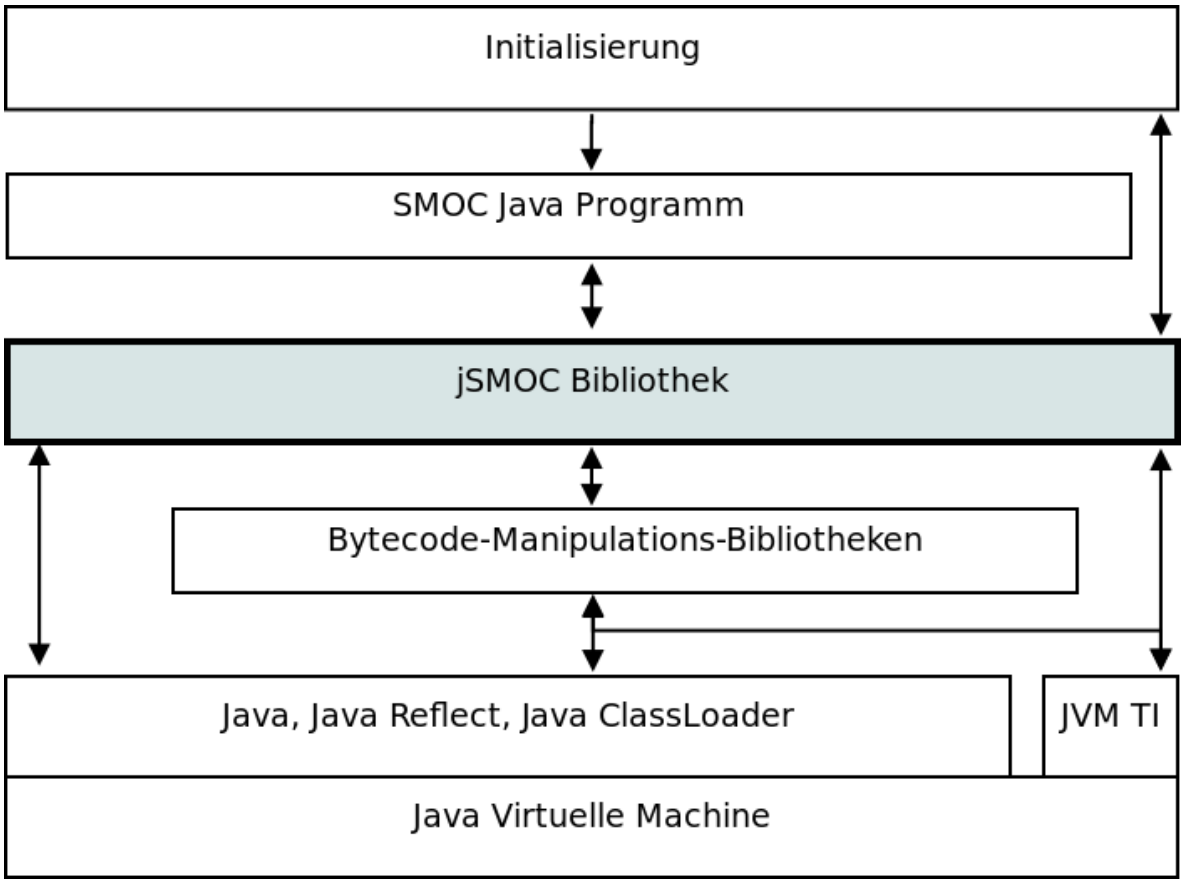


Abbildung 8: Architektur von jSMOC

Die jSMOC-Bibliothek liegt als eine Schicht über den Bibliotheken zur Bytecode-Manipulation und dem java.lang.reflect-Paket sowie dem ClassLoader. Die Initialisierung hat die Hauptaufgabe, eine Verbindung zum JVM-TI herzustellen und das eigentliche Programm in einem

präparierten ClassLoader auszuführen, der eine Codetransformation durchführt. Bei dieser Codetransformation geht es im Wesentlichen darum, den SMOCInstanceWrapper einzubauen und seine Seiteneffekte zu beheben. Der Programmierer soll dabei möglichst wenige Einschränkungen erfahren. Der SMOCInstanceWrapper soll sozusagen nicht sichtbar sein. Eine weitere Aufgabe ist es, die durch den Compiler zugesicherten Methodenaufrufe und Feldzugriffe auf modifizierbare Klassen in unsichere Aufrufe zu transformieren, damit die Klassen ausgetauscht werden können.

Die jSMOC-Bibliothek bietet die Schnittstelle zur Reflexion. Über sie wird die Programmintrospektion vereinfacht und die Modifikation angeboten. Der Kern beinhaltet also Methoden, welche die fehlenden SMOC-Operatoren simulieren und somit die Programmierung flexibler SMOC-Programme ermöglichen.

5.6 Codetransformation

Die Codetransformation hat die Aufgabe, die durch die Einführung des SMOCInstanceWrappers erzeugte Indirektion zu verstecken und dem Programmierer so eine gewohnte Entwicklung zu garantieren. Ebenso hat diese Codetransformation die Aufgabe, die SMOCInstanceWrapper einzuführen.

Die Codetransformation wird vom SMOCLoader angestoßen. Programme werden in seinem Kontext ausgeführt und so wird jede Klasse über den SMOCLoader geladen. Der SMOCLoader wird die Klassen, anstatt sie direkt zu laden, erst transformieren und dann den transformierten Bytecode als Klasse definieren.

5.6.1 Signaturen in Java

Um die Codetransformationen verstehen zu können, muss man die in Java verwendeten Signaturbeschreibungen verstehen. Java wählt hierfür folgende Abkürzungen:

- V für VOID
- Z für BOOLEAN
- C für CHAR
- B für BYTE
- S für SHORT
- I für INT
- F für FLOAT
- J für LONG
- D für DOUBLE
- L{Klasse}; für OBJEKT

Will man nun einen Typ angeben, so werden auf Bytecodeebene immer diese Kurzformen genutzt. Zum besseren Verständnis folgen einige konkrete Beispiele:

1. ()V beschreibt eine VOID-Methode ohne Parameter.

2. (II)Z beschreibt eine BOOLEAN-Methode mit zwei INT-Parametern.

3. (Ljava/lang/Object;I)Ljava/lang/String; beschreibt eine Methode mit einem Parameter Objekt einem Parameter INT und als Rückgabe einen String.

Kritisch anmerken könnte man, dass L für Object steht und J für LONG. Eine Begründung hierfür ist nicht zu finden. Der Typ Z wird vom Compiler hin und wieder durch B ersetzt. Da ASM diese Beschreibungen aufgreift und kein Parser für diese Ausdrücke zu finden ist, stellt die jSMOC-Bibliothek einen solchen bereit. jSMOC erwartet beim dynamischen Aufruf von Methoden ebenfalls eine Signatur, die in dieser Form angegeben werden muss. Der jSMOC-Programmierer muss diese also beherrschen. Dies ist notwendig, damit zwei überladene Methoden unterschieden werden können. Ruft man zum Beispiel die Methode otto("Hallo", 15) auf, es gibt jedoch die Methoden otto(String, long) und otto(String, int) so ist unklar, welche Methode gewählt wurde. Noch kritischer wird es bei der Übergabe von null, da dieser Parameter jedem Typ zugeordnet werden kann. Interessanterweise scheint der Javacompiler damit keine Probleme zu haben. So kann er im folgenden Programm ganz klar entscheiden, dass es sich bei null um einen String handelt:

```
1     public static void x(String abc){
2         System.out.println("String");
3     };
4     public static void x(Object abc){
5         System.out.println("Object");
6     };
7
8     public static void main(String[] args) {
9         x(null);
10    }
```

Der Java Compiler verwendet immer die speziellste Variante einer Methode. Würde zusätzlich eine Methode x(Integer a) angeboten, könnte der Compiler null ebenfalls nicht mehr zuordnen, da Integer genauso speziell ist wie String. In jSMOC würde dies jedoch bedeuten, alle Methoden zu ermitteln und deren Parameter zu vergleichen, und dies bei jedem Aufruf. Da dies zu langsam wird, muss die genaue Signatur bei einem Methodenaufruf mit angegeben werden.

5.6.2 Voranalyse

Im jSMOC-Ansatz werden nur Klassen transformiert, die eine modifizierbare Klasse referenzieren oder selbst modifizierbar sind. Dies ermöglicht es, die Einschränkungen die jSMOC an die Programmierung stellt, teilweise auf modifizierbare Klassen zu beschränken. So können zum Beispiel generische Typen in nicht modifizierbaren Klassen frei verwendet werden.

Um zu erkennen welche Klassen modifizierbare Klassen referenzieren, müssen alle Klassen in einer Voranalyse durchlaufen werden. Dabei wird festgestellt,

1. ob die Klasse modifizierbar ist, also ob die Klasse SMOCObject implementiert;

2. ob die Klasse ein Feld mit einem Typ einer modifizierbaren Klasse besitzt;

3. ob die Klasse eine Methode mit einem modifizierbaren Parametertyp oder Rückgabetyp hat;

4. ob die Klasse eine innere Klasse hat, wenn ja, wird die Ausführung mit einem Fehler abgebrochen;

5. ob eine Methode dieser Klasse eine lokale Variable einer modifizierbaren Klasse besitzt.

Umgesetzt wird die Voranalyse mit dem ASM-Besuchermuster (siehe Grafik 16 - S. 77). Die gewonnene Information über Klassen wird im SMOCLoader abgespeichert, damit er bei anderen Klassen auf diese Informationen zurückgreifen kann.

Wurde bei der Analyse festgestellt, dass eine Klasse eine modifizierbare Klasse referenziert oder selbst modifizierbar ist, so wird die Transformation angestoßen. Um die Transformation für Debuggingzwecke sichtbar zu machen, kann vor und nach der Transformation der Bytecode in lesbarer Form mit dem ASM TraceClassVisitor in die Dateien ./bytecode/Klasse_original.txt und ./bytecode/Klasse_transformed.txt gespeichert werden. Über den Vergleich dieser beiden Dateien können die Veränderungen sichtbar gemacht werden.

5.6.3 Transformationen vereinfachen

Das Vorgehen in dieser Arbeit ist es, komplizierte Ausdrücke in statische Methoden auszulagern und diese dann über die Transformation an den später beschriebenen Stellen einzubauen. Dies hat den Vorteil, dass diese Ausdrücke nun in Java und nicht in Bytecode programmiert werden müssen.

Man könnte den Bytecode dieser Methoden auch einfügen. Dies würde jedoch eine händische Überarbeitung erfordern, da lokale Variablen nicht einfach übertragen werden können. Lokale Variablen werden im Bytecode am Ende einer Methode deklariert. Fügt man den Inhalt der statischen Methoden ein, so kann es zu Konflikten der lokalen Variablen kommen, da zum Beispiel ein Name doppelt verwendet wird. Auch würde das Einfügen der Methoden lediglich einen Methodenaufruf sparen. Ebenso würde es die Übersichtlichkeit der Transformation erheblich beeinträchtigen. In dieser Arbeit wird daher auf das Einfügen verzichtet.

5.6.4 Konstruktoraufrufe

Konstruktoraufrufe müssen verändert werden, da Objekte modifizierbarer Klassen in SMOCInstanceWrapper gekapselt werden müssen. Um diese Modifikation zu verstehen, betrachtet man zuerst einen einfachen Konstruktoraufruf in Java:

```
1   Address a = new Address("christian", "Heidenkopferdell", 45);
2
3   NEW Address                    //Legt Referenz neuer Instanz auf den Stack
4   DUP                            //Duplizieren der Referenz
5
6   LDC "christian"                //Laden des Strings christian
7   LDC "Heidenkopferdell"
8   BIPUSH 45                      //Laden des ints  45
9   INVOKESPECIAL Address.<init>   //Aufruf der Konstruktormethode
10      (Ljava/lang/String;
11      Ljava/lang/String;I)V
12
13  ASTORE 1                       //Speichern in lok. Variable 1
```

Listing 11: Konstuktoraufruf in Bytecode

Zuerst wird also ein Objekt erzeugt und dessen Referenz auf den Stack gelegt. Da die Referenz in Java später in die Variable 1 gelegt werden soll, wird diese Referenz dupliziert. Anschließend werden die Parameter auf den Stack gelegt. Die dann aufgerufene Konstruktormethode nimmt die Parameter und die durch DUP erzeugte Referenz als this vom Stack. ASTORE nimmt die von NEW erzeugte Referenz vom Stack und legt diese in die lokale Variable a ab.

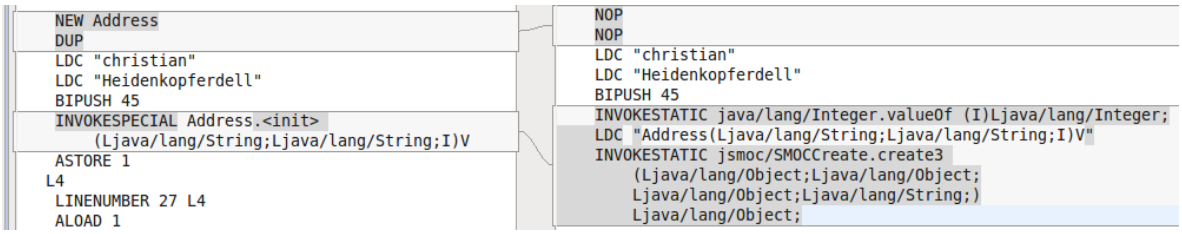


Abbildung 9: Ersetzen des Konstruktoraufrufs einer modifizierbaren Klasse

In jSMOC müssen Objekte entsprechend ihrer Klassenversion erzeugt werden. Zusätzlich müssen die Instanzen gekapselt werden. Anstatt des Konstruktors muss also eine Methode aufrufen, die genau dies leistet. Dazu stehen in der Klasse SMOCCreate statische Methoden zum Erzeugen von Objekten modifizierbarer Klassen bereit. Diese Methoden erwarten als Parameter die Parameter des Konstruktors und zusätzlich einen String, der den Namen der Klasse beinhaltet und die Signatur des Konstruktors, der aufgerufen werden soll. Die Codetransformation wird in Abbildung 9 dargestellt.

Das Erzeugen der Instanz durch NEW geschieht in der Methode SMOCCreate.create3. Diese statische Methode ersetzt einen Konstruktor mit drei Parametern, die als Objekte übergeben werden müssen, und gibt die Referenz auf die erzeugte Instanz zurück. Die Rückgabe funktioniert in Java, indem diese auf den Stack gelegt wird. Das Erzeugen über NEW sowie das Duplizieren der Referenz mit DUP werden somit nicht mehr benötigt und durch NOP-Operationen ersetzt. Da alle Parameter als Objekte übergeben werden, müssen primitive Werte wie int in Objekte umgewandelt werden. Im Beispiel wird der int 45 in das Integer-Objekt 45 umgewandelt. Dies geschieht durch den Aufruf der statischen Methode java/lang/Integer.valueOf() und ist die übliche Vorgehensweise des Java Compilers für das Umwandeln in Objekte.

Wird ein Konstruktor aufgerufen, ohne dass die so entstandene Referenz später benötigt wird, so wird der Compiler kein DUP erzeugen, um die Referenz zu duplizieren. Da jedoch SMOCCreate.create3 eine solche Referenz auf dem Stack hinterlegt, muss diese durch ein POP entfernt werden. Dies ist ebenfalls der übliche Weg des Compilers, wenn Methodenrückgabewerte nicht benötigt werden. Ein Beispiel (vergleiche Transformation in Abbildung 10) hierfür wäre:

```
1 new Address("christian", "Heidenkopferdell", 45);
```

Es gibt create-Methoden mit bis zu 20 Konstruktorparametern. Dadurch ensteht die Einschränkung, dass Konstruktoren einer modifizierbaren Klasse nicht mehr als 20 Parameter haben dürfen. Dies kann leicht auf wesentlich mehr erweitert werden und ist somit keine wirkliche Einschränkung. Dazu müssen lediglich Methoden in SMOCCreate angelegt werden mit den Namen create21, create22, ... und entsprechend der anderen Methoden aufgebaut werden. Das Generieren dieser Methoden kann automatisiert werden. Die Erweiterung kann

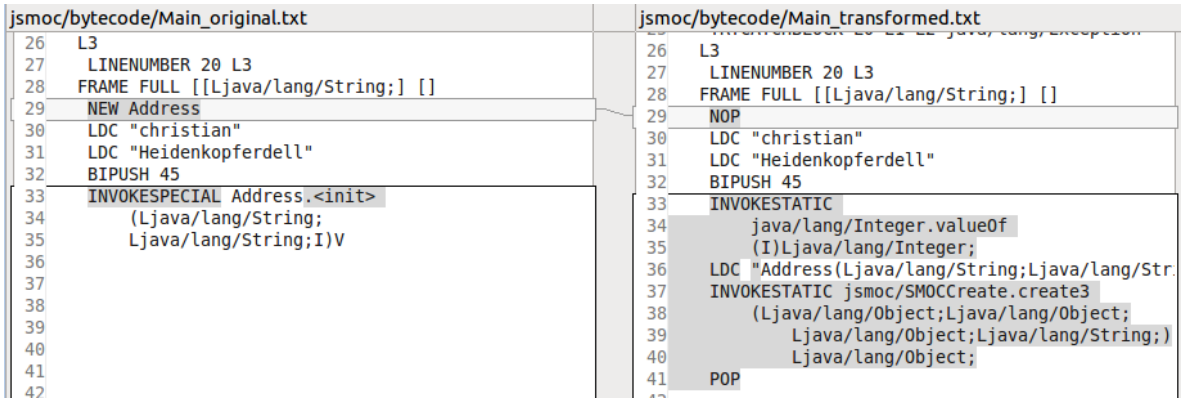


Abbildung 10: Ersetzen des Konstruktoraufrufs ohne DUP

jedoch nicht zur Laufzeit selbst geschehen, da jederzeit neue Konstruktoren mit mehr Argumenten erzeugt werden könnten und die SMOCCreate-Klasse nicht modifizierbar gestaltet werden kann.

5.6.5 Ersetzen von Typen

Beim Erzeugen von Objekten modifizierbarer Klassen, erhalten wir durch die Veränderungen zum Austausch der Konstruktoren nun SMOCInstanceWrapper. Da diese Referenzen in lokale Variablen, Parameter, Rückgabewerte und Felder abgelegt werden, müssen die Typen dieser auf SMOCInstanceWrapper geändert werden, da sonst die VM die Zuweisung nicht erlaubt. Die Codetransformation tauscht also alle Typen von Feldern, lokalen Variablen, Rückgabewerten und Parametern, wenn diese Typen eine modifizierbare Klasse repräsentieren.

Diese Codetransformation hat also einen anderen Charakter als die bisher vorgestellten. Sie entfernt die Nebeneffekte, die durch die Einführung der SMOCInstanceWrapper entstehen.

Diese Transformation kann recht einfach durchgeführt werden: Wann immer ASM auf einen modifizierbaren Typ stößt, wird dieser durch SMOCInstanceWrapper ersetzt. Die Folgen der Methodenaufrufe und Feldzugriffe, die durch die Verallgemeinerung des Typs entstehen, müssen im Folgenden ebenfalls bearbeitet werden.

5.6.6 Methodenaufrufe

Der Methodenaufruf in Java erfolgt über die Bytecodeinstruktion INVOKEVIRTUAL. Methodenaufrufe müssen korrigiert werden, da der als this übergebene versteckte Parameter nun vom Typ ein SMOCInstanceWrapper ist und somit keine Methoden besitzt. Ein Methodenaufruf in Java ist wie folgt aufgebaut:

```
1 Address a = new Address("christian", "Heidenkopferdell", 45);
2 boolean result = a.print(DEVICE_CONSTANT_5);
3
4 ALOAD 1   //Laden von a auf den Stack
5 ICONST_5  //Laden von 5 auf den Stack
6 INVOKEVIRTUAL Address.print (I)Z  //Aufruf der Methode
7 ISTORE 2  //Speichern der Rückgabe in result.
```

Listing 12: Aufruf von print() auf Gerät 5

Bei diesem Beispiel nimmt der Aufruf INVOKEVIRTUAL a und 5 vom Stack und legt true oder false zurück. Eine statische Methode nimmt lediglich die ihr angegebenen Parameter vom Stack. Ersetzt man also den Aufruf print(I)Z durch den Aufruf einer statischen Methode invoke(Ljava/lang/Object;I)Z, so würde man das this als ersten Parameter der statischen Methode invoke als Objekt erhalten. Zusätzlich müsste man dieser statischen Methode noch übergeben, welche Methode aufgerufen werden soll. Da man jedoch nicht für jede erdenkliche Signatur eine Methode anbieten kann, bietet die Klasse SMOCCall statische Methoden an, die lediglich Objekte übernehmen. Es sind Methoden vorhanden, um Aufrufe mit 0 bis 20 Parametern zu ersetzen. Dadurch ergibt sich die Einschränkung, dass Methoden modifizierbarer Klassen maximal 20 Parameter haben dürfen. Die statischen Methoden der SMOCCall-Klasse haben folgende Parametergestaltung: public static Object invokeN(Object that, {N Parameter Objekt}, String methode). Der erste Parameter that ersetzt das this, da dieser Name reserviert ist. Anschließend kommen die Parameter und am Ende der Methodenname mit Signatur. Die Reihenfolge ergibt sich aus der bereits durch den Compiler vorgegebenen Reihenfolge der Objekte auf dem Stack, damit diese nicht verändert werden muss. Die Methode mit Signatur kommt am Ende, da an dieser Stelle am einfachsten etwas auf den Stack hinzugefügt werden kann. Da nur Object als Parameter verwendet wird, müssen alle primitiven Typen auf dem Stack sowie der Rückgabewert transformiert werden. Die statische Methode invokeN ruft dann die eigentliche Methode über Reflexion auf und berücksichtigt dabei die Indirektion des SMOCInstanceWrapper.

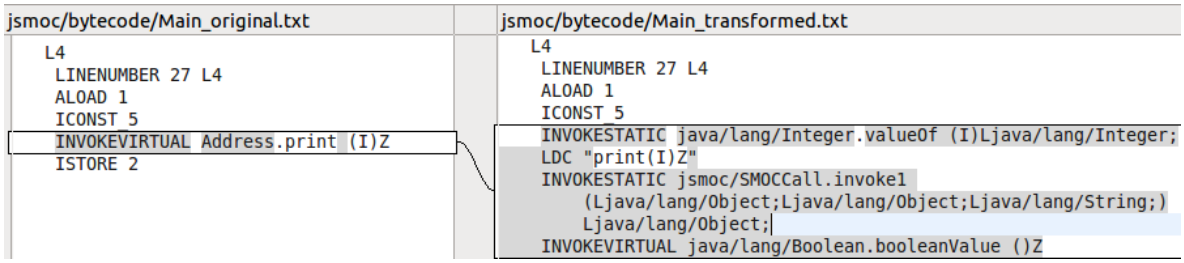


Abbildung 11: Ersetzen des Methodenaufrufs zu einer modifizierbaren Klasse

Beispielhaft sieht man eine solche Transformation in Abbildung 11. Hier wird zuerst der übergebene Wert des Typs int in einen Integer umgewandelt. Danach wird die Methode mit Signatur als String mit LDC auf den Stack gelegt. Anschließend wird die statische Methode SMOCCall.invoke1 aufgerufen und der Rückgabewert vom Objekttyp Boolean wieder zum primitiven boolean konvertiert.

Haben Methoden den Rückgabetyp void, so muss nach dem statischen Aufruf ein POP erfolgen, da sonst ein unerwartetes null auf dem Stack liegen würde.

5.6.7 Felderzugriffe

Ähnlich der Methodenzugriffe müssen auch die Feldzugriffe ersetzt werden. Java benutzt hierfür Bytecode-Instruktionen PUTFIELD und GETFIELD; für statische Felder werden PUTSTATIC und GETSTATIC verwendet. Diese müssen, wenn sie auf modifizierbaren Klassen aufgerufen werden, ebenfalls ersetzt werden, da die Referenz, die auf dem Stack liegt, um das Objekt zu identifizieren, eine SMOCInstanceWrapper-Instanz ist und das eigentliche Objekt kapselt. Da die Anzahl der Signaturmöglichkeiten hier endlich ist, wird in der Klasse SMOCField für

jeden primitiven Typ und Objekt eine statische Methode angeboten, um die Feldzugriffe zu ersetzen. Die Signaturen dieser Methoden sind folgendermaßen aufgebaut:

- void PUTFIELD_[TYP] (Object that, [TYP] value, String fieldName)
- [TYP] GETFIELD_[TYP] (Object that, String fieldName)
- void PUTSTATICFIELD_[TYP] ([TYP] value, String fieldName, String clazz)
- [TYP] GETSTATICFIELD_[TYP] (String field, String clazz)

Dabei kommt [TYP] aus der Menge {L,Z,C,B,S,I,F,J,D}. Dies ist die Menge alle Typen ohne VOID, da es keine VOID-Felder gibt. Diese Methoden entkapseln den in that übergebenen SMOCInstanceWrapper und ermitteln die Feldbelegung unter der aktuellen Klassenversion über Reflexion. Solche Codetransformationen sind beispielhaft in Abbildung 12 zu sehen.

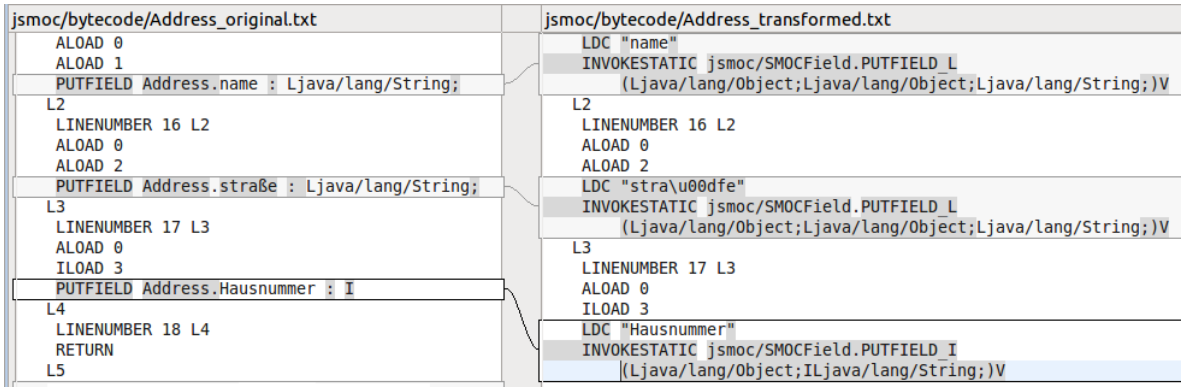


Abbildung 12: Ersetzen eines Feldzugriffs einer modifizierbaren Klasse

5.6.8 getClass()-Transformation

Java bietet bei Objekten die Möglichkeit, die Klasse über getClass() zu ermitteln. Dieser Operator muss auf SMOCInstanceWrapper-Objekten ebenfalls entkapselt werden. Dazu steht in der Klasse SMOCCall eine statische Methode zur Verfügung, welche die Referenz als erstes übergeben bekommt, entkapselt und dann die Klasse zurückgibt. Eine Anpassung des Stacks oder Rückgabetyps ist nicht erforderlich.

Um diese Manipulation in ASM durchzuführen, wird im SMOCMethodAdapter folgender Code verwendet:

```
1 if (name.equals("getClass") && owner.equals("java/lang/Object")) {
2     super.visitMethodInsn(Opcode.INVOKESTATIC, "jsmoc/SMOCCall", name,
3         "(Ljava/lang/Object;)Ljava/lang/Class;");
4     return;
5 }
```

5.6.9 Transformation als Quellcode

Die Bytecode-Transformationen sind schwerer zu erfassen als Quellcode-Transformationen. Zu diesem Zweck wird in diesem Kapitel die Bytecode-Transformation exemplarisch als Quellcodetransformation dargestellt. Die meisten Bytecode-Transformationen konnten entwickelt werden, indem man die Transformation in Java-Quellcode umsetzt, jeweils kompiliert und die Unterschiede des Bytecodes analysiert. Dies ist einfacher, als den kompletten Javacompiler zu analysieren. Diese Vorgehensweise wurde ebenfalls verwendet, um Fehler in Bytecode-Transformationen zu verstehen. Es ist daher hilfreich, die Bytecode-Transformationen auch auf Quelltextebene zu veranschaulichen. Dargestellt werden im Folgenden die Transformation eines Konstruktors, eines Methodenaufrufs und eines Feldzugriffs.
Vorher:

```
1 Address christian = new Address("Christian B.", "Kaiserstr. 35");
2 christian.print(printer);
3 List<Address> addressbook = new ArrayList<Address>();
4 addressbook.add(christian);
5 System.out.println(christian.street);
```

Nachher:

```
1  SMOCInstanceWrapper christian =
2      SMOCCreate.create3("Christian B.", "Kaiserstr.", 35,
3      "(Ljava/lang/String;Ljava/lang/String;I)V");
4
5  SMOCCall.invoke1(christian, printer, "print(Ljava/lang/Object;)V");
6
7  List<SMOCInstanceWrapper> addressbook = new ArrayList<SMOCInstanceWrapper>();
8  addressbook.add(christian);
9
10 System.out.println(SMOCField.GETFIELD_L(christian, "street"));
```

Die Transformation von List<Address> zu List<SMOCInstanceWrapper> muss nicht unbedingt durchgeführt werden. Dies ist lediglich eine Compilerzusicherung und hat auf Bytecodeebene keinerlei Bedeutung. Auf der Bytecodeebene werden alle generischen Typen durch Object ersetzt. In der Quellcode-Transformation müssen diese jedoch geändert werden, da sonst ein Kompilieren nicht möglich ist.

5.7 Manipulation zur Laufzeit

Die jSMOC-Bibliothek ermöglicht die Manipulation zur Laufzeit. Die SMOCReflectClass-Schnittstelle bietet die dafür benötigten Methoden an. Exemplarisch wird das einleitende Beispiel als Modifikation mit der jSMOC-Bibliothek dargestellt: (siehe auch Abbidlung 7 - S. 46).

```
1 SMOCReflectClass address = SMOCReflectProgramm.reflectClass(Address.class);
2 address.beginModification();
3 address.delField("street_and_house_number");
4 address.delMethod("get_house_number", null);
5 address.addMethod("public String get_name() { return name; }");
6 address.addMethod("
```

```
7       public String get_prename() {
8           return name.split(\" \")[0];
9       }
10  ");
11  address.endModification();
```

Der Aufruf von beginModification() erzeugt eine Liste von Modifikationen, die durch die nachfolgenden Aufrufe gefüllt wird. Durch endModification() wird die Liste dann abgearbeitet und die Klasse durch eine neue Version ersetzt. Anschließend werden alle vorhandenen Instanzen durch Instanzen der neuen Klassenversion ersetzt.

6 Abweichungen zum Proposal

Zu dieser Bachelorarbeit wurde vor der Anmeldung ein planendes Proposal erstellt. In diesem Proposal wurden einige Techniken des Ansatzen anders konzipiert, als sie nun umgesetzt wurden. Diese Planungsänderungen haben meistens einen erweiternden Charakter oder sind Reaktionen auf zur Entwicklungszeit aufgetretenen Probleme. Die im Proposal definierten Anforderungen an die jSMOC-Bibliothek konnten bis auf die Methode getMethodSource() erfüllt werden. Es sollen nun diese zur Entwicklungszeit ermittelten Erkenntnisse und Probleme festgehalten werden.

6.1 SMOCInstanceWrapper und UpdateThread

Der UpdateThread verfolgte im Proposal die Idee, alle Referenzen auf Instanzen zu finden und zu ändern, wenn eine neue Klassenversion eingeführt wurde. Dazu wurde die Iteration über jedes Element des laufenden Programms angestrebt. Diese musste wegen der Einschränkungen im JVM-TI, den Stack nur in angehaltenen Threads zu analysieren, in einen separaten Thread ausgelagert werden. Diesem wurde dazu in der Methode endModification() die Liste der Modifikationen übergeben, die abgearbeitet werden konnten, während der Thread in endModification() auf eine Rückmeldung des UpdateThreads wartet und somit angehalten ist. Dieser Ansatz hat jedoch zwei Probleme:

1. Das Protokoll zur Übergabe der Modifikationen an den UpdateThread macht die Erweiterbarkeit der jSMOC-Bibliothek auf Nebenläufigkeit schwerer, da alle laufenden Threads zur Manipulation angehalten werden müssen. Da die suspend-Methode zu gravierenden Fehlern führen kann, wurde der Einsatz von mehreren Threads im Proposal abgelehnt. Der UpdateThread wurde nur benötigt, da der Stack zur Laufzeit analysiert werden musste. Dies ist durch die Einführung der SMOCInstanceWrapper nicht mehr notwendig. Die Modifikation wird jetzt vom aktuell aktiven Thread ausgeführt. Die Bibliothek ist damit immer noch nicht threadsicher. Dies kann jedoch nun an geeigneter Stelle eingefügt und überarbeitet werden. Dazu müssen die replaceInstances()-Funktion sowie die Operatoren zur Klassenmodifikation abgesichert werden. Danach sind ausführliche Tests notwendig, da die Verwendung des JVM-TI mit Nebenläufigkeit aus dem laufenden Programm heraus, welches gleichzeitig analysiert werden soll, nicht untersucht ist.

2. Die Suche aller Instanzen durch Iteration über alle Variablen, Stackframes und Felder ist langsam. Die SMOCInstanceWrapper erlauben das Verändern einer Referenz in konstanter Laufzeit. Nachteilig wirkt sich der Zugriff auf Methoden und Felder aus, da diese nun eine weitere Indirektion beim Zugriff auswerten müssen.

6.2 Generische Typen

Durch die weitere Analyse der generischen Typen konnte ermittelt werden, dass diese zur Laufzeit keine Auswirkung haben sollten. Generische Typen werden lediglich für eine Compilerprüfung verwendet. Zur Laufzeit werden alle generischen Variablen auf die Klasse Object verallgemeinert. Eigentlich sollte also die jSMOC-Bibliothek dann auch mit generischen Typen funktionieren. Unerklärlicherweise stürzt jedoch die JVM in der replaceInstances()-Methode an einer völlig unerwarteten Stelle ab. Diese Stelle hat mit generischen Typen eigentlich nichts

zu tun. So findet der Absturz an der Stelle statt, an der der SMOCHolder über das JVM-TI geladen werden soll und tritt nur auf, sobald eine modifizierbare Klasse generische Typen verwendet.

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00007f9ed38340cb, pid=15830, tid=140320022562560
#
# JRE version: 6.0_20-b20
# Java VM: OpenJDK 64-Bit Server VM (19.0-b09 interpreted mode linux-amd64
                                                          compressed oops)
# Derivative: IcedTea6 1.9.13
# Distribution: Ubuntu 10.10, package 6b20-1.9.13-0ubuntu1~10.10.1
# Problematic frame:
# V  [libjvm.so+0x71a0cb]
#
# An error report file with more information is saved as:
# /home/christian/Programs/eclipse_java/workspace/jsmoc/hs_err_pid15830.log
#
# If you would like to submit a bug report, please include
# instructions how to reproduce the bug and visit:
#   https://bugs.launchpad.net/ubuntu/+source/openjdk-6/
#
```

Listing 13: Fehlermeldung, mit der die JVM bei Verwendung von Generics abstürzt

Da dieser Fehler nicht geklärt werden kann und der Zeitpunkt des Absturzes scheinbar keinen Zusammenhang mit den generischen Typen hat, jedoch durch diese verursacht wird, sollten generische Typen nach wie vor nicht in modifizierbaren Klassen verwendet werden. Es gibt zwei Möglichkeiten der Ursache dieser Fehlermeldung: Erstens könnte die JVM einen Bug haben. Dies ist in Anbetracht der Situation, dass die JVM eine Verbindung zu sich selbst aufbaut und Modifikationen des laufenden Programms durchführt, was weder getestet noch untersucht wurde, nicht gerade unwahrscheinlich. Zweitens könnte eine Bytecode-Transformation fehlerhaft sein und Spätfolgen verursachen.

6.3 Statische Funktionsauslagerung

Die Funktionen zur Instanzenerzeugung, Methodenaufrufe und Felderzugriffe sollten ursprünglich komplett auf der Bytecodeebene umgesetzt werden. Die Entwicklung im Bytecode stellte sich jedoch als komplexer als erwartet heraus. Allein die Methode zum Aufruf von Methoden über Reflexion (SMOCCall.invoke) benötigt 40 Zeilen Java-Quellcode. Diese in Bytecode darzustellen und richtig in einen fremden Kontext einzubauen, wäre eine große Fehlerquelle gewesen und hätte die Entwicklung und Fehlerkorrektur erheblich erschwert. Darum wurden diese Methoden in Java-Quellcode ausgelagert und über den Verlust eines zusätzlichen Methodenaufrufs eingebaut. Da sich diese Vorgehensweise als einfach und entwicklerfreundlich herausstellte und Java-Quellcode übersichtlicher als Bytecode ist, wurden Konstruktoren und Feldzugriffe ebenfalls durch solche statischen Methoden ersetzt. Dies war im Proposal so nicht vorgesehen, hat aber zum dort vorgestellten Ansatz lediglich den Nachteil, dass die Anzahl der Parameter in Funktionen und Konstruktoren auf 20 beschränkt ist. Dies kann jedoch durch Erweitern der SMOCCall- und SMOCCreate-Klassen beliebig erweitert werden. Dabei zählen

Parameter, die über die ...-Syntax[27] eingefügt werden, lediglich als ein Parameter.

6.4 Entfernen von getMethodBody()

Ursprünglich sollte noch der Operator getMethodBody() angeboten werden. Dieser Operator sollte den Java-Quellcode einer Methode zurückgegeben. Irrtümlicherweise wurde dies angegeben, da die ASM-Besuchermethode visitCode(String source) suggerierte, den Quellcode zurückzugeben. Dies ist jedoch nicht der Fall. Die Methode liefert lediglich - falls vorhanden - den Pfad zur Quelltextdatei. Im Allgemeinen sind diese jedoch nicht vorhanden. Daher musste auf diesen Operator verzichtet werden. Man könnte jedoch überlegen, einen solchen Operator durch „Disassembler“ für Javabytecode zu implementieren.

6.5 Verwendung von Javareflexion

Das Proposal ging davon aus, dass die Verwendung der Javareflexion, die eigentlich nur eine Introspektion ist, durch den damals gewählten Ansatz zu Verwirrung des Programmiers führen könnte. Durch die getClass()-Transformation ist dies jedoch nahezu ausgeschlossen. Der Programmierer muss sich nur im Klaren sein, dass er bei dieser Methode die aktuelle Klassenversion und den veränderten Namen zurück bekommt. Möchte er die ursprünglich kompilierte Version haben, so muss er die Klasse.class-Syntax verwenden.

[27] In Java ist es möglich, die Parameterzahl flexibel zu gestalten. So kann eine Methode public void do(int ...ints), mit do(1), do(1,5), do(2,6,3), usw. aufgerufen werden. Java interpretiert ints dann als int-Array.

7 Evaluation

7.1 Testen der Implementierung

Üblicherweise werden Softwareprojekte mit jUnit [10] getestet. Das Framework bietet jedoch keine Möglichkeit, die von jSMOC benötigte Initialisierungsphase zur Codetransformation einzufügen. Daher konnte die Entwicklung von jSMOC nicht über jUnit getestet werden. Vielmehr mussten Einzelprogramme für die zu testenden Sachverhalte geschrieben werden, die dann als Tests ausgeführt wurden. Viele Fehler und Probleme mussten aber auch über die Analyse der Bytecodeausgaben des TraceLogVisitors gelöst werden. Zusätzlich zu diesen Tracelogs bietet die jSMOC Bibliothek ausführliche Fehlermeldungen an, die eine genaue Analyse des Sachverhalts erlauben. Trotz der gravierenden Einwirkungen auf die Geschwindigkeit werden an fast allen Stellen hinweisende Debugausgaben getätigt. Diese können nach Relevanz über die Startparameter gefiltert werden. Die so ausgegebenen Debuginformationen besitzen immer den folgenden Aufbau: Paket.Klasse.Methode:Codezeile Debuginformation und werden auf der Fehlerkonsole ausgegeben.

Man kann diese detailierten Debugausgaben auch in eigenen SMOC-Programmen nutzen. Dazu muss die statische Methode SMOCLog.debug(text) verwendet werden.

Tests existieren für folgende Sachverhalte:

- Reflexion im Sinne der Introspektion der wichtigsten Elemente, wie Felder und Methoden und deren Zugriffsmodifikatoren.
- Manipulation von Methoden und Feldern einer Klasse.
- Integrationstest (dargestellt in Listing 14)

```
1
2       //Konstruktor ohne DUP
3       new Address("christian", "XYZ", 33);
4
5       //Konstruktor mit DUP
6       Address christian = new Address("christian", "Heidenkopferdell", 45);
7
8       //Aufruf von print mit POP
9       christian.print(33, "Hallo", "Whatever");
10
11      //Rückgabe von Print daher kein POP
12      boolean x = christian.print(5);
13
14      //Testausgabe
15      System.out.println(x);
16
17      //Adressbuch
18      ArrayList<Address> book = new ArrayList<Address>();
19      book.add(christian);
20
21      //Generischer Typ von book - Rückgabetest
22      Address a = book.get(0);
23      //Aufruf klappt?
24      a.print(6);
25
26      //getClass() Transformation testen
```

```
27          System.out.println(a.getClass());
28
29          //Reflexionsklasse laden
30          SMOCReflectClass rc = SMOCReflectProgram.reflectClass(Address.class);
31          System.out.println(rc.getName());
32
33          //Aufruf über ?-Operator
34          try {
35              rc.invoke(a, "print(I)V", 7);
36          } catch (Exception e) {
37              e.printStackTrace();
38              System.exit(1);
39          }
40
41          //Testen der Feldintrospektion
42          System.out.println("---------->"
43                  + rc.getValue(a, "name"));
44          System.out.println("---------->"
45                  + rc.getValue(a, "strasse"));
46
47          //Modifikation: zwei Methoden hinzufügen
48          rc.beginModification();
49          rc
50                  .addMethod("public void outi(){
51                  System.out.println(\"<<>
52                          newouti: \"+name); }");
53          rc
54                  .addMethod("public void outi2(){
55                   System.out.println(\"<<>
56                          newouti2: \"+name); }");
57          rc.endModification();
58
59
60          //Klasse ueberpruefen sollte Address_V1 sein
61          System.out.println("My name is: " + a.getClass().getName());
62
63          //Methoden ueber ?-Operator aufrufen
64          rc.invoke(a, "outi()V");
65          rc.invoke(a, "outi2()V");
66          rc.invoke(a, "print(I)Z", 55);
67
68          //Weitere Modifikation
69          //Methode loeschen und Attribut hinzufuegen
70          rc.beginModification();
71          rc.delMethod("print", "(I)Z");
72          rc.addField("private final String newAttr;");
73
74          //Freie Modifikation mit javassists CtClass
75          rc.modify(new Modification() {
76              @Override
77              protected void execute(CtClass clazz) throws Exception {
78                  clazz
79                  .addMethod(CtMethod
80                          .make(
81                              "public void print(){
82                              System.out.println(\">>: \"+newAttr); }",
83                              clazz)
```

```
84                      );
85              }
86          });
87          rc.endModification();
88
89          //?-Operatorzugriff auf neues Feld
90          rc.setValue(a, "newAttr", "cool");
91
92          //Welche Klasse? Sollte Address_V2 sein
93          System.out.println(a.getClass());
94
95          //kann einkommentiert werden, um den Punkt-Operator zu testen :)
96          // a.print(5); //geht nicht mehr :) --> No Such Method Exception
```

Listing 14: Integrationstest der jSMOC-Bibliothek

7.2 Neue Klassen zur Laufzeit

Einer der Operatoren dient zur Erzeugung neuer Klassen zur Laufzeit. Die Verwendung dieses Operators und die Modifikation der Klasse soll exemplarisch dargestellt werden.

```
1  public static void main(String[] args) {
2          //Klasse Mensch zur Laufzeit erzeugen:
3          SMOCReflectClass Mensch = SMOCReflectProgram.newClass("Mensch");
4          Mensch.beginModification();
5          Mensch.addField("private final String name;");
6          Mensch.addConstructor("public Mensch(String name){this.name = name;}");
7          Mensch.addMethod("public void sayHello(String to) {" +
8                  " System.out.println(name+\": hallo \"+to); " +
9              "}");
10         Mensch.endModification();
11
12         //Einen neuen Mensch erzeugen:
13         //Mensch otto = new Mensch("Otto");
14         SMOCObject otto = Mensch.create("(Ljava/lang/String;)V", "Otto");
15         //Methode aufrufen:
16         //otto?sayHello("Peter");
17         Mensch.invoke(otto, "sayHello(Ljava/lang/String;)V", "Peter");
18         //Otto: hallo Peter
19         //Methode sayHello entfernen
20         Mensch.beginModification();
21         Mensch.delMethod("sayHello", "(Ljava/lang/String;)V");
22         Mensch.endModification();
23
24         //Methode aufrufen (Fehler)
25         //otto?sayHello("Peter");
26         Mensch.invoke(otto, "sayHello(Ljava/lang/String;)V", "Peter");
27         //--->Fehler
28     }
```

Listing 15: Programm NewClassOp

Dazu wird im Programm NewClassOp (Listing 15) die leere Klasse Mensch erzeugt (Zeile 3). Danach werden innerhalb einer Modifikation, ein String-Feld name, ein Konstruktor, der das Feld name initialisiert und eine Methode sayHello, die unter der Verwendung des Felds name eine Begrüßung ausgibt, hinzugefügt. Danach wird eine Mensch-Instanz otto erzeugt, die auf

der Konsole, über die Methode sayHello die Begrüßung „Otto: hallo Peter“ ausgibt. Durch eine erneute Modifikation wird die Methode entfernt und anschließend nochmal aufgerufen, was zu einer Fehlermeldung auf der Konsole führt (siehe Listing 16). Zur Erzeugung der leeren Klasse Mensch, wird nur ein Aufruf benötigt. Dies ist, wie in der Operatorenanalyse gefordert, vergleichbar einfach, wie das Erzeugen eines Objekts mit new.

```
Listening for transport dt_socket at address: 8000
Listening for transport dt_socket at address: 8000
Otto: hallo Peter
Listening for transport dt_socket at address: 8000
java.lang.RuntimeException: Method sayHello(Ljava/lang/String;)V not found.
    at jsmoc.SMOCCall.invoke(SMOCCall.java:73)
    at jsmoc.reflect_impl.SMOCReflectClass_impl.
                invoke(SMOCReflectClass_impl.java:460)
    at NewClassOp.main(NewClassOp.java:41)
    ... 5 more
```

Listing 16: Ausgabe des Programms NewClassOp

7.3 Methoden zufällig hinzufügen

In dem Programm NewMethodMod (Listing 17) soll gezeigt werden, dass Modifikationen aktiver Klassen möglich sind. Dazu wird eine leere Klasse benutzt, der rekursiv mit einer gewissen Wahrscheinlichkeit (im Beispiel 80%), eine Methode hinzugefügt wird, die dann erneut eine solche Methode hinzufügt und diese dann aufruft. Zusätzlich wird am Anfang und am Ende jeder Methode eine Ausgabe gemacht, um die rekursive Verschachtelung zu visualisieren. Eine mögliche Ausgabe des Programms sieht man in Listing 18.

```
public class NewMethodMod {
    private static int counter = 0;
    private static EmptyClass ec;

    public static void addMethod(){
        System.out.println("Enter addMethod()");

        SMOCReflectClass EC_Class = SMOCReflectProgram.
                            reflectClass(EmptyClass.class);
        EC_Class.beginModification();
        counter++;
        EC_Class.addMethod(" public void randomly_"+counter+"(){" +
            "java.util.Random rand = new java.util.Random();"+
            "System.out.println(\"Enter randomly_x() in \"+this.getClass());"+
            "if (rand.nextInt(100) < 80){"+
            "   NewMethodMod.addMethod();"+
            "} else {"+
                "System.out.println(\"chain broken\");"+
            "}"+
            "System.out.println(\"Leave randomly_x() in \"+this.getClass());"+
        "}");
        EC_Class.endModification();

        //Rekursiver Aufruf
        EC_Class.invoke(ec, "randomly_"+counter+"()V");

        System.out.println("Leave addMethod()");
```

```
28      }
29
30
31
32
33      //Initialisierung
34      public static void main(String[] args) {
35          ec = new EmptyClass();
36          addMethod();
37      }
38
39  }
```

Listing 17: Programm NewMethodMod

```
1  Listening for transport dt_socket at address: 8000
2  Enter addMethod()
3  Listening for transport dt_socket at address: 8000
4  Enter randomly_x() in class EmptyClass_V1
5  Enter addMethod()
6  Listening for transport dt_socket at address: 8000
7  Enter randomly_x() in class EmptyClass_V2
8  chain broken
9  Leave randomly_x() in class EmptyClass_V2
10 Leave addMethod()
11 Leave randomly_x() in class EmptyClass_V1
12 Leave addMethod()
```

Listing 18: Eine Ausgabe des Programms NewMethodMod

7.4 Softwareupdates zur Laufzeit

Softwareupdates zur Laufzeit sind die Kernidee des JavAdaptor [16]. Mit jSMOC sind solche Updates zur Laufzeit auch möglich. Vereinfacht kann man sich vorstellen, Modifikationen lägen zum Download auf einem Server bereit. Ein Programm setzt eine update-Methode um, die diese Modifikationen herunterlädt und zur Laufzeit in das Programm einbaut. Vereinfacht wird im Programm RuntimeUpdate (Listing 19) eine Methode downloadClassUpdate(int version) bereit gestellt, die nach der Übergabe der aktuellen Version eine Modifikation zurück gibt. Eine weitere Methode updateClass(SMOCReflectClass clazz), übernimmt das Update einer übergebenen Klasse. In diesem Beispiel wird die Klasse EmptyClass aktualisiert, indem zuerst ein Feld hinzugefügt wird, und dann im zweiten Update durch ein anderes ersetzt wird. Zur Visualisierung, wird die Existenz der Felder über Introspektion geprüft und auf der Konsole ausgegeben, wie man im Listing 20 sehen kann.

```
1  public class RuntimeUpdate {
2
3      /**
4       * Methode simuliert den Download einer Modifikation
5       * @param clazz (welche Klasse)
6       * @param version (welche Version)
7       * @return neue Modifikation oder null
8       */
9      public static Modification downloadClassUpdate(SMOCReflectClass clazz,
10                                                    int version){
```

```
11          //Pruefung der Klasse clazz entfaellt...
12
13          //Simuliere den "Download"...
14          if (version <= 0){
15              return new Modification() {
16                  @Override
17                  protected void execute(CtClass clazz) throws Exception {
18                      clazz.addField(CtField.make
19                              ("public String versionUpdate;", clazz))    ;
20                  }
21              };
22          }else
23          if (version <= 1){
24              return new Modification(){
25                  @Override
26                  protected void execute(CtClass clazz) throws Exception {
27                      clazz.removeField(clazz.getField("versionUpdate"));
28                      clazz.addField(CtField.make
29                              ("public String versionUpdate2;", clazz))    ;
30                  }
31              };
32          }else{
33              return null;
34          }
35      }
36
37      /**
38       * Aktualisiert die Klasse clazz
39       * @param clazz
40       */
41      public static void updateClass(SMOCReflectClass clazz){
42          //Herunterladen der Klassenmodifikationen
43          Modification m = null;
44          if (null != (m = downloadClassUpdate(clazz, clazz.getVersion()))){
45              //Ausführen falls herunterladen erfolgreich
46              System.out.println("execute program update "+clazz.getVersion());
47              clazz.beginModification();
48              clazz.modify(m);
49              clazz.endModification();
50          }
51      }
52
53      public static void main(String[] args){
54          EmptyClass ec = new EmptyClass();
55          EmptyClass ec2 = new EmptyClass();
56          SMOCReflectClass EC_Class = SMOCReflectProgram.reflectClass(EmptyClass.class);
57
58          //Zustand ohne Update
59          System.out.println("Field versionUpdate: "+
60              EC_Class.existsField("versionUpdate"));
61          System.out.println("Field versionUpdate2: "+
62              EC_Class.existsField("versionUpdate2"));
63
64          updateClass(EC_Class);
65
66          //Zustand nach dem erstem Update
67          System.out.println("Field versionUpdate: "+
```

```
68              EC_Class.existsField("versionUpdate"));
69          System.out.println("Field versionUpdate2: "+
70              EC_Class.existsField("versionUpdate2"));
71
72          updateClass(EC_Class);
73
74          //Zustand nach dem zweitem Updatem
75          System.out.println("Field versionUpdate: "+
76              EC_Class.existsField("versionUpdate"));
77          System.out.println("Field versionUpdate2: "+
78              EC_Class.existsField("versionUpdate2"));
79      }
80  }
```

Listing 19: Programm RuntimeUpdate

```
1  Listening for transport dt_socket at address: 8000
2  Field versionUpdate: false
3  Field versionUpdate2: false
4  execute program update 0
5  Listening for transport dt_socket at address: 8000
6  Field versionUpdate: true
7  Field versionUpdate2: false
8  execute program update 1
9  Listening for transport dt_socket at address: 8000
10 Field versionUpdate: false
11 Field versionUpdate2: true
```

Listing 20: Ausgabe des Programms RuntimeUpdate

7.5 Messungen

Um jSMOC mit einem gewöhnlichen Ansatz vergleichen zu können, wurde eine manipulierbare Datentabelle mit einem objektorientierten Ansatz (Abbildung 14 - kurz: OO-Ansatz) und einem SMOC-Ansatz (Abbildung 13) umgesetzt.

Tabelle 1: Messwerte des SMOC-Ansatzes

Instanzen	Modifikationszeit ms	Zugriffszeit ms	Speicherauslastung byte
1	113	0	1268872
1000	3789	6	1798856
2000	7674	12	2505872
3000	11703	18	3265024
4000	16237	25	4209064
5000	19893	40	4985192
6000	23327	37	5771904
7000	26421	44	6720808
8000	30232	49	7631984
9000	35156	56	8469976
10000	39496	62	9374488

Tabelle 2: Messwerte des OO-Ansatzes

Instanzen	Modifikationszeit ms	Zugriffszeit ms	Speicherauslastung byte
1	0	0	1699768
1000	2	2	2086472
2000	4	1	2863696
3000	5	2	2864160
4000	9	5	4420864
5000	9	5	3639048
6000	10	6	5974088
7000	12	7	4417384
8000	15	7	7535920
9000	17	8	5200880
10000	18	10	9040912

Aus den Messungen in Tabelle 1 kann man einen durchschnittlichen Modifikationszeitanstieg von 3,938 ms pro Instanz, einen durchschnittlichen Zugriffszeitanstieg von 6,2 ms pro 1000 Instanzen und einen Speicherverbrauch von 811 Byte pro Instanz ablesen.

Beim OO-Ansatz in Tabelle 2 können keine nennenswerten Zeitanstiege gemessen werden. Es ist ersichtlich, dass der OO-Ansatz wesentlich schneller ist. Beim Speicherverbrauch ist ein Zuwachs von 734 Byte pro Instanz zu verzeichnen.

Anmerkungen:

- Die Modifikationszeit beinhaltet das Kompilieren einer Klasse und das Ersetzen der Instanzen. Die Zeit zum Kompilieren kann in der ersten Zeile abgelesen werden, da die Zeit, eine Instanz zu ersetzen, vernachlässigbar ist. Zum Kompilieren werden also etwa 100ms benötigt.
- Der Speicherzuwachs des OO-Ansatzes unterliegt größeren Schwankungen. So sinkt der Speicherverbrauch zwischen 4000 und 5000, 6000 und 7000 sowie zwischen 8000 und 9000 Instanzen unerklärbar. Vor der Messung wurde der GarbageCollector aufgerufen und 5s gewartet.
- Die jSMOC-Bibliothek macht an vielen Punkten Debugausgaben. Auch wenn diese nicht aktiviert sind, erfoglt an diesen Stellen zumindest ein statischer Methodenaufruf. Der Einfluss auf die Zeitmessung ist jedoch zu vernachlässigen.
- Der SMOC-Ansatz benötigt zwei Klassen, SMOCTable und ROW, der OO-Ansatz fünf, Table, Attribute, Row, StringAttribute und IntAttribute. Es kommt pro weiteren Typ eine Attribute-Klasse hinzu, wogegen die Klassenanzahl im SMOC-Ansatz konstant bleibt.

Ergebnisse:

- SMOC ist im gewählten Ansatz um ein vielfaches langsamer als vergleichbare OO-Ansätze.

- Der Zugriff auf in HashMap gespeicherte Felder ist wesentlich schneller als der Zugriff auf Klassenfelder über die Javareflexion.
- Der Speicherverbrauch beider Ansätze ist, wenn man die Schwankungen ignoriert, etwa gleich.
- Die Schwankungen bei der Verwendung der HashMap deuten darauf hin, dass die Speicherauslastungsmessung zumindest unpräzise ist.
- Der SMOC-Ansatz ist übersichtlicher als der OO-Ansatz.

Um die Belastbarkeitsgrenze der jSMOC-Bibliothek zu messen, die auch als Belastbarkeit des JavAdaptors herangezogen werden kann, wurde folgendes Programm geschrieben:

```
1         Row row = new Row();
2         SMOCReflectClass ROW = SMOCReflectProgram.reflectClass(Row.class);
3         int i=0;
4         while (true){
5             i++;
6             ROW.beginModification();
7             ROW.addField("public int test;");
8             ROW.endModification();
9             i++;
10            ROW.beginModification();
11            ROW.delField("test");
12            ROW.endModification();
13            System.out.println("Modification NR: "+i);
14            System.out.println("Classversion "+ROW.getVersion());
15
16            System.gc(); //Garbage Collection
17        }
```

Dieses Programm fügt der Klasse Row ein Feld hinzu und löscht dieses danach wieder. Dies geschieht so lange, bis die JVM wegen zu vieler Klassen abstürzt. Bei der Auswertung konnten folgende Beobachtungen gemacht werden:

- Die Anwendung wird trotz konstanter Anzahl von Instanzen langsamer.
- Das JVM-TI verweigert ab etwa 5000 Modifikationen hin und wieder unerklärlicherweise den Verbindungsaufbau.
- Nach 48h braucht eine Modifkation etwa 4s. Zu diesem Zeitpunkt wurden knapp über 86.000 Modifikationen durchgeführt.
- Nach den 48h wurde die Ausführung der VM beendet. Dabei verweigerte diese zuerst die Reaktion und eclipse lieferte die Fehlermeldung „Die VM konnte nicht beendet werden“. Kurz darauf war die VM jedoch trotzdem beendet.

Diese Beobachtungen lassen folgende Schlüsse zu:

- Die Anzahl der möglichen Klassen und somit die Anzahl der Modifikationen ist lediglich speichertechnisch begrenzt.
- Das JVM-TI hat mit einer hohen Anzahl von Modifikationen ungeklärte Probleme.

- Die Geschwindigkeit der Modifikationen in der jSMOC-Bibliothek ist von der Anzahl der durchgeführten Modifikationen abhängig.

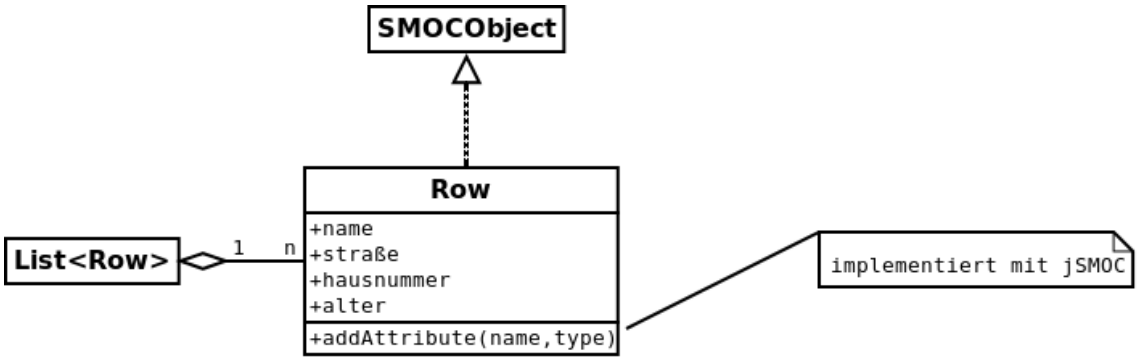


Abbildung 13: SMOC-Ansatz

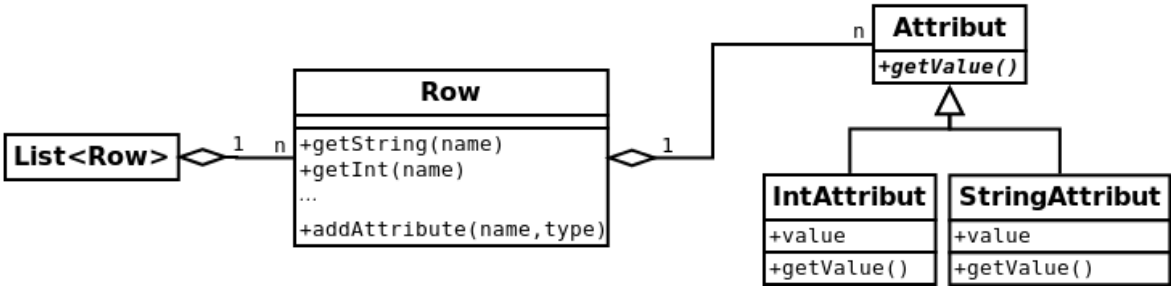


Abbildung 14: OO-Ansatz

Die Tests wurden auf einem Intel(R) Core(TM)2 Quad CPU Q9400 mit 2.66GHz (Model 23, 3072KB Cache) mit 8GB Ram unter Linux 2.6.35-32-generic #65-Ubuntu SMP Tue Jan 24 13:47:56 UTC 2012 x86_64 GNU/Linux bei deaktiviertem Auslagerungsspeicher durchgeführt.

8 Zusammenfassung und weiterführende Arbeiten

Selbstmodifizierender Code kann gefährlich sein und unstrukturierte, unübersichtliche Programme zur Folge habe. Diese Arbeit zeigt jedoch, dass der selbstmodifizierende Code zumindest in der objektorientierten Programmierung seinen Schrecken verliert, wenn man sich die Gefahren bewusst macht. Durch die Konkretisierung von Modifikationen des Arbeitsspeichers auf exakte Operatoren sind selbstmodifizierende Programme in Java umsetzbar geworden. Man findet sich wieder in der dynamischen Softwareentwicklung. Starre Klassenkonstrukte haben dazu geführt, dass große und großartige Programme entwickelt wurden. Durch Bytecode-Manipulation, dynamische Softwareupdates zur Laufzeit (JavAdaptor) und JIT-Compiler wird klar, dass diese starren Schranken auch ein Hindernis werden können. Programme, die sich modifizieren, könnten Versionschaos, Updates und Bugfixes ablösen und Programme ermöglichen, die ohne Neustart Fehler ihre korrigieren. Die Flexibilität, die bisher nur Skriptsprachen boten, könnte nun mit der Geschwindigkeit und Typsicherheit von kompilierten Sprachen kombiniert werden.

Diese Arbeit liefert mit jSMOC eine Bibliothek, welche die Modifikationsoperatoren in Java so genau beschreibt und implementiert, wie sie sich die dynamische Programmierung erwünscht. Klassen können nahezu beliebig verändert und umstrukturiert werden. Die Geschwindigkeit lässt noch zu wünschen übrig, aber die Vermutung liegt nahe: Werden die Operatoren als konkrete Operatoren einer VM umgesetzt und in einer neuen Programmiersprache vereint, wird sich dies zumindest verbessern. Das Evaluationsbeispiel zeigt, dass Klassenkonstrukte mit SMOC einfacher werden können. Neue Entwicklungsmuster können entstehen und die Entwicklung selbstadaptiver Systeme vereinfachen.

Die SMOC-AG, geführt von Dr. Frank Padberg, hat bereits festgestellt, dass die Entwicklung selbstmodifizierender Software nicht nur eine Frage der Programmiersprache ist. Man braucht erweiterte Entwicklungstools[28], bessere Debugger und neue Diagrammtypen, um Selbstmodifikationen planen zu können. Statische UML-Klassendiagramme können Dynamik von Modifikationen nicht erfassen oder beschreiben. Programme brauchen auch ein Selbstbild, wie die Bachelorarbeit[29] von Alexander Cullmann zeigt. So reiht sich auch diese Arbeit in die ersten Versuche der SMOC-AG, selbstmodifizierbare Software zu erforschen ein.

Ein weiteres ungelöstes Problem sind die aktiven Methoden. Es bleibt auch in dieser Arbeit ungeklärt, wie eine zurzeit aktive Methode ersetzt oder verändert werden kann. Das Problem der aktiven Methoden bezieht sich jedoch nicht nur auf die aktuell aktive Methode. Eine Methode wird immer von einer anderen Methode aufgerufen, die dadurch mit aktiv ist. Diese Kette aktiver Methoden kann nicht sinnvoll verändert werden. Dieses Problem erkannten auch die Entwickler von JavAdaptor [16]. Da in jSMOC Klassen nicht entfernt werden, bleiben aktive Methoden immer in der alten Klassenversion erhalten, bis diese neu aufgerufen werden.

Neben den bereits genannten offenen Probleme der selbstmodifizierenden Software, können auch folgende weiterführende Arbeiten genannt werden, die sich auf diese Bachelorarbeit beziehen:

- Umsetzung der jSMOC-Bibliothek mit Nebenläufigkeit
- Reduzierung der Einschränkungen der jSMOC-Bibliothek

[28] Eine Eclipse-basierte Entwicklungsplattform für selbstmodifizierende Programme - Masterarbeit an der Universität des Saarlandes von Sascha Maaß

[29] Selbstbilder in selbstmodifizierender Software - Alexander Cullmann - Bachelorarbeit an der Universität des Saarlandes

- Umsetzung der SMOC-Operatoren in einer eigenen Programmiersprache, welche diese konkret als Operatoren realisiert. Dazu könnte ähnlich wie in Java ein VM Weg gewählt werden.

Danksagung

Mit diesen Zeilen möchte ich nun noch allen danken, die mir bei dieser Arbeit zur Seite gestanden sind. An erster Stelle möchte ich hier meinen Betreuer Dr. Frank Padberg nennen, der mich stets forderte, mit seinem Wissen zur Seite stand und fast wöchentlich für Treffen der SMOC-AG und zur Betreuung seiner Studenten den Weg von Karlsruhe nach Saarbrücken auf sich nahm.

Außerdem möchte ich auch meinen Eltern Dr. Hans-Jürgen Bürckert und Silvia Bürckert danken, die mein Studium finanzieren. Ansonsten möchte ich noch Frau Dr. Mira Spassova, meinem Mitbewohner Sascha Zickenrott und meinem Vater für die ausführlichen Rechtschreibprüfungen danken und möchte jedem noch fündigen Leser anbieten, die gefundenen verbleibenden Rechtschreibfehler doch gerne für sich zu behalten.

Zu guter Letzt danke ich meiner Freundin, meinen Freunden, meinen Mitstudenten, meinem Mitbewohner und sonstigen Menschen, die mich über die ganze Zeit der Ausarbeitung und den letzten drei Monaten bis zur Abgabe hin ertragen mussten. Mögen sie ihren Frieden wieder finden.

Saarbrücken 14. März 2012

Christian Felix Bürckert

9 Anhang

Im Anhang werden Einschränkungen, eine Installationsanleitung, bekannte Fehlermeldungen sowie einige Abbildungen aufgeführt.

9.1 Einschränkungen von jSMOC

Die Entwicklung mit der jSMOC-Bibliothek stellt folgende Einschnitte in die elementaren Bestandteile der Programmiersprache Java:

1. Es sollte nicht mehr als ein Thread verwendet werden. Falls doch, so muss zumindest gesichert werden, dass immer nur ein Thread eine Modifikation durchführt.
2. Modifizierbare Klassen dürfen nicht generisch getypt sein.
3. Die Bibliothek erlaubt keine inneren Klassen.
4. Modifizierbare Klassen müssen einen leeren Konstruktor anbieten. Welche Feldbelegung darin festgelegt wird, spielt keine Rolle.
5. Modifizierbare Klassen sollten keine statischen Elemente besitzten. Falls doch, sollte man sich bewusst sein, dass diese beim Erstellen der Klassenkopie ebenfalls kopiert werden. Es ist zu prüfen ob alle Klassenkopien auf die angedachten statischen Elemente zugreifen und nicht etwa teilweise auf die der Vorversion.
6. Bibliotheken, die ebenfalls einen präparierten ClassLoader benötigen, können unvorhersehbare Reaktionen zeigen.
7. Bei der Verwendung von Frameworks, ist auf die korrekte Initialisierung der jSMOC-Bibliothek zu achten.
8. Modifizierbare JFrame-Klassen wurden nicht getestet.
9. Neue Pakete dürfen nicht mit jsmoc, smoc, sun oder java beginnen, da diese von der Codetransformation ausgeschlossen sind.
10. Methoden und Konstruktoren sollten nicht mehr als 20 Parameter haben. Falls doch müssen die Klassen SMOCCall und SMOCCreate erweitert werden.
11. Analyse- und Debuggerwerkzeuge, die eine Verbindung über das JVM-TI herstellen, können nur eingeschränkt verwendet werden, da die Verbindung durch die Modifikationen belegt wird.

9.2 Schrittweise Installation

Im Folgenden werden die Schritte erklärt, mit denen man eine Umgebung für ein selbstmodifizierendes Programm in Eclipse [30] erstellt.

1. Herunterladen und Installieren einer aktuellen Eclipse-Version: Entwickelt und getestet wurde mit Version: 3.7.0 Build Id: I20110613-1736.

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

2. Erstellen eines leeren Javaprojekts. Verwendet wurde für die Entwicklung und Tests:
   java version ”1.6.0_20”
   OpenJDK Runtime Environment (IcedTea6 1.9.13) (6b20-1.9.13-0ubuntu1 10.10.1)
   OpenJDK 64-Bit Server VM (build 19.0-b09, mixed mode)
3. Erstellen eines „lib“-Ordners im Projektverzeichnis.
4. Herunterladen der javassist.jar ins „lib“-Verzeichnis.
   Verwendet wurde die Version 3.16.1.GA
5. Herunterladen des Quellcodes von ASM ins „lib“-Verzeichnis.
   Verwendet wurde die Version 3.3.1
6. Herunterladen des Quellcodes von jSMOC ins Stammverzeichnis
7. Einbinden von javassist.jar in den Build-Path
8. Einbinden des Quellcodes von ASM als Source-Ordner
9. Einbinden des Quellcodes von jSMOC als Source-Ordner
10. Erstellen eines Ordners „bytecode“ zur Ausgabe der Tracelogs im Wurzelverzeichnis
11. Erstellen eines Source-Ordners für das SMOC-Programm
12. Erstellen einer RUN-Konfiguration für die Klasse jsmoc.init.SMOCVM mit den Programmargumenten *–run <Start Klasse> [Programmargumente]* und VM-Argumenten *-noverify -Xbatch -Xint -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=8000*
13. Zum Starten des Programms die eingerichtete RUN-Konfiguration ausführen.

9.3 Programmargumente für jSMOC

Um Fehler in SMOC-Programmen zu finden, können die jSMOC-Ausgaben aktiviert werden. Dazu können folgende Parameter vor –run eingefügt werden:

1. –debug: Ausgaben der jSMOC-Bibliothek auf der Fehlerkonsole.
2. –test: Startet vor der Ausführung des Programms die jSMOC-Tests
3. –tracelog: visualisiert die Bytecodetransformationen im Verzeichnis bytecode. Vor und nach der Transformation wird der Bytecode lesbar ausgegeben. Mit Vergleichsprogrammen kann diese Transformation schön visualisiert werden. (In Eclipse kann die Compare-With-Each-Other-Funktion genutzt werden.)
4. –musthave [FILTER1,FILTER2,...]: von –debug erzeugte Ausgaben werden nur ausgegeben, sofern eins der Filterwörter in der Ausgabe vorkommt.
5. –mustnothave [FILTER1,FILTER2,...]: von –debug erzeugte Ausgaben werden nicht ausgegeben, sofern eines der Filterwörter darin vorkommt.

9.4 Bekannte Probleme

jSMOC liefert bei jeder Modifikation die Ausgabe „Listening for transport dt_socket at address: 8000“. Diese kommt aus dem JVM-TI-Connector und kann nicht unterbunden werden. Die Ausgabe bedeutet lediglich, dass eine Verbindung zum JVM-TI aufgebaut wurde.

Führt man viele Modifikationen in schneller Abfolge durch, so schlägt die Verbindung zum JVM-TI manchmal fehl. Die Ursache dafür ist unbekannt, die jSMOC-Bibliothek wird jedoch so lange versuchen, die Verbindung aufzubauen, bis dies gelingt. Bei jedem Fehlschlag wird die Warnung „Connection to VM could not be established ... “ ausgegeben. Zwischen zwei Verbindungsversuchen wird eine Sekunde gewartet, um der JVM Zeit zu geben, ihre Last abzuarbeiten.

Ein häufiger Fehler ist es, SMOC-Programme nicht über die SMOCVM zu starten. Die jSMOC-Bibliothek erkennt dies, sobald SMOCReflectProgramm das erste mal verwendet wird, und gibt die Fehlermeldung „Programs using the jSMOC-Library should be started using the SMOCVM's main method!“ aus. Sollte dieser Fehler auftreten, so ist die RUN-Konfiguration zu prüfen und entsprechend der Anleitung im Kapitel 9.2 zu konfigurieren.

9.5 Abbildungen

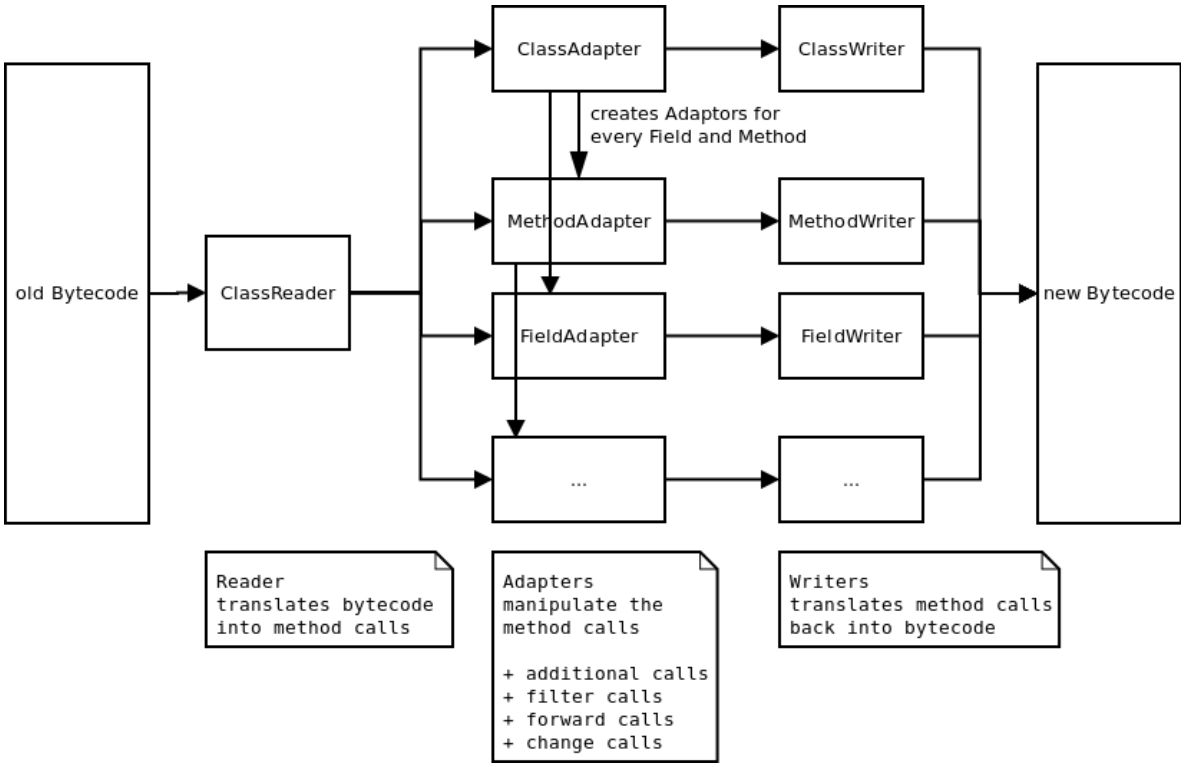


Abbildung 15: ASM-Modifikationsschema

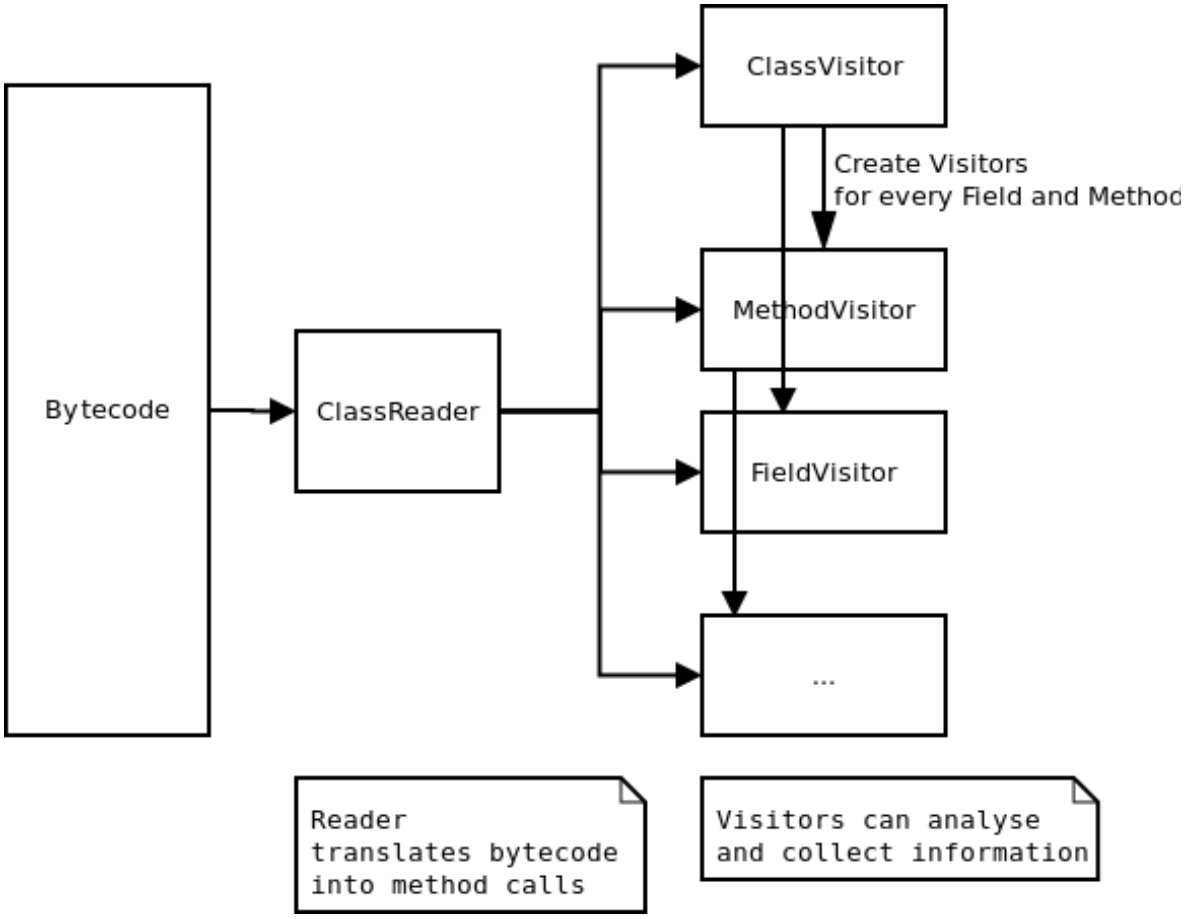


Abbildung 16: ASM-Analyseschema

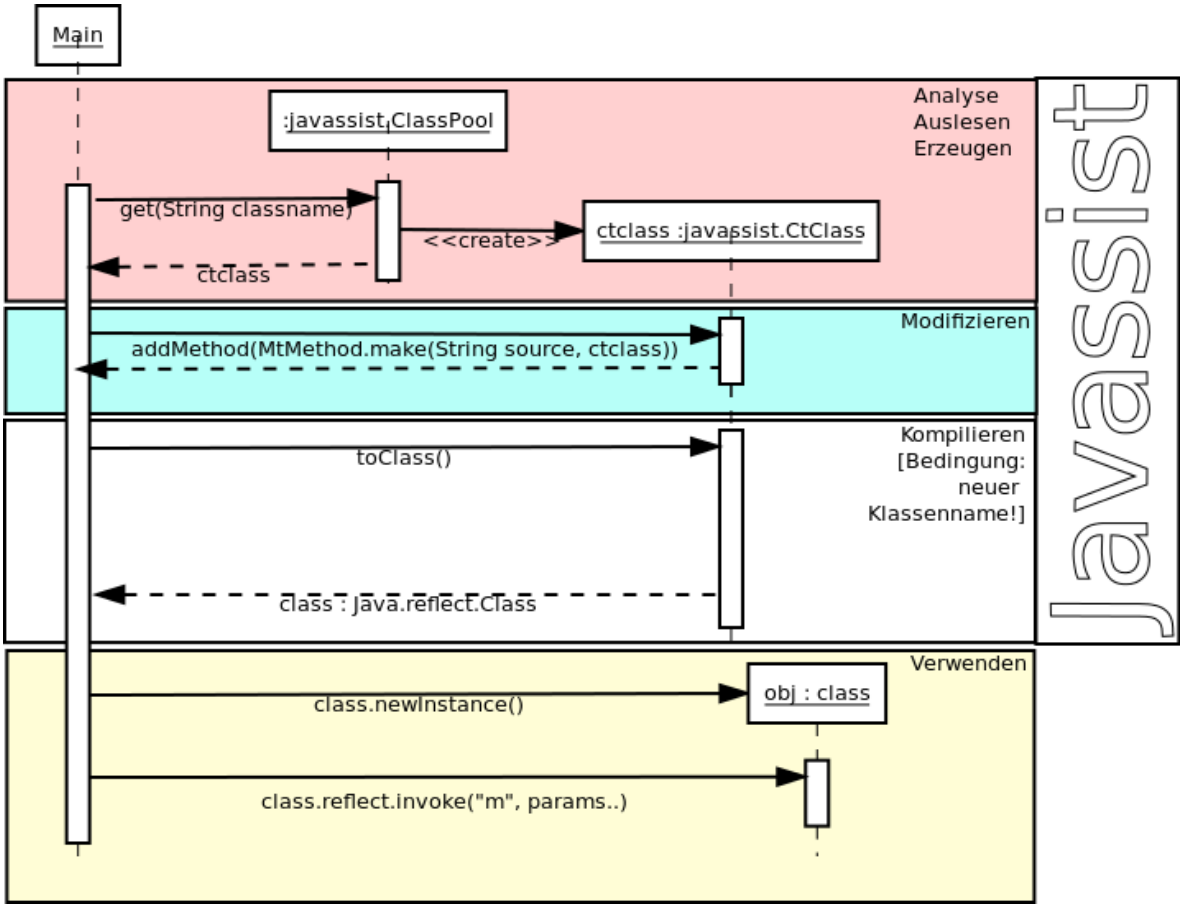


Abbildung 17: Klassenmanipulation mit Javassist

Literatur

[1] Péter Ször (Architect) and Peter Ferrie (Software Engineer). "hunting for metamorphic", 2002 `http://enterprisesecurity.symantec.com/PDF/metamorphic.pdf`.

[2] Godmar Back. DataScript - A Specification and Scripting Language for Binary Data. In Generative Programming and Component Engineering, pages 66–77. Springer, 2002.

[3] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. SIGPLAN Not., 39(10):331–344, October 2004.

[4] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In Adaptable and extensible component systems, 2002.

[5] Shigeru Chiba. Javassist - A Reflection-based Programming Wizard for Java. In International Business Machines Corp, page http://www.javassist.org, 1998.

[6] Shigeru Chiba and Muga Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In 2nd International coference on Generative Programming and Component Engineering (GPCE '03), volume 2830 of Springer Lecture Notes in Computer Science, pages 364–376. Springer-Verlag, 2003.

[7] Tal Cohen and Joseph (Yossi) Gil. Three approaches to object evolution. In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09, pages 57–66, New York, NY, USA, 2009. ACM.

[8] Pascal Costanza. Dynamic Object Replacement and Implementation-Only Classes. In 6th International Workshop on Component-Oriented Programming (WCOP 2001) at ECOOP 2001, 2001.

[9] A. Goldberg and D. Robson. Smalltalk-80, the Language and its Implementation. Addison-Wesley, Reading, Massachusetts, 1983.

[10] Andrew Hunt and David Thomas. Pragmatic Unit Testing: In Java with Junit. Pragmatic Bookshelf, Raleigh, NC, 2003.

[11] Gregor Kiczales and Jim Des Rivieres. The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, USA, 1991.

[12] Edward J. Klimas, Suzanne Skublics, and David A. Thomas. Smalltalk with style. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[13] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - A Framework for Load-Time Transformation of Java Class Files. pages 100–110, 2001.

[14] Dahm M. Byte Code Engineering with the JavaClass API. Technical report, Institut für Informatik, Freie Universität Berlin B-17-98, January 1999.

[15] Pattie Maes. Concepts and experiments in computational reflection. SIGPLAN Not., 22:147–155, December 1987.

[16] Mario Pukall, Alexander Grebhahn, Reimar Schröter, Christian Kästner, Walter Cazzola, and Sebastian Götz. JavAdaptor: Unrestricted Dynamic Software Updates for Java. In Proceedings of the 33rd International Conference on Software Engineering, pages 989–991, May 2011.

[17] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Flexible Dynamic Software Updates of Java Applications: Tool Support and Case Study. Technical Report 04, School of Computer Science, University of Magdeburg, April 2011.

[18] Eric Tanter, Marc Segura-devillechaise, Jacques Noye, and Jose Piquer. Altering Java Semantics via Bytecode Manipulation, 2002.

[19] Michiaki Tatsubori, Shigeru Chiba, Marc olivier Killijian, and Kozo Itano. OpenJava: A Class-based Macro System for Java. In Reflection and Software Engineering, pages 117–133. Springer-Verlag, 2000.