Fachgebiet Software-Architekturen
Hasso-Plattner-Institut
Universität Potsdam

Masterarbeit

Modularisierung virtueller Maschinen

Implementierung und Evaluierung einer Architekturbeschreibungssprache

Stefan Marr
30.09.2008

Betreuung
Dr. Michael Haupt
Prof. Dr. Robert Hirschfeld

DANKSAGUNG

Ich möchte mich bei allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben. Ganz besonders möchte ich mich bei Dr. Michael Haupt und Prof. Dr. Robert Hirschfeld für ihre Betreuung und ihre Unterstützung bedanken, die mir sehr dabei geholfen hat, dieser Arbeit zu einem erfolgreichen Abschluss zu bringen.

Darüber hinaus möchte ich mich bei den Studenten bedanken, die an den Vorlesungen *Virtuelle Maschinen* in den Jahren 2007 und 2008 teilgenommen und mir ihre Erweiterungen der CSOM-VM für diese Arbeit zur Verfügung gestellt und somit die technische Grundlage für meine Untersuchungen gelegt haben.

Bei Michael Perscheid bedanke ich mich sehr für die Betreuung der Kontrollgruppe des Experiments. Mein Dank geht auch an Arne Bergmann, Juri Engel, Frank Feinbube, Gregor Gabrysiak, Jan Klimke, Murat Knecht, Thomas Kowark, Daniel Richter, Michael Seegers und Robert Wierschke für ihre Mitwirkung als Probanden am Experiment.

Abschließende möchte ich mich bei Tobias Pape und Tobias Vogel für das Korrekturlesen dieser Arbeit und ihre zahlreichen Anregungen und Verbesserungsvorschläge bedanken.

Abstract

Today, high-level language virtual machines are custom-made special purpose systems optimized for specific domains like embedded devices. Their implementation suffers from high complexity, strongly intertwined subsystems and little maintainability. This is caused by insufficient modularization techniques failing to represent the system architecture in the code. To overcome this situation, the *Virtual Machine Architecture Definition Language* (VMADL) has been proposed. In this work, the language features of VMADL are investigated and a VMADL compiler is implemented to conduct a case study on an existing Smalltalk VM written in C named CSOM.

The aim of this case study is to test the language on CSOM and to modularize it in a way that it is possible to build a product family of VMs out of it, to be able to benefit from a product-line approach for customizing VMs. In a first step, CSOM is reverse engineered and several feature implementations are analyzed for their specific requirements on modularization. Based on this examination, a CSOM specific feature-oriented class definition language is added to VMADL to allow a completely modularized implementation of all analyzed features.

Afterwards, the case study is evaluated using several metrics to prove that the code overhead caused by VMADL is minimal and necessary for the architecture and modularization. Furthermore, it is shown that there is no negative impact on the runtime performance of the system. To prove the positive influence of VMADL on architectural comprehension and maintainability, an experiment with students is designed and conducted. Eventually, this work demonstrates how VMADL can be used to explicitly express the architecture of a VM in the code, improve modularization, and add the necessary variability to build a VM product family.

Zusammenfassung

Virtuelle Maschinen für Hochsprachen werden meist als Spezialanfertigungen für einen bestimmten Zweck entwickelt. Sie sind jeweils für genau ein Anwendungsgebiet z. B. den Einsatz im Bereich eingebetteter Systeme optimiert. Ihre Implementierungen sind aufgrund der hohen Komplexität und engen Verknüpfungen der einzelnen Systemteile nur unzureichend modularisiert. Daher ist die Architektur solcher Systeme nur schlecht erkennbar und der Wartungsaufwand für sie entsprechend stark erhöht. Um statt einzelnen Spezialanfertigungen im Bereich der virtuellen Maschinen auf Techniken für die Entwicklung von Produktfamilien zurückgreifen zu können, wurde die Virtual Machine Architecture Definition Language (VMADL) vorgeschlagen. In dieser Arbeit werden ihre Sprachmittel untersucht, ein Compiler für die Verwendung von VMADL in Verbindung mit C implementiert und die Sprache in einer Fallstudie auf eine CSOM genannte, bereits vorhandene Smalltalk-VM angewandt.

Das Ziel der Fallstudie ist es, die Sprachmittel von VMADL zu testen und CSOM so zu modularisieren, dass es möglich ist, eine Produktfamilie von VMs zu entwickeln. Dazu wird im ersten Schritt ein Reverse-Engineering für CSOM durchgeführt und verschiedene Erweiterungen auf ihre jeweils spezifischen Anforderungen an Modularisierungsmechanismen untersucht. Daraus ergibt sich der Bedarf für einen featureorientierten Ansatz, der mit der für CSOM entworfenen Klassenbeschreibungssprache ClassDL ebenfalls vorgestellt wird. Zum Abschluss werden die aus den einzelnen Erweiterungen extrahierten Features und ihre Modularisierung mit VMADL betrachtet.

Zur Evaluierung der Fallstudie werden verschieden Metriken genutzt, um zu zeigen, dass der zusätzliche Quelltext durch VMADL minimal ist und allein durch die für das Architekturverständnis und die Modularität benötigten Konstrukte entsteht. Die Untersuchungen zeigen zudem, dass kein negativer Einfluss auf die Performance vorhanden ist. Zusätzlich wird ein Experiment entworfen und durchgeführt, das zeigen soll, ob Probanden, bei Verwendung von VMADL das gestellte Problem schneller lösen können und dabei die bestehende Architektur weniger beeinflussen, als dies ohne VMADL der Fall wäre. Letztendlich zeigt diese Arbeit, dass mit VMADL die Architektur einer virtuellen Maschine durch höhere Modularität verbessert und durch die zusätzliche Flexibilität als Produktfamilie gestaltet werden kann.

ERKLÄRUNG

Hiermit erkläre ich, dass ich zum Verfassen dieser Arbeit keine anderen Quellen und Hilfsmittel als die angegebenen verwendet habe.

Potsdam, den 22.09.2008

Stefan Marr

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

QUELLTEXTVERZEICHNIS

TABELLENVERZEICHNIS

1 Einleitung

1.1 Zielsetzung der Arbeit

Virtuelle Maschinen (VMs) gehören mit zu den komplexesten Softwaresystemen, da sie als Ausführungsplattform für Anwendungssoftware vielen verschiedenen Anforderungen genügen müssen. Diese Arbeit soll dabei helfen, diese Komplexität leichter handhabbar zu machen. Von Michael Haupt et al. wurde VMADL, die *Virtual Machine Architecture Definition Language* vorgeschlagen [HAT+08]. Diese Architekturbeschreibungssprache wurde mit dem Ziel entworfen, die Systemarchitektur im Quelltext sichtbarer zu machen. Explizite Sprachmittel für die Abbildung der Architektur sollen den Entwicklern die Arbeit mit dem System erleichtern, indem sie in die Lage versetzt werden, Abhängigkeiten und Interaktionen zwischen Teilsystemen leichter zu erkennen und somit die Zusammenhänge im System detaillierter zu erfassen.

Das Ziel dieser Arbeit ist es, die vorgeschlagenen Sprachmittel zu untersuchen und auf CSOM, eine virtuelle Maschine für Smalltalk, anzuwenden. Im Rahmen dieser Untersuchungen sollen die Sprachmittel entsprechend erweitert bzw. angepasst werden, um den erwünschten höheren Grad an Modularität zu erreichen. Darüber hinaus sollen die eingesetzten Techniken der aspektorientierten Programmierung (AOP) und VMADL einer Untersuchung unterzogen werden, um im Anschluss bewerten zu können, ob sie geeignete Mittel sind, um das Ziel eines höheren Grads an Modularität und einer besseren Verständlichkeit des Systems erreichen zu können.

Ein weiteres Ziel ist es, mit der gewonnenen Modularität eine CSOM-Produktfamilie zu schaffen, mit der es möglich wird eine Produktlinie aufzubauen, deren Exemplare jeweils für die speziellen Bedürfnisse eines konkreten Einsatzgebietes wie bspw. eingebettete Systeme optimiert sind. Einerseits soll dies die Möglichkeiten aufzeigen, die eine vollständige Modularisierung in Bezug auf erhöhte Variabilität bietet. Andererseits lassen sich so die Grenzen der Modularisierung und eventuell negative Auswirkungen auf andere Eigenschaften der virtuellen Maschine feststellen, die es zu vermeiden gilt.

Die Ergebnisse werden anschließend untersucht und bewertet. Neben der Verwendung von Metriken für Quelltext- und Performanceuntersuchungen soll außerdem mithilfe eines Experiments geprüft werden, ob die Verwendung von VMADL Auswirkungen auf die Wahrnehmung der Architektur durch Entwickler hat und ihnen hilft mit diesen zusätzlichen Informationen bessere Ergebnisse bei der Arbeit an der virtuellen Maschine zu erreichen.

Zur Durchführung dieser Untersuchungen, sollen im Rahmen dieser Arbeit die notwendigen Werkzeuge implementiert werden. Dazu gehört ein Compiler für VMADL zur Verwendung mit C und AspectC++ sowie die technische Infrastruktur, um CSOM in Form einer Produktfamilie erstellen und über ein Konfigurationswerkzeug die gewünschte Modulkombination auswählen zu können.

1.2 Grundlage dieser Arbeit

VMADL bildet die Grundlage dieser Arbeit. Vorgeschlagen wurde sie mit dem Ziel, die starken Modulverflechtungen zu reduzieren, indem virtuelle Maschinen als Komposition von Service-Modulen realisiert werden [HAT+08]. Service-Module sind Module auf der Architekturebene, die aus mehreren Implementierungsmodulen bestehen können.

Mithilfe bidirektionaler Schnittstellen sollen sie es ermöglichen, besondere Ereignisse anderen Modulen in geeigneter Form an der Schnittstelle bereitzustellen, um so die Verflechtung über aspektorientierte Programmierung reduzieren zu können.

Neben dieser Entflechtung der Module soll es mit der vorgeschlagenen Sprache möglich sein, die Modulbeziehungen auf der Architekturebene direkt im Quelltext anzugeben, um diese für die Entwickler explizit zu machen. In Sprachen wie C, Java oder Smalltalk ist die Systemarchitektur allein durch die Beziehungen bestimmt, die sich durch Funktionsaufrufe und Abhängigkeiten im Quelltext ergeben. Diese können jedoch nur mit hohem Aufwand aus dem Quelltext extrahiert werden und sind somit für die Entwickler keine optimale Quelle, um sich einen Eindruck vom Zusammenspiel der Module im Gesamtsystem zu verschaffen.

Die Idee, Modulschnittstellen mit den Mitteln der AOP zu erweitern, wurde mit *Open Modules* [Ald05] und *Crosscutting Interfaces* (XPI) [GSS+06] bereits früher vorgestellt. Im Unterschied zu diesen Ansätzen sind die mit VMADL vorgeschlagenen Service-Module jedoch nicht länger auf Implementierungs- sondern auf der Architekturebene angeordnet und erzwingen keine komplette Modularisierung mithilfe von aspektorientierter Programmierung. Stattdessen wird es möglich die AOP zur Beschreibung von Modulinteraktionen zwischen für die Architektur relevanten Modulen zu verwenden und sie somit pragmatisch einzusetzen, um die Verständlichkeit des Systems zu erhöhen.

1.3 Beitrag dieser Arbeit

Der Hauptbeitrag dieser Arbeit besteht in der Weiterentwicklung und Evaluierung von VMADL. Es wird die ursprüngliche Idee der Sprache in eine Form überführt, die mithilfe von Architekturmodulen und Modulinteraktionen auf der Architekturebene die Modularisierung und Realisierung von VMs in Form von Produktfamilien erlaubt.

Es wird gezeigt, wie mit den Techniken der aspekt- und featureorientierten Programmierung eine Architekturbeschreibungssprache geschaffen werden kann, die Teil der Implementierung ist und damit die Probleme von anderen Architekturbeschreibungssprachen vermeidet, die als zusätzliche Systembeschreibung auf einer übergeordneten Ebene, unabhängig von der Implementierung sind. Zusätzlich werden die Ergebnisse evaluiert und die Vor- und Nachteile analysiert. Für die Evaluierung wurde ein VMADL-Compiler implementiert und VMADL auf eine existierende VM angewendet.

Ein weiterer Beitrag dieser Arbeit ist der Entwurf und die Durchführung eines Experiments mit Studenten, um die Auswirkungen von VMADL auf die Architekturwahrnehmung und die Wartbarkeit einer VM zu untersuchen. Diese Arbeit ist damit ein weiteres Beispiel dafür, welche Aspekte für eine solche Untersuchung zu beachten sind. Es wird auf die zu berücksichtigen Probleme und möglichen Verfälschungen eingegangen, die für die Gewinnung aussagekräftiger Daten nötig sind.

1.4 Struktur der Arbeit

Zur Einführung wird im Abschnitt 2 der Anwendungsbereich dieser Arbeit, die Implementierung virtuellen Maschinen, vorgestellt. Darauf folgt eine Betrachtung der besonderen Komplexität in virtuellen Maschinen, um den Bedarf für eine spezielle Architekturbeschreibungssprache (*architecture description language*, ADL) zu motivieren, die Entwicklern dabei helfen soll, die dargestellten Problematiken beim Entwurf und der Wartung von VMs besser handhaben zu können. Da es keinen allgemeinen Konsens darüber gibt, welche Eigenschaften wesentlich für eine ADL sind,

werden in Abschnitt 2.4 kurz die beiden Hauptrichtungen betrachtet und die für die Beschreibung von Architekturen virtueller Maschinen relevanten Elemente herausgearbeitet.

Im Anschluss daran folgt in Abschnitt 3 die Vorstellung der *Virtual Machine Architecture Definition Language*. Nach einer kurzen Einführung in die Grundkonzepte wird der Sprachumfang anhand eines Beispiels vorgestellt. Abschnitt 3.3 beginnt mit der Erörterung des ersten VMADL-Compiler-Prototyps für die von Haupt et al. [HAT+08] durchgeführten Untersuchungen. Darauf aufbauend wird der neu implementierte VMADL-Compiler, der den vollen Sprachumfang unterstützt, in Abschnitt 3.4 erläutert.

Nach der Vorstellung der VMADL wird in Abschnitt 4 ihre Anwendung auf CSOM vorgestellt. Nach einer kurzen Einführung und Vorstellung von CSOM werden die Ziele dieser Fallstudie definiert. Daran schließt sich in Abschnitt 4.3 die Erläuterung der Reverse-Engineering-Phase an. Hier wird zuerst das Vorgehen zur Analyse des CSOM-Quelltexts vorgestellt, um im Anschluss daran die gewonnenen Erkenntnisse in Form der extrahierten Architektur sowie der besonderen Merkmale der untersuchten Feature-Implementierung darzustellen. Im Abschnitt 4.4 werden die Überarbeitungsschritte von der ursprünglichen CSOM-Implementierung zur Anwendung von VMADL beschrieben, um darauf aufbauend in Abschnitt 4.5 die eigentliche Anwendung von VMADL darzustellen. Neben der Vorgehensweise werden in diesem Abschnitt die speziell für CSOM entworfene featureorientierte Klassendefinitionssprache ClassDL vorgestellt und die in der mit VMADL implementierten CSOM vorgefundenen Modulinteraktionen analysiert, um die Auswirkungen von VMADL einschätzen zu können.

Eine Evaluierung dieser Ergebnisse mithilfe objektiver Kriterien wird in Abschnitt 5 vorgenommen. Zu den verwendeten Metriken gehören statische Größen, die Aufschluss über die Modularisierung geben und Benchmarks, die Einflüsse auf die Laufzeiteigenschaften der VM aufzeigen sollen, die durch die Verwendung von aspektorientierter Programmierung entstehen. Abschnitt 5.3 dokumentiert den Entwurf, die Durchführung und die Ergebnisse eines Experiments mit Studenten, das die tatsächlichen Auswirkungen auf die Wahrnehmung der Architektur und die Wartbarkeit des Systems aufzeigen soll.

Im Anschluss daran werden in Abschnitt 6 verschiedene Ansätze betrachtet, die sich ebenfalls mit Modularisierung, Architektur und Komponentensystemen beschäftigen.

Eine Zusammenfassung der Ergebnisse und ein Ausblick auf mögliche Folgearbeiten, die auf den hier gewonnenen Erkenntnissen aufbauen können, schließt diese Arbeit ab.

2 Implementierung virtueller Maschinen

2.1 Abstraktionsgewinn durch virtuelle Maschinen

Bei der Betrachtung von virtuellen Maschinen, kurz VMs genannt, kann man zwei wesentliche Typen unterscheiden. Von Smith und Nair [SN05] werden sie in *System-* und *Prozess-VMs* eingeteilt. Unter System- oder auch Hardware-VMs versteht man Softwaresysteme, die einem Gastsystem eine komplette Systemumgebung bereitstellen. Dabei werden typischerweise wesentliche Hardware-Komponenten nachgebildet und es wird möglich, gleichzeitig mehrere Betriebssysteme auf derselben Hardware auszuführen.

Mit dieser Art von virtuellen Maschinen soll sich diese Arbeit jedoch nicht beschäftigen. Betrachtet werden stattdessen die *Prozess-* oder *Applikations-* bzw. *High-Level-Language-VMs*. Diese stellen einem Anwendungsprogramm eine Ausführungsumgebung bereit, die unabhängig vom darunterliegenden System ist und auf allen unterstützten Plattformen der Anwendung dieselben wohldefinierten Schnittstellen bietet.

Virtuelle Maschinen für Hochsprachen sind heute ein Grundbaustein in der Softwareentwicklung, da sie Entwicklern im Allgemeinen eine höhere Abstraktion und Mächtigkeit durch Mechanismen wie z. B. automatische Speicherverwaltung, Metaprogrammierung und Sicherheitsfunktionalität bieten. Sie erlauben somit eine effizientere Entwicklung von Anwendungsprogrammen, als dies mit klassischen systemnahen Sprachen wie C der Fall ist. Gleichzeitig versprechen VMs eine höhere Portabilität von Anwendungsprogrammen, da auf allen unterstützten Plattformen einheitliche Schnittstellen und damit gleiche Ausführungsumgebungen bereitstehen und somit Programme in derselben Art und Weise ausgeführt werden können. Dies verringert einerseits den Aufwand, Anwendungsprogramme auf verschiedenen Plattformen ausführen zu können und bietet andererseits die Möglichkeit, über ein Zwischenformat wie z. B. Smalltalk- oder Java-Bytecode den Austausch von Programmen über Plattformgrenzen hinweg in einer ausführbaren Form zu ermöglichen.

Die Implementierung dieser Abstraktionen ist, wie im folgenden Abschnitt dargelegt wird, ein komplexes Unterfangen. Softwaresysteme, die solche Prozess-VMs realisieren erreichen eine mit Betriebssystemen vergleichbare Komplexität. Im Unterschied zu diesen bieten sie jedoch im Allgemeinen eine andere Funktionalität, die durch die Eigenschaften der implementierten Sprachen bestimmt werden, und bilden eine zusätzliche Schicht über dem Betriebssystem. Ein Teil der durch VMs bereitgestellten Funktionalität überschneidet sich jedoch mit den Aufgaben von Betriebssystemen, wodurch es zu Redundanz kommt. Ein Versuch diese Redundanz zu vermeiden stellt z. B. *LiquidVM* [BEA08b] dar. Bei dieser angepassten JRockit Java-VM handelt es sich um ein System, das direkt ohne zusätzliches Betriebssystem auf einem Hypervisor bzw. Typ 1 *Virtual Machine Monitor* [Gol72] zum Einsatz kommt. Dadurch werden redundante Elemente eingespart, aber es müssen auch typische Betriebssystemfunktionen, wie z. B. ein Dateisystem, selbst implementiert werden.

Bei Microsofts Projekt *Singularity* [HL07] wurde dieses Problem von der anderen Richtung angegangen. Anstatt eine VM um Betriebssystemfunktionalität zu erweitern, wurde das komplette Betriebssystem in einer Hochsprache implementiert und die Konzepte dieser Sprache [FAH+06] konsequent für das Betriebssystem und auch auf die Anwendungsprogramme eingesetzt. Somit ist das Betriebssystem gleichzeitig eine

Prozess-VM für eine Hochsprache. Bei diesen Ansätzen wird jedoch nicht das Problem der Komplexität in virtuellen Maschinen selbst gelöst.

2.2 Komplexität in virtuellen Maschinen

2.2.1 Anforderungen an virtuelle Maschinen

Neben der Anforderung, eine spezielle Hochsprache korrekt und effizient zu implementieren, kommt für viele VMs die Anforderung hinzu, auf möglichst vielen Plattformen verfügbar zu sein. So sollen die VMs auf allen für ihre Anwendungsgebiete relevanten Betriebssysteme lauffähig sein. Weiterhin müssen verschiedene Hardwareplattformen und insbesondere CPU-Architekturen berücksichtigt werden, um die gewünschten Eigenschaften auf allen relevanten Systemen anbieten zu können.

Da es bereits seit einiger Zeit praktikabel ist, auch auf eingebetteten Systemen Hochsprachen wie bspw. Java [Sun99, MO98, AKVN05] einzusetzen, ist das Spektrum der zu unterstützenden Systeme sehr breit gefasst. Es reicht von Systemen wie Smartcards oder Gerätesteuerungen, über mobile Anwendungen wie einfache Handys, aber auch leistungsfähige Smartphones, bis hin zu Standarddesktop- oder High-Performance-Computing-Systemen.

Diese vielfältigen Einsatzgebiete für virtuelle Maschinen spiegeln sich auch in unterschiedlichen Anforderungen wider. Je nach verfügbaren Ressourcen können VMs in verschiedene Richtungen optimiert werden. So kann entweder der Speicherbedarf minimiert oder die Performance unter höherer Speicherverwendung maximiert werden. Für mobile Systeme müssen Strategien für eine optimale Energieverwendung herangezogen werden. Im Gegensatz dazu ist auf dem Desktop die höchstmögliche Reaktionsgeschwindigkeit auf Benutzereingaben wichtiger.

Andere Anforderungen betreffen die Sicherheit und bspw. das Nachladen von Programmcode aus dem Internet. Dieser muss in einer Umgebung ausgeführt werden, die nicht jede beliebige Operation auf dem lokalen Rechner ohne Weiteres zulässt und somit den Nutzer vor Schadsoftware schützen kann.

2.2.2 Spezialanfertigungen, optimiert für einzelne Einsatzgebiete

Wie bei Betriebssystemen ist auch bei virtuellen Maschinen festzustellen, dass es für besondere Einsatzgebiete jeweils eigene und für einen bestimmten Anwendungsfall optimierte Implementierungen gibt. Beispiele sind hier allein im Java-Bereich Suns HotSpot-VM [Sun08a] für Desktop- und Serveranwendungen, BEAs JRockit [BEA08a] für Server, JavaCard [Sun08b, JJ05] für Smartcards, Googles Dalvik-VM [Goo08, Bor08] für Smartphones und andere Java-VMs für den Embedded-Bereich wie die KVM [Sun99], Kaffe [Kaf08] oder die JamaicaVM [aic08]. Darüber hinaus gibt es noch viele andere kommerzielle, aber auch Open-Source-Implementierungen von Java-VMs, die jeweils spezielle Eigenschaften mitbringen sollen. Speziell für die Forschung an VMs wurden bspw. Jikes [AAC+99] und Maxine [Mat08] entwickelt.

In diesen Implementierungen werden jeweils unterschiedliche Entwurfsentscheidungen getroffen und es kommen unterschiedliche Implementierungsstrategien bzw. Algorithmen zum Einsatz, um die VMs für ihr spezielles Einsatzgebiet zu optimieren. Sie sind somit Spezialanfertigungen für genau einen Bereich und ihre Verwendung ist dementsprechend eingeschränkt.

Wie bereits angedeutet, kommen im Bereich von mobilen Geräten Anforderungen wie bspw. Vorgaben zum Energieverbrauch hinzu. Dies führt zu einer Optimierung der VM für eine spezielle Hardware. Die Implementierung kann z. B. so angepasst werden,

dass die Speicherverwendung auf das Hardwarelayout der Speicherbestückung abgestimmt ist, um es dem Energiesparsystem zu ermöglichen Teile des Gerätes zu drosseln oder komplett zu deaktivieren und so die Akkulaufzeit entsprechend zu erhöhen. Dadurch ist die VM dann allerdings für genau diese spezielle Hardwarekonfiguration optimiert und kann nicht einfach auf eine andere übertragen werden, ohne angepasst werden zu müssen.

Solche Anforderungen kommen jedoch nur zu den bereits umfangreichen Anforderungen an eine VM hinzu. Die so entstehenden Systeme sind daher zumeist nicht so modularisiert, dass ihre Architektur für die Entwickler klar ersichtlich wird. Stattdessen sind die Strukturen nur implizit vorhanden. Zusätzlich sind die angebotenen Modularisierungstechniken der verwendeten Implementierungssprachen nur unzureichend und die Implementierungen von zusammengehöriger Funktionalität erstrecken sich über eine Vielzahl von Modulen. Diese könnte auf der Architekturebene in einem Modul gekapselt werden, sodass die ein Modul nur genau eine klar abgegrenzte Funktionalität bzw. ein Feature implementiert und nicht mehr viele Features in einzelnen Teilen. Eine Technik in diesem Sinne ist die aspektorientierte Programmierung [KLM+97], deren Kenntnis für diese Arbeit vorausgesetzt wird.

Durch die unvollständige Modularisierung und impliziten Beziehungen zwischen den Modulen ist es für die Entwickler schwierig einen Überblick über das System zu bekommen und die Dokumentation, die diese Information enthalten sollte, ist in den meisten Fällen nicht vorhandene oder veraltete [SAO05], sodass die Wartung des Systems zusätzlich erschwert wird.

2.2.3 IMPLEMENTIERUNGSSTRATEGIEN

Um Komplexität leichter handhabbar zu machen, wurden neben den verschiedenen Modularisierungstechniken, die von Prozeduren, über Klassen und Paketen bis hin zu Services und Subsystemen, also Modularisierung auf Architekturebene reichen, verschiedene Sprachen entwickelt, die unterschiedliche Mechanismen und Paradigmen zur Komplexitätsbewältigung bereitstellen.

Der klassische Ansatz zur Implementierung einer VM ist momentan noch immer die Verwendung einer systemnahen Programmiersprache wie C oder C++, um das wesentliche Fundament und teilweise auch große Teile der Klassenbibliotheken zu implementieren. Bei den populären VMs zählen dazu Suns *HotSpot*, Microsofts *Common Language Runtime* (CLR), sowie die Implementierungen verschiedener Scriptsprachen wie JavaScript, PHP, Python und Ruby. Dieser Ansatz hat sich seit Langem bewährt und es gibt viel Erfahrung, VMs auf diese Weise zu implementieren. Zusätzlich besteht die Überzeugung, dass durch die Systemnähe der Programmiersprache, eine performantere Implementierung erreicht werden kann, als dies mit Sprachen höhere Abstraktion möglich ist. Die Systemnähe ist jedoch zugleich ein Nachteil, da durch das geringe Abstraktionsniveau zumeist auf die Vorteile von aktuellen Hochsprachen verzichtet werden muss. Dazu gehören die Vorteile eines sicheren Speichermodels, automatischer Speicherverwaltung oder *Reflection* [Smi82].

Ein anderer Ansatz ist die Implementierung einer metazirkulären VM. Hierbei wird die VM nahezu vollständig in der zu implementierenden Sprache entwickelt. Erste Ansätze in diese Richtung gab es bereits mit Lisp in den 1960ern mit einem metazirkulären Lisp-Interpreter [McC60]. Später wurde dann S1 Lisp [BGS82, BGS83] mit dem Ziel entwickelt, möglichst viele Teile der VM in Lisp selbst zu implementieren.

Ein Zwischenschritt auf dem Weg zu metazirkulären VMs ist die Verwendung eines eingeschränkten Dialekts der zu implementierenden Sprache. Für Squeak, eine

Smalltalk-VM, wird eine Sprache namens Slang [IKM+97] verwendet. Diese ist mit dem Ziel entworfen worden, eine Sprache in der Smalltalk-Syntax zu haben, die auch in der VM ausgeführt werden kann. Als weitere wesentliche Eigenschaft sollte sie jedoch sehr einfach nach C übersetzt werden können, um einen Standardcompiler für die Erstellung der VM verwenden zu können. Einen ähnlichen Ansatz verfolgt PyPy [RP06] mit RPython [AACM07]. Auch hier kommt eine Teilmenge der zu implementierenden Sprache zum Einsatz, um damit die VM zu implementieren. Im Vergleich zu Slang ist RPython jedoch ungleich mächtiger und kann einen Großteil der Sprachkonstrukte der zu implementierenden Sprache verwenden. Zur Vereinfachung der Implementierung wurden jedoch die meisten dynamischen Sprachfunktionalitäten außen vor gelassen. RPython ist zwar eine dynamisch typisierte Sprache, verwendet aber ein statisches Typsystem, sodass keine Werte inkompatibler Typen ein und derselben Variable zugewiesen werden können. Zusätzlich gibt es keine Mehrfachvererbung mehr und das nachträgliche Verändern von Klassen, also das Hinzufügen oder Entfernen von Feldern oder Methoden ist nicht möglich. Wie Slang wird aber auch RPython in einem Transformationsschritt zuerst in z. B. C übersetzt, um dann mit einem Standardcompiler die ausführbare VM zu erstellen.

Zu den aktuellen metazirkulären VMs gehören Jikes [AAC+99], eine Java-VM von IBM für die Forschung, Maxine [Sun08c], ein ähnliches Forschungsprojekt von Sun und Klein VM [USA05] für Self, die ebenfalls bei Sun entstanden ist. Diese drei VMs sind jeweils in der zu implementierenden Sprache geschrieben und benötigen daher eine lauffähige VM dieser Sprache, um zum ersten Mal kompiliert und ausgeführt werden zu können. Sie können jedoch den vollen Nutzen aus der Metazirkularität ziehen, da jeweils nur sehr kleine Teile in anderen Sprachen wie C oder Assembler implementiert sind.

Unabhängig von der konkreten Implementierungsstrategie bleiben jedoch grundsätzlich die Bestandteile einer VM und damit die Komplexität der Implementierung gleich. Diese Bestandteile werden im nächsten Abschnitt genauer betrachtet und es wird untersucht, welche Beziehungen zwischen ihnen existieren, die die Komplexität beeinflussen.

2.3 Architektur virtueller Maschinen

2.3.1 Grundbestandteile

Die Untersuchung verschiedener VMs von Haupt et al. [HAT+08] hat gezeigt, dass diese sehr unterschiedlich aufgebaut sind und die gewählten Implementierungsstrategien sehr voneinander abweichen. Diese Einschätzung wird bei der zusätzlichen Betrachtung von Maxine [Mat08] und PHP [PHP08] ebenfalls bestätigt. Aber trotz der großen Unterschiede lassen sich in allen VMs Gemeinsamkeiten erkennen und wesentliche Grundbestandteile identifizieren, die für die Konstruktion einer virtuellen Maschine relevant sind.

Dargestellt ist die verallgemeinerte Architektur in Abbildung 1. Es wurden nur die wesentlichsten Grundbestandteile aufgenommen, die in einer Großzahl von VMs zu finden sind. Die Abbildung ist aufgeteilt in drei Bereiche. Der unterste Teil enthält die vom Betriebssystem bereitgestellte Funktionalität, um bspw. Dateien einlesen zu können. Hier dargestellt sind Daten, die das Anwendungsprogramm und die Klassenbibliothek der implementierten Sprache umfassen. Diese können nach dem Start von der VM eingelesen und zur Ausführung gebracht werden. Die beiden Bereiche darüber zeigen die Teile, die in der Implementierungssprache realisiert sind und darüber die

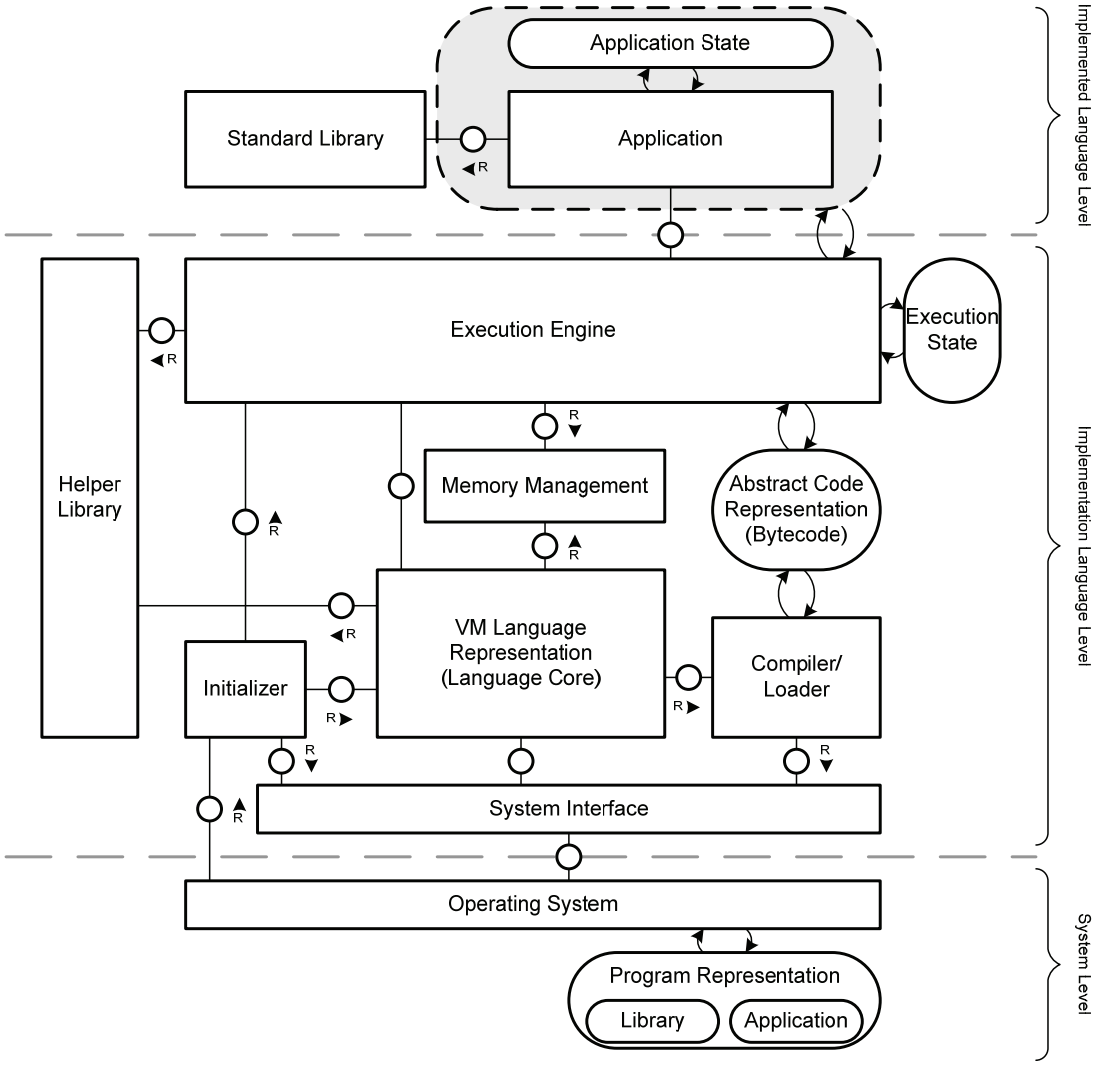


Abbildung 1 Architektur einer virtuellen Maschine (FMC-Aufbaubild)

eigentliche Anwendung und die Klassenbibliothek in der implementierten Sprache. Je nach gewählter Implementierungsstrategie können sich diese Bereiche auch vermischen bzw. wie bei metazirkulären VMs komplett verschmelzen.

Der *Initializer* stellt den Einstiegspunkt in die VM dar. Je nach Implementierung werden die verschiedenen VM-Teile von ihm initialisiert und auf die Ausführung vorbereitet. Die *VM Language Representation* stellt die Abstraktionen für bspw. Klassen und Methoden der implementierten Sprache bereit. Diese können jedoch bewusst allgemeiner gehalten werden, um nicht nur für eine spezielle, sondern für viele verschiedene Sprachen geeignet zu sein, wie dies bspw. in der CLR realisiert ist. Zum Laden von Bibliotheken oder Verarbeiten von Quelltext kann an dieser Stelle eine Verbindung zum *Compiler* bestehen.

Je nach Ausführung verarbeitet der Compiler direkt Quellcode oder im Falle vom Java und der CLR vorkompilierten Bytecode, lädt diesen in den Speicher und bereitet ihn für die Ausführung durch die *Execution Engine* auf. Wenn es sich um Bytecode bzw. eine vergleichbare Zwischendarstellung handelt, könnte dieses Teilsystem in der Implementierung durch eine *Loader*-Komponente ersetzt werden.

Die *Execution Engine* leistet die eigentliche Ausführung für den Fall, dass ein Interpreter verwendet wird, oder stellt die zur Ausführung nötigen Rahmenbedingungen für z. B. einen Just-in-Time-Compiler (JIT) sicher. Im allgemeinen Fall wird davon aus-

gegangen, dass der *Compiler* bzw. *Loader* und ein eventuell verwendeter JIT-Compiler getrennte Komponenten sind und über eine geeignete Zwischendarstellung miteinander kommunizieren.

Ein weiterer wesentlicher Bestandteil ist das *Memory Management*. In den meisten Hochsprachen wird eine automatische Speicherverwaltung eingesetzt. Für die Realisierung einer sogenannten Garbage-Collection gibt es verschiedene Ansätze, die als Teil des Memory Managements betrachtet werden, aber keine zwingende Voraussetzung für eine VM darstellen. Unabhängig davon, ob und wenn ja, welche Form der automatischen Speicherverwaltung implementiert ist, benötigen die meisten Module eine Möglichkeit Speicher anfordern und verwenden zu können.

Das Modul *Helper Library* stellt grundlegende Algorithmen oder Datenstrukturen bereit, die für die Implementierung der VM benötigt werden. Dazu können bspw. Hash- und Sortier-Algorithmen, Listen oder Hash-Tabellen gehören. Wie bei allen Modulen gilt auch hier, dass je nach gewählter Implementierungsstrategie die Helfer-Bibliotheken auch in der implementierten Sprache geschrieben sein können und damit in den oberen Teil der Abbildung gehören würden. Speziell im Fall der Helper Libraries würden sie dann Teil der *Standard Library* werden, was wiederum die Wiederverwendung erhöhen und die Programmierung vereinfachen würde, da das Programmierparadigma der implementierten Sprache verwendet werden kann.

VMs für den Produktiveinsatz enthalten darüber hinaus meist noch andere Funktionalität. Da Performance ein wesentliches Thema ist, sind adaptive Optimierung für einen JIT-Compiler, grundlegende Optimierungen wie *Tagged Integers*/*NonPointer* [GR83] oder *Threaded Interpretation* [Bel73] oft Bestandteil einer VM. Zusätzlich sind Threads und Synchronisierung bzw. Nebenläufigkeit allgemein eine wichtige Funktionalität. Je nach Sprache kommen dann eventuell noch Spezialitäten wie die *Virtual Images* [GR83] bei Smalltalk dazu.

In vielen Fällen kommt es durch diese zusätzlichen Elemente zu einer sehr starken Erhöhung der Komplexität, da Optimierungen in vielen Fällen die Verständlichkeit des Systems verringern und viele der genannten Elemente Auswirkungen aufeinander haben. Die typischen Interaktionen und Abhängigkeiten dieser Bestandteile werden im nächsten Abschnitt näher betrachtet.

2.3.2 Modulabhängigkeiten und Interaktionen

In virtuellen Maschinen zählt die Komplexität der Modulabhängigkeiten zu den großen Problemen, die ein Verständnis des Systems und somit die Wartung erschweren. Die typischen Implementierungssprachen bieten nicht die geeigneten Modularisierungsmechanismen, sodass es zu viele zyklische Abhängigkeiten zwischen den Modulen kommt. Dies erschwert die Abschätzung der Auswirkung von Änderungen, da durch die sich stark überschneidenden Sachverhalte, die Kapselung nur unzureichend ist. Eine klassische Schichtung lässt sich daher nur für sehr wenige Module verwenden. Das für das objektorientierte Design gedachte *Acyclic Dependencies Principle* [Mar96] lässt sich im Fall von so komplexen Abhängigkeiten nur schwer und zulasten der Verständlichkeit verwirklichen.

Zu den typischen Modulen, die vorwiegend als Bibliothek verwendet werden, gehören die *Helper Libraries*, das *Memory Management* und *System Interface*. Diese Einordnung ist jedoch implementierungsabhängig. So benötigt in einigen VMs das System Interface einen Mechanismus, um Parameter und Rückgabewerte in die Repräsentation des Betriebssystems bzw. zurück in das Format der VM übersetzen zu können und hängt damit von Details der *VM Language Representation* ab.

Beim Memory Management kann es je nach gewähltem Garbage-Collection-Verfahren (GC) beliebig komplizierter werden. So wird im Allgemeinen das Objekt-Layout beeinflusst, um bspw. ein Markierungsbit oder ein Referenzenzähler zu allen Objekten hinzuzufügen. Dazu kommen dann im Fall einer Reference-Counting-GC [DB76] über alle Module verteilt der notwendige Code, um die Referenzzähler bei Zuweisungen anzupassen. Damit wird einerseits die *VM Language Representation* und andererseits jedes Modul mit entsprechenden Zuweisungen beeinflusst. Bei anderen GC-Verfahren wird meist ein erweitertes Wissen über das Objekt-Layout benötigt, um den Objektgraphen traversieren [McC60] und gegebenenfalls auch Referenzen anpassen [FY69] zu können. Dies kann je nach verwendeter Implementierungssprache unterschiedliche Auswirkungen auf die Module haben. In einer objektorientierten Sprache wird es eventuell notwendig sein, für die Klassen spezifische Methoden bereitzustellen, um das Traversieren oder Anpassen der Referenzen ermöglichen zu können, wodurch die Modularität der GC erschwert wird, da diese Funktionalität nicht im GC-Modul lokalisiert ist.

Ähnlich verhält es sich mit verschiedenen anderen möglichen Erweiterungen und Optimierungen. Einige davon werden in der Fallstudie in Abschnitt 4.3.3 genauer erläutert. Dazu zählen bspw. die *Tagged Integers/Pointer*. Bei der Verwendung dieser Optimierung müssen Integer- bzw. Zeigeroperationen in der ganzen VM angepasst werden. Dies lässt sich, wie in der Fallstudie genauer gezeigt wird, abhängig von der verwendeten Implementierungssprache, nicht in allen Fällen modularisieren.

Die *Execution Engine* und die *VM Language Representation* sind ebenfalls zwei sehr stark miteinander verwobene Module. Die Execution Engine greift in vielen Implementierungen auf eine explizite Darstellung für Stack und Methoden zurück und verwendet diese, um das Ausführungsmodell zu implementieren. Zusätzlich stellen sie bestimmte Anforderungen an die Klassen, um eine effiziente Implementierung zu ermöglichen. Gleichzeitig bieten aber viele Sprachen bzw. VMs auch die Möglichkeit reflektive Verfahren zu nutzen, um den aktuellen Ausführungszustand zu inspizieren und teilweise sogar direkt zu manipulieren. Beim Einsatz eines JIT-Compilers ist dies entsprechend komplex und erfordert eine enge Verzahnung von Execution Engine und VM Language Representation. Sichtbar wird die Komplexität unter anderem bei der Implementierung von Continuations [SW00] oder Closures [Mir99]. Hier muss trotz einer möglichst vollständigen Ausnutzung der Performance der nativen Ausführung die Möglichkeit vorhanden sein, den Kontext in Form einer programmatischen Einheit zu erfassen und auch über reflektive Mechanismen zu verarbeiten.

Im Zusammenhang mit JIT-Compilern und Optimierung müssen viele Eigenschaften der implementierten Sprache beachtet werden. Bei objektorientierten Sprachen ist Polymorphie ein wichtiger Punkt. So kann über Call-Site-Caches [HCU91] der Nachrichtenversand optimiert werden. Ein JIT-Compiler kann auch die Vererbungsbeziehungen und die tatsächliche Nutzung der Polymorphie ermitteln, um entsprechend zu optimieren. Bei adaptiver Optimierung kann über ausführungsabhängiges Methoden-Inling oder *Trace-based JIT Compilation* [GPF06] Performance gewonnen werden. Für solche Optimierungen ist es jedoch notwendig, die Ausführung der VM zu überwachen und zu analysieren. Zur Ermittlung der nötigen Daten gibt es verschiedene Strategien [AFG+05], die jeweils unterschiedliche Auswirkungen auf die VM-Module haben. Einfaches Monitoring und eine Verwendung von Performancecountern könnten in der Execution Engine implementiert werden. Alternativ könnte der Bytecode durch den Compiler instrumentiert werden, um an relevanten Punkten

Profiling-Code einzufügen. Eine andere Variante wäre das Sampling. Hier wird die Ausführung in bestimmten Zeitabständen angehalten und der Aufrufstack analysiert. Dies erfordert entsprechende Eingriffe in die Execution Engine und eventuell Unterstützung durch die *VM Language Representation*, um an alle relevanten Informationen zu gelangen.

Viele dieser Optimierungen sind jedoch nicht dauerhaft möglich und je nach Sprache muss Vorsorge für Situationen getroffen werden, in denen sich bspw. dynamisch Klassenhierarchien verändern. Zusätzlich spielen andere Module der VM wie bspw. Garbage-Collection, Unterstützung für native Threads oder auch die Unterstützung von Sicherheitsmechanismen jeweils in diese Optimierungen hinein und müssen entsprechend mit unterstützt werden.

Eine konkrete Darstellung von Problemen dieser Art wird im Abschnitt 4.3.3 im Zusammenhang mit der Untersuchung von CSOM gegeben.

2.3.3 Auswirkungen auf die Modularisierung

Wie im letzten Abschnitt beschrieben, sind viele Module einer VM stark von anderen Modulen abhängig und ihre Implementierungen lassen sich mit den üblichen Modularisierungstechniken nicht vollständig entkoppeln. Eines der angeführten Beispiele war die Garbage-Collection. Diese beeinflusst das Objektlayout abhängig von der Wahl des konkreten Garbage-Collection-Mechanismus. Im Falle eines einfachen Mark/Sweep-Algorithmus würde ein einzelnes Bit ausreichen. Bei der Verwendung von Reference-Counting ist hingegen mehr Speicherbedarf für den Zähler einzuplanen. Neben dem Objektlayout sind aber auch der Interpreter und der JIT-Compiler von der Wahl des GC-Mechanismus beeinflusst. Im Fall der Mark/Sweep-GC müssen alle globalen Zeiger in der VM, die auf für die weitere Ausführung relevante Objekte verweisen, ermittelt werden können und im Fall der Reference-Counting-GC müssen die Referenzzähler bei allen Zuweisungen entsprechend angepasst werden. Ähnliche Verflechtungen bestehen mit einer Multi-Threading-Implementierung. Je nach GC-Mechanismus muss diese speziell abgestimmt werden, um nebenläufiges Verhalten sicher ermöglichen zu können. Von dieser Art gibt es, wie bereits angedeutet, diverse Abhängigkeiten zwischen den Funktionalitäten, sodass eine vollständige Modularisierung mit den typischerweise für die Implementierung verwendeten Sprachen nicht oder nur sehr schwer zu erreichen ist.

Diese Verflechtungen sorgen für die besondere Komplexität in virtuellen Maschinen und haben direkten Einfluss auf den Entwicklungs- und Wartungsaufwand für solche Systeme. Besonders bei Spezialanfertigungen, die für genau einen Anwendungsbereich entworfen wurden, kann so jede zusätzliche Anpassung mit erheblich erhöhtem Aufwand verbunden sein, da die Architektur des Systems auf den Anwendungsbereich optimiert wurde. So ist das Hinzufügen von Features wie eines umfassenden Sicherheitsmechanismus und der Möglichkeit des Sandboxing mit tiefen Eingriffen und erheblichen Änderungen am Gesamtsystem verbunden. Die Portierung auf andere Plattformen ist ebenso aufwendig, da bei den typischen Spezialanfertigungen zugunsten von Performance auf zusätzliche Abstraktionsebenen verzichtet wird.

Die Optimierung einer VM, egal ob zur Minimierung von Ressourcenanforderungen oder zur Erhöhung der Ausführungsgeschwindigkeit, erfordert zumeist tiefe Eingriffe in die VM und beeinflusst maßgeblich die Architektureigenschaften, da sie mit dem Verzicht auf Abstraktion und Indirektion einhergeht. Zusätzlich werden oft triviale und damit wartbare Implementierungsstrategien gegen wesentlich komplexere, aber dafür

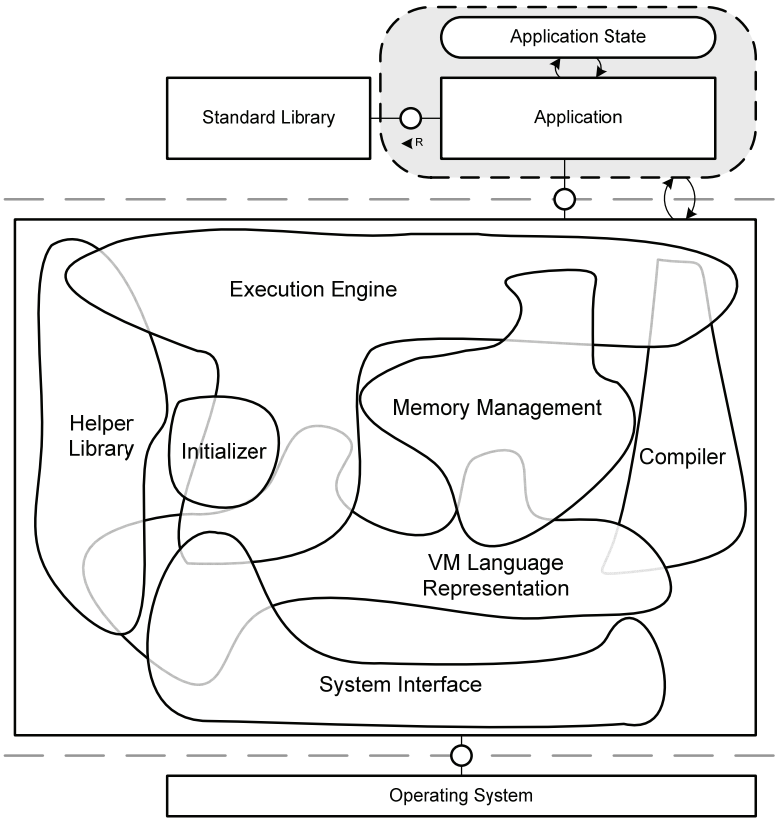


Abbildung 2 Tatsächliche Modularisierung in einer virtuellen Maschine, frei nach Haupt et al. [HAT+08] (FMC-Aufbaubild mit Veranschaulichung unklarer Modulgrenzen)

optimierte Strategien ausgetauscht. Zu den Optimierungen kann aber auch die Verwendung systemnaher Programmiersprachen bis hin zu Assembler gehören, wenn die verwendete Implementierungssprache durch die gewählten Abstraktionen bestimmte Konstruktionen nicht erlaubt und die zur Verfügung stehenden Compiler in dem Bereich nicht die gewünschten Ergebnisse liefern. Die Verletzung der Sprachkonzepte der Hauptimplementierungssprache erschwert jedoch die Wartung, da die Auswirkungen nicht klar ersichtlich sind und die Implementierung zusätzlich durch die Anforderungen anderer Module beeinflusst sein kann.

Die Schwierigkeit für die Entwickler besteht bei der Entwicklung und Wartung von VMs darin, alle Abhängigkeiten und Einflüsse zwischen den Modulen zu kennen und die Auswirkungen von Änderungen genau abschätzen zu können. Viele der Module sind so miteinander verflochten, dass sie nicht unabhängig voneinander entwickelt werden können. Änderungen in einem Modul wirken sich jeweils auch auf andere aus, die dann entsprechend angepasst werden müssen. Abbildung 2 zeigt, wie sich die Module tatsächlich im Quelltext widerspiegeln. Die in Abbildung 1 entworfene Architektur ist letztendlich nicht direkt erfassbar, da die Bestandteile der VM so stark integriert sind, dass sie zu einem komplexen, eventuell hochoptimierten System verschwimmen und die Modulgrenzen nicht mehr fassbar sind. Die Architektur bleibt somit nur ansatzweise erhalten und die Modulbeziehungen sind nicht als explizite Struktur sichtbar, da dazu nicht die geeigneten Sprachmittel bereitstehen.

In Produktiv-VMs wie Suns HotSpot-VM ist es für Entwickler sehr schwierig alle Module genau zu kennen, um die Auswirkungen von Änderungen abschätzen zu können. Die ca. 250.000 Zeilen Quelltext [Sun08a] sind von einzelnen Entwicklern nicht mehr zu überblicken. Das Grundproblem liegt darin, dass die Modulinteraktionen nicht

direkt für die Entwickler sichtbar, sondern als Abhängigkeiten und Aufrufbeziehungen implizit im Quelltext enthalten sind. Im Idealfall gibt es natürlich entsprechende Dokumentation, die diese Architektur beschreibt, jedoch hat Dokumentation meist das Problem, dass sie nicht mit dem System Schritt hält und Änderungen nicht oder nur unzureichend in die Dokumentation übertragen werden [SAO05]. So kann sie in den meisten Fällen nur einen groben Überblick über das System bieten, die Entwickler jedoch nicht mit den nötigen Details versorgen.

Im Bereich der Betriebssysteme sind die Probleme sehr ähnlich. Ein Beispiel, welches die Problematik veranschaulicht, ist die Integration von Multiprozessor-Unterstützung in den Linux-Kernel [SL04]. Diese wurde zwar bereits in Version 2.0 des Kernels übernommen, jedoch wurden erst nach und nach alle Teile auf das neue feingranulare Synchronisierungssystem umgestellt und selbst in der aktuellen Version 2.6 sind noch Teile vorhanden, die den alten und ineffizienten Mechanismus verwenden. Dies ist unter anderem darin begründet, dass die Synchronisierung nicht komplett modularisiert werden konnte und es bei einem so umfangreichen System wie dem Linux-Kernel auch nicht möglich ist, alle Auswirkungen einer Änderung sofort abschätzen zu können. Wenn dies nicht geeignet von der Implementierungssprache unterstützt wird, erhöht sich der Wartungsaufwand deutlich und Änderungen werden nicht durchgeführt, wenn sie bezüglich ihres Aufwands nicht abschätzbar oder weniger wichtig sind, wobei dies jedoch die Inkonsistenzen und damit den Wartungsaufwand verstärkt.

Viele der oben beschriebenen Module einer VM lassen sich auf Architekturebene mit einer gewissen Abstraktion geeignet erfassen und beschreiben. Auch Interaktionen und Abhängigkeiten lassen sich für einen Entwickler geeignet darstellen und ermöglichen ein einfacheres Verständnis der Zusammenhänge, um entsprechend effizienter Änderungen oder Wartungsarbeiten an der VM durchführen zu können. Für die Vermittlung eines schnellen Überblicks eignen sich besonders Modellierungssprachen zur grafischen Darstellung wie bspw. UML [Obj07] oder FMC [KGT06].

Problematisch an der Darstellung des Systems auf einer Ebene mit einem derart hohen Abstraktionsgrad wie der Architekturebene ist jedoch die Abbildung auf die Implementierung. Wie von Knöpfel et al. beschrieben [KGT06], gibt es verschiedene Ansätze der Dekomposition. So lässt sich die Architektur bspw. nach funktionalen oder nach hierarchischen Gesichtspunkten verfeinern. In klassischen Implementierungssprachen gehen diese Informationen jedoch verloren, da keine Sprachmittel mit Bezug auf die Architektur vorhanden sind. Dieser Mangel ist bereits seit längerem Gegenstand der Forschung und hat verschiedene Architekturbeschreibungssprachen hervorgebracht. Im nächsten Abschnitt werden die Eigenschaften von solchen Sprachen betrachtet, die für die Entwicklung von virtuellen Maschinen relevant sind.

2.4 Architekturbeschreibungssprachen

Für den Begriff ADL bzw. *Architecture Description Language* gibt es verschiedene Definitionen und Interpretationen. Insgesamt scheint es jedoch keine allgemein anerkannte Definition zu geben. Daher unterschieden sich Ansätze oder Sprachen, die als ADLs bezeichnet werden, sehr stark. Nach Vestal konzentriert sich eine ADL auf Strukturen höherer Ebene, die die Gesamtarchitektur einer Anwendung ausmachen, anstatt die Implementierungsdetails eines bestimmten Quelltext-Moduls zu betrachten [Ves93].

Die genauen Vorstellungen gehen jedoch nach Medvidovic und Taylor sehr weit auseinander [MT00]. Auf der einen Seite stehen die Vertreter der Ansicht, dass eine ADL vor allem das Verständnis und die Möglichkeiten zur Kommunikation über ein System verbessern sollen. Dafür sind ihnen zufolge entsprechend einfache, verständlich und möglicherweise mit grafischer Syntax ausgestattete ADLs sinnvoll. Auf der anderen Seite stehen die Vertreter der Ansicht, dass ADLs eine formale Syntax und Semantik sowie mächtige Analysewerkzeuge, Parser, Compiler, Code-Synthese-Werkzeuge und Laufzeitunterstützung mitbringen müssen. In ihrer Untersuchung machen sie einige wesentliche Eigenschaften von ADLs aus. Dazu gehört die explizite Modellierung von Komponenten, Konnektoren und deren Konfiguration sowie geeignete Werkzeugunterstützung für architekturbasierte Entwicklung und Evolution.

Als Definition für den Begriff Softwarearchitektur, verwenden sie die von Shaw und Garlan [SG96] gegebene:

Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

Aber auch in diesem Bereich gibt es noch keine Definition, die allgemein anerkannt ist. So listet das *Software Engineering Institute* der *Carnegie Mellon University* verschiedenste Variationen auf [CMU08]. Die beiden wesentlichen Elemente, die viele dieser Definitionen gemeinsam haben, sind die Ideen von markanten Teilen, aus denen ein System zusammengesetzt wird, sowie der Aspekt, diese Teile zu komponieren, bzw. ihre Interaktion zu beschreiben.

Dies sind wiederum die Punkte, die wohldefiniert sein sollten, um mit der Komplexität in virtuellen Maschinen umgehen zu können. Es sollen die Zusammenhänge im System deutlicher gemacht, die Modularisierung verbessert und somit das Verständnis des Systems erleichtert werden. Dies soll so erreicht werden, dass die Architekturelemente und ihre Interaktionen, ähnlich wie bei ArchJava [ACN02] explizit im Quelltext dargestellt werden. Damit soll eine zusätzliche Abstraktionsschicht über der Implementierung vermieden werden, einerseits um die damit verbundenen Probleme der Synchronisierung von Modell und Implementierung zu vermeiden, aber andererseits auch, um die Akzeptanz und Nutzwert für die Entwickler zu erhöhen. Damit benötigt eine ADL für VMs eine an die Implementierungssprache angelehnte Syntax und entsprechende Compiler-Unterstützung. Diese Form einer ADL würde damit zwischen den beiden von Medvidovic und Taylor beschriebenen Extremen liegen und auf Eigenschaften für formale Verifizierbarkeit zugunsten der Flexibilität verzichten. Für VMs ist es wichtiger die Einsatzmöglichkeiten nicht unnötig einzuschränken und nicht bestimmte Optimierungsmöglichkeiten von vornherein auszuschließen, nur um Anforderungen an Analysierbarkeit oder Verifikation gerecht zu werden, die in diesem Fall erst einmal keine Rolle spielen.

Die Kerneigenschaften einer Beschreibungssprache für die Architektur virtueller Maschinen sind damit die Folgenden:

- ein Modulkonzept zur Abgrenzung von Funktionalität auf der Architekturebene
- die Beschreibung von Modulinteraktionen, auf der Ebene ihrer Schnittstellen, klar abgegrenzt von der Modulimplementierung
- eine an die Implementierungssprache angelehnte Syntax
- Compiler-Unterstützung, um die Sprache in Kombination mit der eigentlichen Implementierungssprache verwenden zu können.

Im Vordergrund steht der Vorteil für die Entwickler, die durch zusätzliche Sprachmittel die Architektur direkt im Quelltext wiederfinden können und die Möglichkeit haben sollen, Interaktionen zwischen den Architekturelementen an wohl definierten Stellen im Quelltext zu platzieren. Damit wird das aktuelle Problem behoben, dass relevante Modulinteraktionen überall im Quelltext definiert sind. Die zusätzliche Strukturierung soll letztendlich das Systemverständnis und damit die Wartbarkeit verbessern.

3 VMADL

3.1 EINFÜHRUNG

Die *Virtual Machine Architecture Definition Language*, kurz VMADL, ist eine Beschreibungssprache für Module und deren Interaktionen auf Architekturebene. Vorgeschlagen wurde sie von Haupt et al. [HAT+08], um die bisher sehr starken Verflechtungen zwischen Modulen einer virtuellen Maschine auflösen zu können und einen höheren Grad an Modularität zu erreichen.

Die Ziele dieser Beschreibungssprache sind es, für Module eine bidirektionale Schnittstellenbeschreibung, die Funktionsdefinitionen und Ereignisse definiert, auf Architekturebene anzugeben und darauf aufbauend die Interaktion zwischen den Modulen anhand dieser Schnittstellen beschreiben zu können. Durch die Bereitstellung von Ereignissen neben typischen Funktionsdefinitionen wird die Interaktion gegenüber herkömmlichen Ansätzen mit Callbacks vereinfacht, da die Semantik klarer und reicher ist. Im Gegensatz zur Verwendung von AOP ohne Berücksichtigung von Modulgrenzen wird zusätzlich die Kapselung bewahrt [Ald05, GSS+06].

Die klare Abgrenzung von Modulen und ihren Interaktionen soll eine deklarative Abbildung der Architektur bewirken, die den Programmierern ein einfacheres Verständnis der Beziehungen und Abhängigkeiten zwischen den Modulen ermöglicht, um damit die Wartbarkeit der Software zu erhöhen. Darüber hinaus soll VMADL die Grundlage dafür bilden, virtuelle Maschinen in Form von Software-Produktlinien [DKO+97] zu entwickeln. Dafür wird ein entsprechend hoher Grad an Modularität und eine entsprechende Variabilität bei der Kombination dieser Module benötigt, die angelehnt an Sprachen der featureorientierten Programmierung [ALRS05b, Bat06] realisiert werden soll.

Ein weiteres Ziel von VMADL ist es, einen Rahmen zu bieten, der möglichst unabhängig von den verwendeten Implementierungssprachen des implementierten Systems ist und damit zusammen mit verschiedenen Sprachen verwendet werden kann. Somit ist die hier vorgestellte Syntax nur als ein Beispiel für eine konkrete Ausprägung der Sprache zu sehen.

Als Grundkonzepte stellt die Sprache sogenannte *Service Modules* und *Service Module Combinations* bereit, die im Wesentlichen den Komponenten- und Konnektor-Begriffen anderer ADLs entsprechen.

Ein Service-Modul *kann aus mehreren Implementierungsmodulen bestehen, und fasst diese auf der Architekturebene zusammen. Es definiert eine bidirektionale Schnittstelle, die von anderen Service-Modulen genutzt werden kann und neben typischen Funktionen und öffentlichen Ressourcen des Moduls besondere Ereignisse nach außen bereitstellt, die für andere Service-Module von Interesse sind.*

Ein Service-Modul repräsentiert dabei ein Feature.

> *Ein* Feature *ist eine Zusammenfassung von individuellen Anforderungen, die eine geschlossene, identifizierbare Einheit von Funktionalität bilden. [TWFL98]*

Für ein Service-Modul sollte im Allgemeinen gelten, dass dessen Funktionalität eine Bedeutung für die Systemarchitektur hat und eine wesentliche Charakteristik der virtuellen Maschine widerspiegelt. Beispiele für Service-Module sind die in Abschnitt 4.3.3 analysierten Features.

Die Service-Modulkombinationen bzw. *Service Module Combinations*, als zweites Grundkonzept, bietet die Möglichkeit spezielle Interaktionen zwischen den Modulen zu beschreiben, die über eine einfache Nutzung im Sinne einer API hinausgehen und z. B. mit den Mitteln der aspektorientierten Programmierung [KLM+97] realisiert werden können. Diese Modul-Kombinationen dienen dazu, die Modul-Interaktionen zu strukturieren, um sie den Programmierern zugänglicher zu machen und gleichzeitig die Variabilität und Komponierbarkeit der Service-Module sicherzustellen.

3.2 SPRACHUMFANG

Wie bereits erwähnt, sind die grundlegenden Sprachkonzepte Service-Module und Service-Modulkombinationen. In Quelltext 1 sind diese und die in VMADL möglichen Sprachkonstrukte dargestellt. Abbildung 3 zeigt eine Darstellung der Sprachkonzepte mithilfe von FMC.

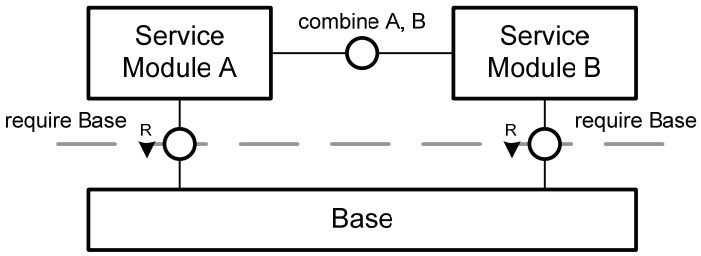


Abbildung 3 Grundkonzepte von VMADL (FMC-Aufbaubild)

Service-Module werden mit dem Schlüsselwort `service` eingeleitet und mit einem eindeutigen Bezeichner versehen. Im Beispiel sind dies die Service-Module *A* und *B* sowie das *Base*-Service-Modul. Die Darstellung in Abbildung 3 zeigt die besondere Bedeutung von *Base*. Dieses Service-Modul dient als Bibliothek und wird von den anderen beiden Modulen verwendet. *A* und *B* hingegen interagieren wechselseitig miteinander, wodurch eine gegenseitige Abhängigkeit zwischen ihnen besteht.

Im Quelltext wird die Verwendung einer Bibliothek über das `require`-Schlüsselwort und den Namen des Service-Moduls realisiert. Per Konvention werden diese Abhängigkeiten als Erstes in einer Service-Modul-Definition angegeben, wie in Zeile 6 gezeigt. In FMC lässt sich diese Besonderheit der Beziehung durch die Angabe eine Request-Richtung beschreiben. Damit ist explizit ausgesagt, dass das *Base*-Service-Modul keine Schnittstellen der anderen beiden Service-Module verwendet. In der Abbildung ist darüber hinaus explizit eine Grenze eingezeichnet, um die verschiedenen Schichten des Systems abzutrennen und den unteren Bereich als Basis- oder Bibliotheksschicht zu kennzeichnen.

```
01      service Base {
02          void Base_doFoo();
03      }
04
05      service ServiceModuleA {
06          require Base;
07
08          // declaration of functions
09          // and import or #include directives
10          #include <stdint.h>
11          void ServiceModuleA_doSomething(int64_t);
12          ...
13
14          expose { /* list of pointcut definitions */ }
15
16          MyNamedSection { ... }
17
18          startup {
19             void initialize_me();
20             expose { ... }
21          }
22
23          shutdown { ... }
24      }
25
26      service ServiceModuleB {
27          require Base;
28
29          void ServiceModuleB_doBar(char*);
30      }
31
32      combine ServiceModuleA, ServiceModuleB {
33          // advices on service module level
34          advice execution(ServiceModuleB_doBar())
35          : before() { ... }
36          ...
37
38          replace ServiceModuleA.NamedSection { ... }
39      }
```

Quelltext 1 VMADL Sprachkonstrukte

Nach der Angabe der Abhängigkeiten können sich, abhängig von der eigentlichen Implementierungssprache, z. B. Abhängigkeiten auf Implementierungsebene und typische API-Definitionen anschließen. Im Beispiel wird in Zeile 2 eine C-Funktion definiert, die von anderen Service-Modulen verwendet werden kann. In Zeile 10 wird eine `#include`-Direktive genutzt, um eine Standardbibliothek einzubinden. Der Bereich, in dem diese Angaben gemacht werden, ist implizit und wird nicht näher bezeichnet. Im Gegensatz dazu gibt es in VMADL noch explizite Bereiche. Diese wiederum lassen sich in vordefinierte und benannte Bereiche unterteilen.

Ein vordefinierter Bereich ist für die Definition von Ereignissen vorgesehen. Er wird mit dem Schlüsselwort `expose` gekennzeichnet und enthält, definiert durch die verwendete Aspektsprache, die Deklaration von sogenannten Pointcuts, welche relevante Ereignisse für andere Service-Module zur Verfügung stellen. Die Pointcuts sind dabei Beschreibungen bestimmter Situationen im Ablauf des Programms [KLM+97]. Diese explizite Bereitstellung von Pointcuts in der Modulschnittstelle ist angelehnt an die Idee der *Crosscutting Interfaces* (XPI) [GSS+06] und der *Open Modules* [Ald05]. Dieses Vorgehen dient dazu, eine starke Bindung von anderen Service-Modulen an die konkrete Implementierung eines Service-Moduls zu vermeiden. Andere Service-Module dürfen nur Pointcuts verwenden, die auf öffentlichen Methoden definiert sind oder von den Service-Modulen explizit bereitgestellt werden.

Mit den `startup`- und `shutdown`-Bereichen gibt es noch zwei weitere Bereiche, die eine spezielle Semantik mitbringen. Sie sind dafür vorgesehen die Initialisierung

des Systems zu vereinfachen, indem die Entwickler in diesen Bereichen Routinen angeben, die zu der entsprechenden Phase während des Systemstarts oder des Systemherunterfahrens ausgeführt werden sollen. Ebenso ist die Möglichkeit vorgesehen, spezielle Ereignisse nur innerhalb dieser Phasen zur Verfügung zu stellen, um z. B. einfacher auf Ereignisse während der Initialisierung reagieren zu können.

Benannte Bereiche, wie in Zeile 16 dargestellt, dienen der Gliederung der Schnittstellenbeschreibung und können alle Konstrukte enthalten, die auch auf der Ebene des Service-Moduls selbst verwendet werden können. Zusammen mit dem Konzept der Service-Modulkombination dienen sie jedoch auch als Ziele für die Umdefinition von Schnittstellenteilen.

Service-Modulkombinationen dienen als eigenständiges Sprachkonzept dazu, Modul-Interaktionen auf Schnittstellenebene zu beschreiben und z. B. über die bereits erwähnten benannten Bereiche Teile von anderen Service-Modul-Schnittstellen abändern zu können. Im Beispiel ist in Zeile 34 ein sogenannter Advice angegeben, der auf einer Funktion aus der Schnittstelle von `ServiceModuleB` definiert ist. Welche Syntax dieser Advice hat und wie er anzugeben ist, wird durch die verwendete Aspektsprache definiert, nicht jedoch von VMADL. Für die konzeptuelle Konsistenz wird für diese Advices jedoch die Vorgabe gemacht, dass sie nur auf den öffentlichen Schnittstellen von Service-Modulen arbeiten dürfen. Dies ist einerseits notwendig, um die Service-Modulkombination nicht von der Implementierung eines Service-Moduls abhängig zu machen, aber andererseits auch, um den Entwicklern die Möglichkeit zu geben, anhand der Service-Modul-Schnittstelle zu erkennen, an welchen Stellen Änderungen über aspektorientierte Programmierung vorgenommen werden können und welche Teile als geschützt vor solchen Eingriffen zu betrachten sind. Dieses `combine` zur Beschreibung der Service-Modulkombination wird in Abbildung 3 wird über die entsprechend benannte Verbindung zwischen den beiden Modulen dargestellt. Diese Verbindung repräsentiert eine allgemeine, bidirektionale Kommunikation zwischen den Service-Modulen.

Service-Modulkombinationen stellen im Gegensatz zu Service-Modulen keine eigene Schnittstelle bereit, sind aber dennoch vollständige Einheiten der Implementierung. Sie bieten alle Möglichkeiten, die auch ein Modul der Implementierungssprache bietet. So können nicht nur Advices definiert, sondern auch Funktionen oder Klassen, die in dieser Implementierungseinheit verwendet werden sollen, realisiert werden. Da sie aber keine Schnittstelle bereitstellen, sind diese Implementierungen in ihrer Verwendung auf die `combine`-Konstrukte beschränkt.

Ein besonderer Mechanismus für erweiterte Variabilität wird mit dem `replace`-Schlüsselwort gekennzeichnet. Mit diesem Schlüsselwort, dem Namen des Service-Moduls und des zu ersetzenden Bereichs, kann dessen Definition geändert werden. Eingesetzt werden kann dies unter anderem, wenn die Implementierungssprache Konstrukte bereitstellt, die nicht direkt geändert werden können. Ein Beispiel wäre die Abänderung von Makros für C durch spezielle Service-Module oder auch der Austausch von Pointcut-Definitionen, wenn diese beim Vorhandensein eines Service-Moduls angepasst werden müssen.

3.3 Testlauf für VMADL mit C und AspiCERE2

Die erste Version des VMADL-Compilers wurde genau auf den ursprünglich vorgesehenen Sprachumfang [HAT+08] und die für die von Haupt et al. durchgeführten Experimente zugeschnitten. Der VMADL-Compiler unterstützt die Definition von

Service-Modulen mit der Möglichkeit Funktionsdefinitionen, C-Präprozessoranweisungen, Advices und den Startup-Bereich zu verwenden. Für die damaligen Untersuchungen wurde die von Bram Adams entwickelte Aspektsprache Aspicere2 [AS07] verwendet, welche für die Definition von Pointcuts und Advices eine Mischung aus Prolog und C verwendet. Der VMADL-Compiler ist in der Lage, die VMADL-Definitionen in die für Aspicere2 benötigten Prolog- und C-Implementierungsdateien zu überführen, um anschließend Aspicere2 für die eigentliche Kompilierung der CSOM-VM verwenden zu können.

Letztendlich diente dieser VMADL-Compiler jedoch nur als Prototyp und Test, ob das verwendete ANTLR 3 [Par07] für diese Aufgabe in Kombination mit C als Sprache für die Compiler-Implementierung geeignet ist. Da sich mit ANTLR sehr schnell die nötige Grammatik entwickeln ließ und die Laufzeitbibliothek für C einfach zu verwenden war, ließ sich der Prototyp recht problemlos implementieren. C wurde gewählt, da CSOM ebenfalls in C implementiert ist und somit ein relativ homogenes Umfeld aus Werkzeugen und Produkt hätte erreicht werden können. Jedoch sind die mit C verbundene explizite Speicherverwaltung und das maschinennahe Programmiermodell sehr aufwendig, sodass diese Entscheidung revidiert wurde.

Da ANTLR in Java geschrieben ist und somit eine Java-Laufzeitumgebung für die Kompilierung der Grammatik vorhanden sein muss, wurde die Implementierung des VMADL-Compilers selbst ebenfalls auf Java umgestellt, da hier die deutlich höhere Abstraktion der Sprache eine wesentlich effizientere Entwicklung erlaubte, ohne eine komplett neue Anforderung an die Entwicklungsumgebung zu stellen.

Für die Durchführung der weiteren Arbeiten wurde außerdem die Aspektsprache Aspicere2 gegen AspectC++ [SGSP02] ausgewechselt. Der Grund war jedoch nicht die Aspektsprache an sich, denn letztendlich ist die für die Verwendung mit VMADL benötigte Funktionalität in beiden Sprachen vorhanden. Die beiden Werkzeuge unterscheiden sich jedoch sehr in ihrem grundsätzlichen Charakter.

Aspicere2 ist ein Forschungswerkzeug und erfordert einen erheblichen Aufwand, um auf aktuellen Linux-Systemen zum Einsatz gebracht werden zu können. Zudem ist es bisher nur im Rahmen der Forschung, für die es entwickelt wurde, zum Einsatz gekommen.

Dazu im Gegensatz hat AspectC++ Produktqualität. Es wird auch im kommerziellen Umfeld genutzt und aktiv von einem kleinen Personenkreis gepflegt. Zusätzlich zur kommerziellen Unterstützung, die sich z. B. in der Integration für Eclipse und Visual Studio äußert, gibt es eine Dokumentation, welche die wesentlichen Konstrukte der Sprache und auch die Werkzeuge nutzbar macht. Zudem ist der Einsatz auf einem aktuellen Linux-System ohne Probleme möglich. Weiterhin steht die Entwicklungsversion öffentlich zur Verfügung und fehlerbereinigte Versionen können sehr einfach aus den verfügbaren Quellen erstellt und genutzt werden. Dies ist sehr vorteilhaft, da gemeldete Probleme momentan recht schnell behoben werden. Insgesamt erschien damit der Einsatz von AspectC++ als unproblematischer.

3.4 VMADL-Compiler für C und AspectC++

3.4.1 Funktionsumfang und generierte Implementierungsdateien

Der implementierte VMADL-Compiler unterstützt den kompletten in Abschnitt 3.2 beschriebenen Sprachumfang. Im Anhang 1 ist die ANTLR-Grammatik angefügt, die für die Implementierung verwendet wurde. Um die Integration in den vorhandenen Build-Prozess von CSOM so einfach wie möglich zu gestalten, ist der Compiler mit einer

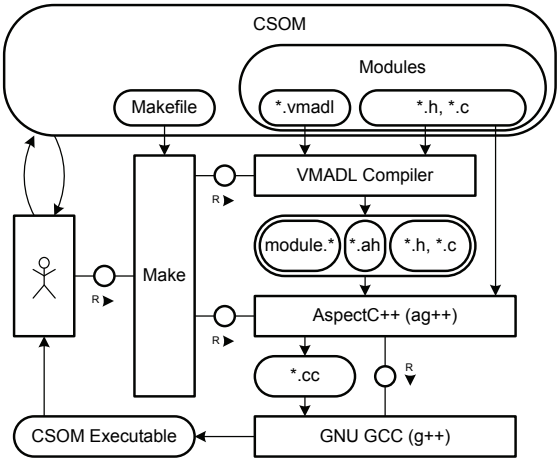


Abbildung 4 Zusammenspiel der Compiler (FMC-Aufbaubild)

Kommandozeilenschnittstelle ausgestattet, die eine Verwendung mit `make` ermöglicht. Außerdem wurde darauf geachtet, dass die Fehlermeldungen des Compilers die `make`-Konventionen berücksichtigen, wodurch diese sich sehr gut in die IDE-Unterstützung für `make` integrieren und z. B. in Eclipse ohne Weiteres als Markierungen direkt an der passenden Stelle im Quelltext angezeigt werden.

Um dem Grundgedanken der Unabhängigkeit von bestimmten Implementierungssprachen Rechnung zu tragen, wurden die Grammatik und die Implementierung so gestaltet, dass möglichst wenig Besonderheiten von C berücksichtigt werden müssen. Die Grammatik macht sich die Eigenschaft von C-ähnlichen Sprachen zunutze, dass geschweifte Klammern jeweils paarweise und in korrekter Verschachtelung auftreten. So muss nicht der komplette C-Sprachumfang in der Grammatik berücksichtigt werden und eine Anpassung auf z. B. Java oder C# erscheint relativ einfach. Zur Generierung der C-Header- und der AspectC++-AH-Dateien werden einfache Java-Format-Strings verwendet. Diese können im Falle einer Umstellung für eine andere Implementierungssprache abgeändert werden.

In Abbildung 4 sind die Interaktionen zwischen den verwendeten Compilern dargestellt. Für die Benutzer ist die Kompilierung weitgehend transparent und durch `make` gekapselt. Das Makefile beschreibt die einzelnen Kompilierungsschritte. Zuerst werden im dargestellten Beispiel die VMADL-Dateien von CSOM vom VMADL-Compiler in die Header-, Implementierungs- und AspectC++-Headerdateien übersetzt, die anschließend von AspectC++ zusammen mit der eigentlichen CSOM-Implementierung verarbeitet und in C++-Implementierungsdateien übersetzt werden. Diese werden letztendlich vom normalen C++-Compiler in eine ausführbare Form übersetzt.

Da die Implementierung von CSOM eine Nachahmung objektorientierter Programmierung verwendet, war es nötig, eine zusätzliche Sprache zu verwenden, um einige Möglichkeiten, die aus der featureorientierten Programmierung bekannt sind, verwenden zu können. Die speziell zu diesem Zweck entworfene Sprache ClassDL wird in Abschnitt 4.5.2 genauer vorgestellt und ist in den VMADL-Compiler integriert.

3.4.2 Konventionen für VMADL und C

Die Einführung von Service-Modulen hat erwartungsgemäß Auswirkungen auf die Gestaltung der Implementierung des C-Programms. Dabei ist der Umgang mit `#include`-Anweisungen der wichtigste Punkt. Das für VMADL gewählte Modell sieht vor, dass Implementierungsdateien, die Schnittstellenfunktionalität eines Moduls nutzen wollen,

diese über ein `#include <module.ModuleName>` importieren können. Die Module sollen jedoch keine eigenen Header-Dateien bereitstellen, die von anderen Modulen genutzt werden sollen. Alle Definitionen, die von Implementierungsmodulen eines anderen Service-Moduls genutzt werden sollen, müssen daher im Service-Modul mit definiert werden. Innerhalb eines Service-Moduls können jedoch weiterhin Header-Dateien genutzt werden, um das Service-Modul selbst geeignet untergliedern zu können.

Insgesamt hat sich der VMADL-Compiler im Rahmen der im nächsten Abschnitt beschriebenen Fallstudie als robustes und nützliches Werkzeug erwiesen. Durch die Berücksichtigung bestimmter Eigenheiten von CSOM, wie der Nachahmung objektorientierter Programmierung in C, ist aber vermutlich ein gewisser Anpassungsbedarf vorhanden, um diesen Compiler im Rahmen eines anderen Projekts einzusetzen. Unabhängig vom Compiler ist die Anwendbarkeit von VMADL mit anderen Sprachen oder Implementierungen jedoch nicht eingeschränkt.

4 Die Fallstudie CSOM

4.1 CSOM – Eine Smalltalk-VM für die Lehre

An der Universität von Århus in Dänemark wurde 2001/2002 in der Programmiersprache Java eine virtuelle Maschine mit dem Namen *Simple Object Machine* (SOM) entwickelt, die später von Michael Haupt und Tobias Pape am Hasso-Plattner-Institut in Potsdam nach C portiert wurde und dort den Namen CSOM bekam.

SOM wurde nach C portiert, um eine sehr einfache und maschinennahe VM zur Verfügung zu haben. Bei der Portierung wurde die klassenorientierte Struktur der Implementierung beibehalten und über die in C verfügbaren Mechanismen mithilfe von Makros nachgeahmt. Da CSOM vorwiegend für die Lehre eingesetzt wird, sollte sie die wesentlichen Elemente typischer VMs mitbringen und den Studenten die Möglichkeit bieten, verschiedene Programmierexperimente durchführen zu können. Genutzt wird sie momentan im Rahmen der Vorlesung *Virtuelle Maschinen* am HPI. Die in dieser Fallstudie verwendeten Erweiterungen für CSOM sind von Studenten für die Vorlesung implementiert worden.

Die für diese Zwecke an die Studenten ausgegebene Version von CSOM besteht aus nur wenigen, essenziellen Komponenten. Dazu zählen unter anderem ein Compiler, der einen dateibasierten Smalltalk-Dialekt in Bytecodes mit 16 verschiedenen Instruktionen übersetzt, ein passender Interpreter, eine einfache Mark/Sweep-Garbage-Collection [McC60] und eine kleine Klassenbibliothek, sowie einige Benchmarks. Damit steht ein recht einfaches System bereit, das etwa 10.000 Zeilen Quelltext umfasst und somit auch im Rahmen einer Vorlesung von Studenten bearbeitet werden kann.

Erweiterte Funktionalitäten, die für Smalltalk-Systeme normalerweise typisch sind, wie bspw. eine grafische Oberfläche, eine integrierte Entwicklungsumgebung oder eine Unterstützung für *Virtual Images* [GR83] sind nicht vorhanden. Ebenso wurde auf Performanceoptimierungen verzichtet, wenn sie zusätzliche Komplexität bedeuten. So ist der Interpreter durch ein einfaches `switch` auf dem Bytecode realisiert, anstatt komplexere Verfahren wie *Threaded Code* [Bel73] zu verwenden. Somit ist CSOM eine sehr kompakte und trotzdem voll funktionstüchtige VM, die sich auch gut für weiterführende Experimente wie das hier durchgeführte eignet.

4.2 Ziele der Fallstudie

Das Hauptziel der Untersuchung ist, festzustellen, wie VMADL auf eine VM angewendet werden kann und welche Problematiken sich dabei ergeben. Davon abhängig lässt sich der Sprachumfang von VMADL anpassen, um mit den auftretenden Modularisierungsproblemen geeignet umgehen zu können. Die gewonnenen Erkenntnisse sind bereits in die Sprachkonstrukte in Abschnitt 3 eingearbeitet worden. Ein wesentlicher Aspekt ist aber auch die anschließende Evaluierung des Nutzens von VMADL sowie der damit zusammenhängenden Vor- und Nachteile. Die Evaluierung wird im Abschnitt 5 beschrieben.

Mit der Anwendung von VMADL ist jedoch ein weiteres Ziel verbunden. Die damit verbundenen Verbesserungen für die explizite Darstellung der Architektur im Quelltext soll nicht nur allgemein die Wartbarkeit von CSOM verbessern, sondern durch eine optimierte Modularisierung neue Möglichkeiten für die Gestaltung der virtuellen Maschine an sich bieten.

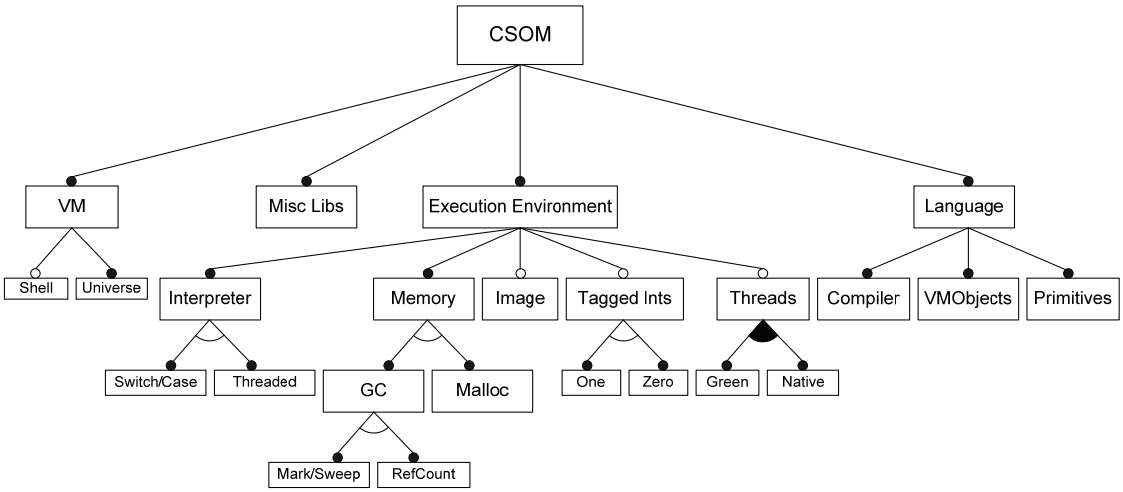


Abbildung 5 CSOM-Produktfamilie (Feature-Diagramm)

Durch die mit VMADL mögliche klare Abgrenzung von Service-Modulen soll es möglich sein, Module so zu implementieren, dass sie mithilfe eines geeigneten Konfigurations- bzw. Compiler-Werkzeugs frei wählbar genutzt werden können. Letztendlich soll dies zu einer Produktfamilie von CSOM-VMs führen, die es ermöglicht, eine virtuelle Maschine speziell für einen bestimmten Einsatzzweck zusammenstellen zu können und dabei die Vorteile der Produktfamilie voll auszuschöpfen. Abbildung 5 stellt eine Möglichkeit für die Gestaltungsflexibilität einer solchen Produktfamilie in Form eines Feature-Diagramms [Cza98] dar. Diese Darstellung kann als konzeptuelles Ziel für die Modularisierung und Gestaltung der Architektur von CSOM gesehen werden. Das tatsächliche Endresultat wird in den möglichen Freiheitsgraden aufgrund von verschiedenen Einschränkungen und Abhängigkeiten zwischen einzelnen Service-Modulimplementierungen eingeschränkter sein.

Das Ziel dieser Fallstudie ist es damit einerseits, VMADL auf eine bestehende VM anzuwenden und zu evaluieren und andererseits, die damit gewonnene Modularität zu nutzen, um eine VM-Produktfamilie zu entwickeln.

4.3 Reverse-Engineering von CSOM

4.3.1 Vorgehen zur Analyse des Quelltexts

Der erste Schritt, um VMADL auf CSOM anwenden zu können, ist die Analyse von CSOM und eine Modellierung der vorhandenen Module und ihrer Beziehungen. Dies ist notwendig, um ein besseres Verständnis der Zusammenhänge in CSOM zu erlangen und anschließend VMADL geeignet anwenden zu können.

Bei der Untersuchung zeigte sich schnell, dass das alleinige Lesen des Quelltexts nicht das gewünschte Ergebnis brachte und eine andere Möglichkeit gefunden werden musste, die Beziehungen zwischen den Modulen aus dem Quelltext zu extrahieren. Der erste Versuch dazu war die Anwendung von Doxygen [Hee08] auf CSOM, um die C-Include-Beziehungen zwischen den Modulen zu ermitteln. Da dies jedoch zu grob-granular war und so die Information, welche Funktionalität von einem Modul genutzt wird, verborgen blieb, musste ein anderes Werkzeug gefunden werden. Call-Graphen brachten hier leider auch keine deutliche Verbesserung, da sie zu viele Informationen enthielten und so groß wurden, dass sie nicht mehr handhabbar waren. Tatsächlich interessant für die Analyse war hingegen, welches Modul welche Funktionalität in einem anderen Modul verwendet. Um dies zu ermitteln, wurde ein für die CSOM-

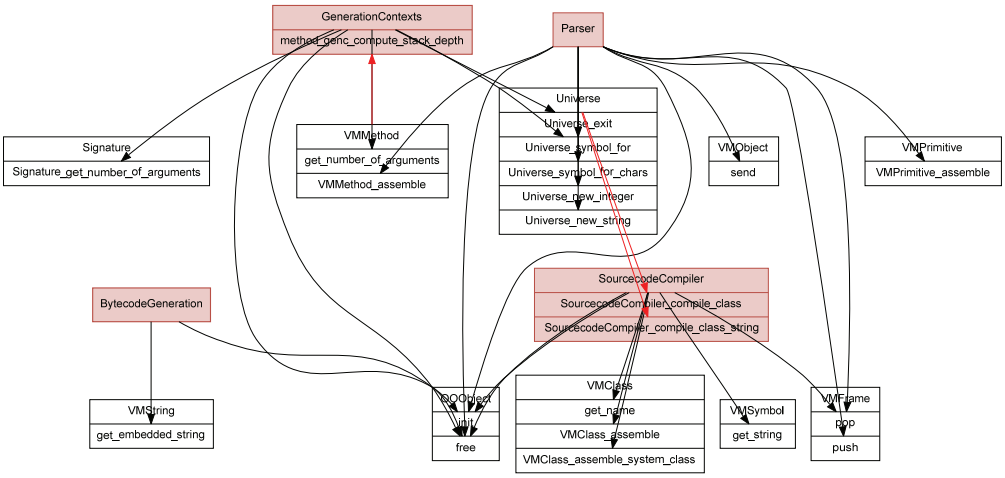


Abbildung 6 Aufrufbeziehungen zwischen Modulen des Compiler und anderer Module (Dot-Graph)

Implementierung geeignete vereinfachte C-Grammatik verwendet, um die Module und ihre definierten Funktionen zu ermitteln und anschließend über eine relativ grobe Schätzung herauszufinden, welches Modul diese Methode aufruft. Die verwendete Heuristik ist ein einfacher Zeichenkettenvergleich auf den Quelltext-Dateien ohne Berücksichtigung der Semantik. Für CSOM war diese vereinfachte Herangehensweise zufriedenstellend genau. Implementiert wurde dies über ein Groovy-Skript[1] [Laf08], welches eine Dot-Graphendefinition ausgibt, die anschließend mit Dot [AT&08] gerendert werden konnte. Diese Visualisierung stellt jeweils ein Modul in den Vordergrund und zeigt die abhängigen Module, wie in Abbildung 6 zu sehen. Die Bestandteile des betrachteten Moduls wurden hier farblich hinterlegt. Funktionen, die von anderen Modulen verwendet werden, sind durch einen roten Pfeil gekennzeichnet. Die Verwendung von Funktion anderer Module wird durch schwarze Pfeile dargestellt.[2]

Nachdem auf der Basis dieser Visualisierung das im nächsten Abschnitt beschriebene Modell der Architektur erstellt werden konnte, wurden die einzelnen CSOM-Erweiterungen untersucht, die geeignet modularisiert werden sollten. Bei den Erweiterungen handelt es sich, wie bereits erwähnt, um die Ergebnisse der Programmierübungen der Vorlesung *Virtuelle Maschinen* von Michael Haupt aus den Sommersemestern 2007 und 2008 am HPI. Es konnten Implementierungen für *Green Threads*, *Native Threads*, *Reference Counting GC*, *Threaded Interpretation*, *Tagged Integers* (Eins- und Null-basiert) sowie die Implementierung der Smalltalk *Virtual Images* für eine Untersuchung verwendet werden. Da sich CSOM jedoch weiter entwickelt hatte, war es nötig, die Implementierungen aus dem Jahr 2007 auf die aktuelle Quelltext-Basis anzupassen.

Nach diesen Vorbereitungen konnten die Änderungen, die durch die Erweiterungen in CSOM eingebracht wurden, analysiert werden. Hierbei galt es, alle Veränderungen am Grundsystem zu identifizieren. Anschließend konnte entschieden werden, ob diese relevant für das implementierte Feature sind und damit Teil des Service-Moduls werden sollten, oder ob es sich um Änderungen handelt, die sich im Sinne einer Umgestaltung bzw. Erweiterung ergeben haben. Mit den Ergebnissen dieser Analyse war es möglich mit der eigentlichen Überarbeitung zu beginnen.

---

[1] Das Skript ist auf der CD unter `/source/CSOM-Viz/` abgelegt.

[2] Die aus dem CSOM-Quelltext generierten Dot-Dateien und Graphen sind auf der CD unter `/Arbeit/dot/` abgelegt.

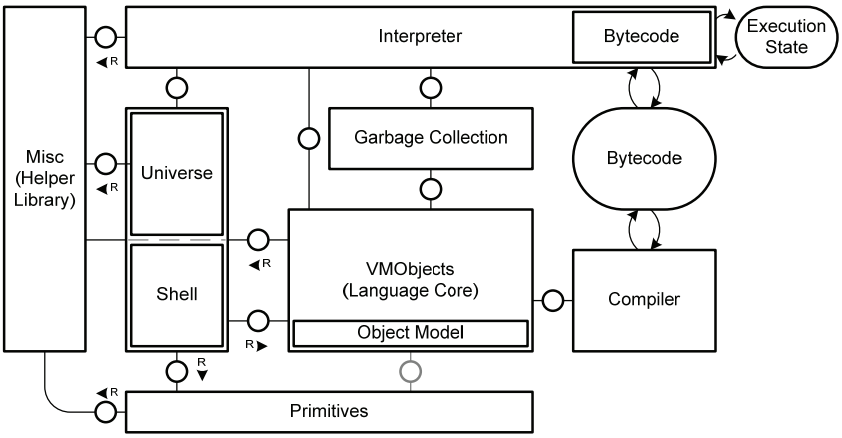


Abbildung 7 Architektur von CSOM, abgeleitet von allgemeinem Architekturbild (FMC-Aufbaubild)

Die Ergebnisse der Voruntersuchung werden im nächsten Abschnitt genauer vorgestellt, um darauf aufbauend in Abschnitt 4.5 die eigentliche Anwendung von VMADL auf CSOM vorstellen zu können.

4.3.2 Architektur der Ausgangsversion

Da CSOM selbst möglichst einfach sein soll, ist auch die Architektur recht übersichtlich. Abbildung 7 zeigt die Architektur von CSOM, angelehnt an die allgemeine Architektur von VMs aus Abbildung 1. Die wesentlichen Komponenten sind hier direkt unter dem Namen, den sie in CSOM haben, wiederzufinden. Da der Quelltext, wie in C üblich, in einzelne Implementierungsmodule aufgeteilt ist, die nach ihrer Funktionalität geordnet und bestimmten Verzeichnissen bzw. Architekturmodulen zugeordnet sind, lässt sich ein grober Eindruck des Systems bereits auf Verzeichnisebene erlangen. Dieser reicht jedoch nicht aus, um ein genaues Verständnis des Systems und insbesondere der Modulinteraktionen zu erlangen. Das Ergebnis der Analyse von CSOM ist in Abbildung 8 dargestellt. Die beiden Abbildungen sind von ihrem Informationsgehalt nahezu identisch, Abbildung 8 zeigt die Modulbeziehungen jedoch expliziter. Die Pfeile zwischen den Modulen stellen die gerichteten Benutzungsbeziehungen dar, die durch `#include`-Direktiven bzw. Funktionsaufrufe im Quelltext repräsentiert sind.

Das Service-Modul, welches die Module `Universe` und `Shell` enthält, wird in CSOM als *VM* bezeichnet. `Universe` übernimmt die Aufgaben des *Initializers* und bereitet die VM auf die Ausführung der Anwendung bzw. der `Shell` vor. Zusätzlich wird von `Universe` die Funktionalität bereitgestellt, um die Smalltalk-Ebene mit der C-Ebene zu verbinden. So werden hier alle globalen Objekte und verschiedene Factory-Funktionen implementiert, um Objekte für die Smalltalk-Ebene zu erzeugen.

Der Sprachkern für Smalltalk und auch grundlegende Funktionalität für die VM selbst wird im *VMObjects*-Service-Modul implementiert. Die Nachahmung von OOP für C ist in einem Untermodul *ObjectModel* definiert. Darauf aufbauend werden die grundlegenden Klassen für Smalltalk implementiert. `VMObject` definiert das Layout für Objekte allgemein, um darauf die Objekte von der Smalltalk-Sprachebene abbilden zu können. Die Felder eines Objekts werden dazu über ein Array von Zeigern definiert, die auf beliebige Instanzen von `VMObject` verweisen können. Alle Zuweisungen zu Objektfeldern auf der Smalltalk-Ebene werden auf diese Felder abgebildet. Zusätzlich stellt `VMObject` unter anderem ein Smalltalk-`send` bereit. Die Auflösung des Ziels, für das `send`, übernimmt hingegen `VMClass` mit der `lookup`-Routine, aber auch das Laden der

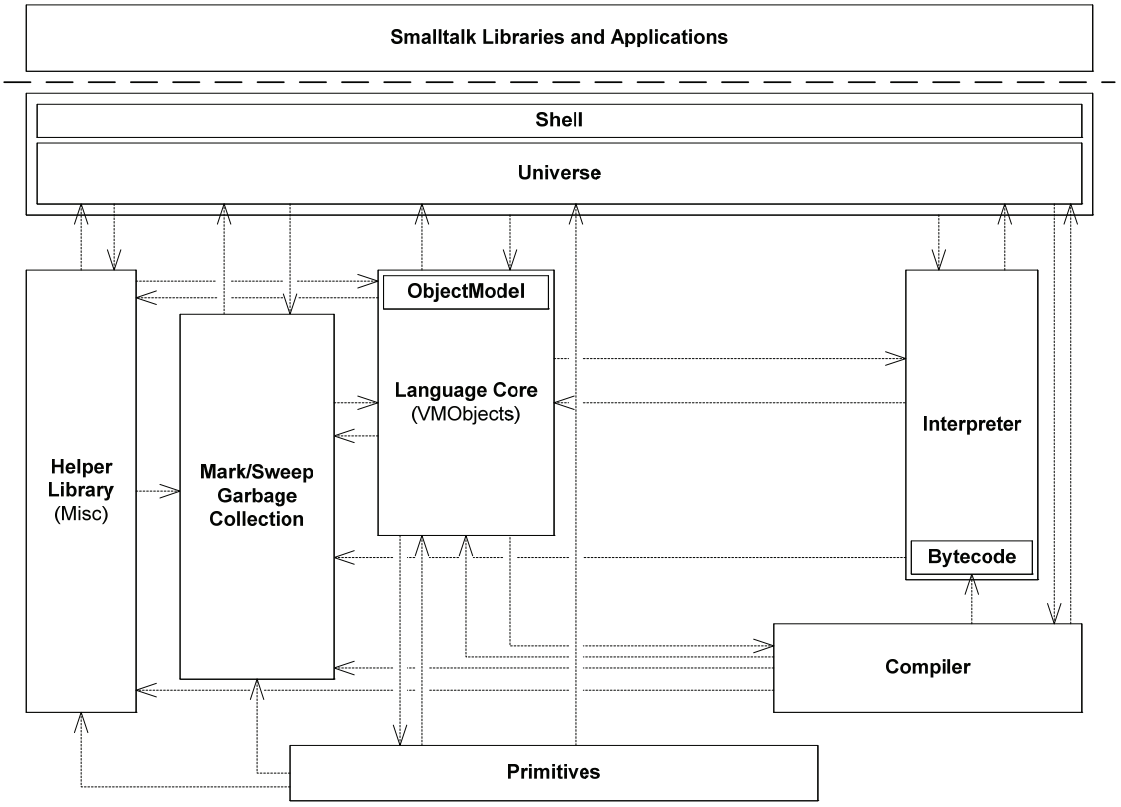


Abbildung 8 Architektur von CSOM mit detaillierteren Nutzungsbeziehungen (UML-Klassendiagramm)

Primitiv-Funktionen einer Klasse. Für die in Smalltalk definierten Methoden nutzt `VMClass` die vom *Compiler* erstellten `VMMethod`-Objekte, die den kompilierten Bytecode enthalten. Darüber hinaus wird das Objektlayout von der Mark/Sweep-Garbage-Collection beeinflusst, da sie ein zusätzliches Feld erfordert. Eine andere Aufgabe der *VMObjects* ist es die Unterstützung der primitiver Typen wie Integer, Double oder Strings zu realisieren.

Das *Compiler*-Service-Modul ist für die Verarbeitung der `*.som`-Dateien zuständig, welche den Smalltalk-Quelltext der Klassenbibliothek und der Anwendung enthalten. Er überführt sie in den für den *Interpreter* verständlichen Bytecode. Die Verwendung durch andere Module ist dabei relativ gering, wie auch bereits in Abbildung 6 zusehen war. In CSOM sind nur die Service-Module *VM* und *VMObjects* vom *Compiler* abhängig.

Als Ausführungseinheit kommt in CSOM, wie bereits erwähnt, ein sehr einfach gehaltener *Interpreter* zum Einsatz. Dieser interagiert vorwiegend mit dem *VMObjects*-Service-Modul, da dort wie bereits erwähnt die Implementierungen für Klassen, `VMClass`; Methoden, `VMMethod`; Blöcke, `VMBlock` und Frames, `VMFrames` definiert sind, welche die grundlegende Funktionalität für die Ausführungsumgebung und den Aufbau eines Verarbeitungsstacks bereitstellen. Im *Interpreter*-Service-Modul sind außerdem die Bytecodes definiert, die über einen einfachen `switch` verarbeitet werden. Für die Ausführung nutzt CSOM die Objektrepräsentation einer Stackmaschine. Die `VMFrame`-Objekte bilden einen Stapel von Methodenaktivierungen, die jeweils den Aufrufkontext und aktuellen Zustand über z. B. Aufrufparameter, aktuellem Programmzähler und durch die Methoden den auszuführenden Bytecode beschreiben. Über den `switch` realisiert der *Interpreter* den sogenannten Bytcode-Dispatch und ruft

die jeweils passende Interpreterroutine auf, die letztendlich durch die Modifikation des Stapels die Ausführung realisiert.

Funktionalität, die nicht direkt in Smalltalk abgebildet werden kann, oder für die eine Smalltalk-Implementierung nicht effizient genug wäre, werden in sogenannte Primitiven ausgelagert. Dies sind in C implementierte Funktionen, die direkt mit dem Smalltalk-Ausführungsstack interagieren und die benötigte Funktionalität bereitstellen. Dazu gehört bspw. die Addition von Integern, Array-Operationen oder die Interaktion mit der VM, um z. B. eine Klasse zu laden. Die Primitiven der Klassenbibliothek werden dazu im Service-Modul *Primitives* zusammengefasst und als eigene Bibliothek gesondert kompiliert. Beim Laden einer Klasse werden sie bei Bedarf nachgeladen.

Verschiedene Service-Module, wie *Compiler*, *VMObjects* oder *VM* benötigen für ihre Implementierung Datenstrukturen wie Hashmaps oder Listen. Diese komplexeren Strukturen oder Algorithmen, die jedoch nicht Teil des für die Smalltalk-Ebene erreichbaren Systems sind, werden im *Misc*-Service-Modul implementiert und können gemeinsam genutzt werden. Eine weitere Besonderheit dieses Moduls ist die Nutzung der expliziten Speicherverwaltung, die ebenfalls vom *Memory-Management*-Service-Modul angeboten wird. Die Datenstrukturen aus *Misc* sind damit nicht Teil des Smalltalk-Heaps und werden explizit verwaltet, um hier eine höhere Speichereffizienz zu erreichen. Das *Memory-Management*-Service-Modul bietet dazu eine zusätzliche Schnittstelle an, die in der aktuellen Implementierung direkt auf die C-Routinen zur Speicherallokation zurückgreift. Die anderen Module nutzen hingegen ausschließlich die implizite Speicherverwaltung. Dazu wird im Objektlayout, wie bereits erwähnt, ein zusätzliches Feld eingefügt. Davon abgesehen ist die Mark/Sweep-Garbage-Collection aber nahezu transparent implementiert und hat keine weiteren relevanten Abhängigkeiten zu anderen Modulen.

4.3.3 Analyse der CSOM-Erweiterungen

Wie bereits erwähnt, liegen für diese Untersuchung insgesamt acht unabhängig voneinander implementierte Erweiterungen von CSOM vor. Die Studenten haben jeweils genau ein Feature realisiert, welches als unabhängige Spezialanfertigung verstanden werden kann. Die Features wurden so implementiert, dass sie genau die geforderte Funktionalität aufweisen, aber es wurde nicht in besonderem Maße auf eine Modularisierung geachtet, die eine Integration verschiedener Erweiterungen bzw. Features ermöglichen würde.

Ziel dieser Analyse ist es, die verschiedenen Erweiterungen zu betrachten und die Änderungen im Vergleich zur ursprünglichen CSOM zu identifizieren. Diese Änderungen sollen bewertet und kategorisiert werden. Dabei sollen allgemeine Verbesserungen bzw. Umstrukturierung und featurebezogene Modifikationen unterschieden werden.

4.3.3.1 Mark/Sweep-Garbage-Collection

Die automatische Speicherverwaltung mit einem Mark/Sweep-Algorithmus wurde bereits in den 1960ern in Lisp eingesetzt und ist damit der vermutlich älteste Ansatz [McC60]. Die Grundidee besteht darin, dass dem Programm Speicher aus einem bestimmten bekannten Bereich auf Anforderung zugewiesen wird und sobald dieser Speicherbereich aufgebraucht ist und eine weitere Anforderung kommt, erst eine Mark- und anschließend eine Sweep-Phase durchgeführt werden, um ungenutzten Speicher freizugeben. In der Mark-Phase werden, ausgehend von allen globalen Objektzeigern in der VM, dem sogenannten Root-Set, alle erreichbaren Objekte traversiert

und markiert. Somit bleiben alle nicht erreichbaren und damit nicht mehr relevanten Objekte unmarkiert und können in der Sweep-Phase freigegeben bzw. die von ihnen belegten Bereiche im Speicher als frei markiert werden, um sie erneut zuweisen zu können.

Die Implementierung der Mark/Sweep-GC war in ihrer ursprünglichen Fassung nicht allein auf das *GC*-Service-Modul beschränkt. Bedingt durch Inkonsistenzen in der Implementierung konnte bei den von `VMObject` erbenden Klassen nicht einfach über die für die Smalltalk-Ebene sichtbaren Felder iteriert werden, um den Objektgraphen zu traversieren. Dies führte dazu, dass in allen Klassen, die von `VMObject` abgeleitet wurden, spezielle Methoden für das Markieren der erreichbaren Objekte und auch für das Freigeben implementiert wurden. Nach einer Überarbeitung und der Herstellung der Konsistenz in der Behandlung dieser Felder konnten diese Methoden aus den Klassen entfernt werden. Somit war es bereits ohne die Verwendung von VMADL möglich die Modularität der Mark/Sweep-GC wesentlich zu steigern.

Zu den Änderungen, die nicht direkt modularisiert werden konnten, zählt eine Erweiterung der Kommandozeilenverarbeitung, um Debug-Informationen und Statistiken über die GC abzufragen, sowie die Erweiterung von `Universe`, um an die Tabelle mit den für die Smalltalk-Ebene globalen Objekten zu kommen und damit ein vollständiges Root-Set zu erhalten. Zusätzlich wird die Konstruktor-Phase von Objekten geschützt, damit sie nicht vom GC unterbrochen wird. In solch einem Fall könnten Objekte, auf die noch kein Verweis gesetzt wurde, aber für die bereits Speicher reserviert wurden, gleich wieder verworfen werden und dies würde zu einem Speicherfehler führen. Als weitere Änderung kommt das bereits erwähnte `gc_field` zum Objektlayout der Klasse `VMObject` hinzu, welches für das Markieren der Objekte benötigt wird.

4.3.3.2 Reference-Counting-Garbage-Collection

Ein weiteres Verfahren zur automatischen Speicherverwaltung ist das Reference-Counting [DB76]. Wie der Name schon andeutet, wird bei diesem Verfahren die Anzahl der auf ein Objekt verweisenden Referenzen gezählt. Dazu werden bei allen Zuweisungen jeweils die Zähler an den betroffenen Objekten angepasst. Sobald an einem Objekt der Zähler auf Null reduziert wird, kann dieses Objekt freigegeben werden. Dies bedeutet dann auch, dass alle Referenzzähler von Objekten, die von diesem Objekt aus erreichbar sind, reduziert werden müssen. Problematisch ist dieses Verfahren jedoch bei zyklischen Datenstrukturen, da so Objekte im Speicher verbleiben, die nicht mehr erreichbar sind, aber durch gegenseitige Referenzen nicht automatisch beseitigt werden.

Die Implementierung des Reference-Counting-GCs ist selbst relativ einfach. Jedoch sind für diese Form der GC an sehr vielen Stellen in der VM kleine Ergänzungen notwendig, um die Referenzzähler jeweils korrekt anzupassen. Ein typisches Beispiel ist im Interpreter zu finden. Dort gibt es einen globalen Zeiger, der den aktuellen Frame referenziert. Bei einer Zuweisung zu diesem Zeiger muss beim alten Objekt entsprechend der Zähler verringert und beim neuen Objekt der Zähler erhöht werden. Ähnliches ist bei Objekten nötig, die innerhalb von Funktionen referenziert werden, wenn sich im Verlauf der Funktionsausführung eventuell der Referenzzähler verändert. Die meisten Wertzuweisungen werden im *VMObjects*-Service-Modul vorgenommen. Dort müssen entsprechend die Setter-Methoden und die `set_indexable_field`-Methode der `VMArray`-Klasse angepasst werden. Zusätzlich wurde `VMObject` um das Feld `ref_count` erweitert, um den Zähler zu speichern.

Im *VM*-Service-Modul wurden verschiedene Ergänzungen vorgenommen, um ähnlich wie für die Mark/Sweep-GC Statistiken anzeigen zu können. Zusätzlich wurde aber auch die Registrierung von globalen Objekten in `Universe` geändert, um den Referenzzähler auf einen Wert zu setzen, der angibt, dass diese Objekte von der GC nicht behandelt werden sollen.

4.3.3.3 Threaded Interpretation

Bei Threaded Interpretation handelt es sich um eine Technik zur Verringerung des Interpretierungsaufwands [Bel73]. Normalerweise wird Bytecode interpretiert, indem der aktuelle Bytecode gelesen, decodiert und anschließend die passende Methode aufgerufen wird. Mit Threaded Interpretation lassen sich die Anzahl der Speicherzugriffe und Sprünge für die Interpretation deutlich reduzieren. Dafür werden vor der Ausführung einer Methode die Bytecodes in die Speicheradressen übersetzt, an denen der Code liegt, der für ihre Ausführung zuständig ist. Diese Adressen werden aufgereiht und dienen nun direkt der Ausführung. Dieser Code wird auch als *Direct Threaded Code* bezeichnet. Anstatt jeden Bytecode einzeln zu interpretieren, wird dies einmalig im Voraus getan und bei der Ausführung muss nur noch jeweils der Zeiger auf die nächste Sprungadresse erhöht werden, die dann direkt am Ende einer dem originalen Bytecode zugeordneten Routine angesprungen werden kann.

Da es sich hierbei um eine Optimierung des Interpreters handelt, sind erwartungsgemäß relativ wenig Änderungen am Gesamtsystem vorhanden und die neue Funktionalität ist fast komplett im *Interpreter*-Service-Modul lokalisiert. Im Wesentlichen ersetzt das Modul für die Threaded Interpretation Methoden des Interpreters. Die Implementierung dieser Spezialanfertigung prüft dazu zur Laufzeit welche Form der Interpretation verwendet werden soll und ruft die passende Routine auf. Dies lässt sich jedoch auch so abändern, dass diese Entscheidung bei der Kompilierung getroffen wird, was wiederum die Änderungen gegenüber der Ausgangsversion von CSOM reduziert, da beide Implementierungen des Interpreters mit derselben Schnittstelle arbeiten können.

Das `Universe`-Modul wurde nur um die Verarbeitung von Kommandozeilenparametern erweitert. Relevanter ist hingegen die Änderung im *VMObjects*-Service-Modul. Dort wurden neue Methoden für den Zugriff auf den *Threaded Code* hinzugefügt, der zusätzlich zum normalen Bytecode in den Methodenobjekten gespeichert wird. Entsprechend wurde der Konstruktor von `VMMethod` angepasst, um mehr Speicher zu reservieren. Ebenfalls angepasst wurden die `set_`- und `get_bytecode_index`-Methoden von `VMMethod`, sowie die Hilfsmethoden im `Bytecode`-Modul, um Bytecode-Länge und Namen abfragen zu können.

4.3.3.4 Smalltalk Virtual Images

Im Smalltalk-80-Standard wurde die Idee des *Virtual Images* als eines der beiden Hauptbestandteile betrachtet [GR83]. Das andere ist die virtuelle Maschine selbst. Ein Virtual Image soll alle Objekte des Systems enthalten und damit das komplette System abbilden. Die Smalltalk-VM ist in der Lage dieses Image zu laden und die Ausführung an der Stelle fortzusetzen, an der sie vorher beendet wurde, und kann entsprechend auch die Ausführung zu einem beliebigen Zeitpunkt beenden und den zu diesem Zeitpunkt bestehenden Zustand als Virtual Image persistieren.

Für die Unterstützung dieser *Virtual Images* musste nicht nur die Implementierung auf C-Ebene angepasst, sondern auch die Smalltalk-Klassenbibliothek erweitert werden. So wurde die Klasse `Image` eingeführt und die Klasse `System` um eine

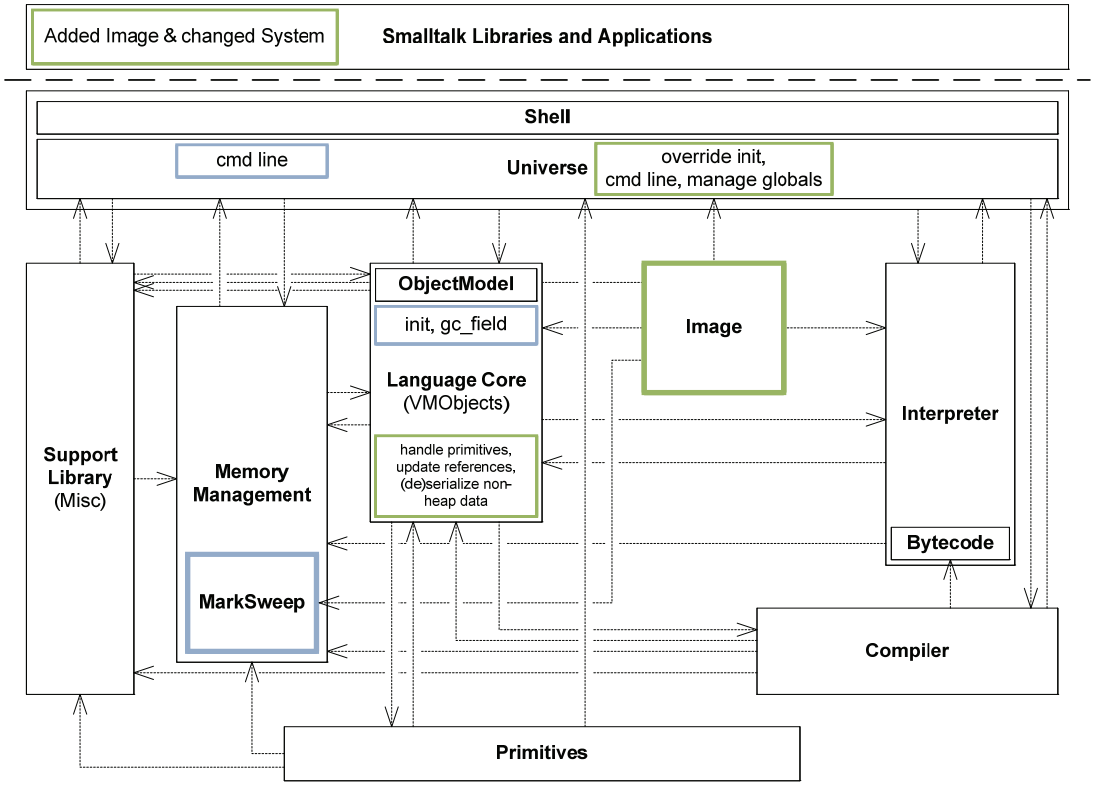


Abbildung 9 Einflüsse und Änderungen des Image-Service-Modules an CSOM (UML-Klassendiagramm mit Veranschaulichung von Änderungen)

Methode zum Anhalten des Systems und anschließenden Persistieren des Images erweitert. In diesem Zusammenhang wurde auch die Bibliothek der Primitivmethoden um die nötige Funktionalität für die beiden Klassen erweitert.

Auch bei dieser Implementierung ließ sich durch geeignetes Umstrukturieren die Lokalität des Features verbessern und im Service-Modul zusammenfassen. So wurden Funktionen, die zum Interpreter und zum `Universe` hinzugefügt wurden, jedoch nur für das *Image*-Service-Modul relevant sind, in dieses ausgelagert.

Problematisch sind jedoch Änderungen, die an den `VMObjects`-Klassen vorgenommen wurden. Hier wurden Methoden zum Korrigieren von Zeigern nach dem Laden des Images und zum Serialisieren bzw. Deserialisieren von Daten, die nicht auf dem Garbage-Collection-Heap liegen hinzugefügt. Nach der Behebung des in Abschnitt 4.3.3.1 erwähnten Konsistenzproblems bei der Behandlung der Smalltalk-Felder in Klassen sollte die Zeigerkorrektur ohne diese Methoden möglich sein. Die Methoden zur Behandlung von Nicht-Heap-Daten können erst entfernt werden, nachdem die Nutzdaten der `VMString`- und `VMSymbol`-Objekte komplett auf den Heap verlagert wurden. Beides ist momentan zwar angedacht, aber bisher nicht realisiert.

Das `Universe`-Modul wurde um Unterstützung für die Kommandozeilenparameter zum Laden der Images erweitert. Interessanter ist jedoch die Umgestaltung des Initialisierungsprozesses. Dieser war ursprünglich in einer einzelnen, sehr umfangreichen Funktion implementiert. Diese wurde in Abschnitte untergliedert, um den Prozess beim Laden eines Images entsprechend abändern zu können.

Problematisch ist die für die Realisierung der Virtual Images verwendete Strategie, da diese die Implementierung abhängig von der Verwendung der Mark/Sweep-GC-Implementierung macht. Die Designentscheidung den vorhandenen Speicherbereich der GC direkt zu persistieren vereinfacht zwar die Implementierung etwas, führt jedoch

dazu, dass dieses Service-Modul nicht zusammen mit einer beliebigen anderen GC eingesetzt werden kann. Eine direkte Traversierung des Objektgraphen mit anschließender Erstellung des Images wäre hier flexibler gewesen. Zusätzlich zur Abhängigkeit zur GC kommt außerdem noch die zusätzlich eingeführte Abhängigkeit zu einer externen Bibliothek (zlib) hinzu.

In Abbildung 9 sind die Änderungen durch die Implementierung des *Image*-Service-Moduls an CSOM dargestellt. Violett hervorgehoben sind die Änderungen, die durch die Mark/Sweep-Garbage-Collection verursacht werden. Die grünen Elemente resultieren hingegen direkt aus den Änderungen für die Virtual Images.

4.3.3.5 Green Threads

In Smalltalk-80 wurden Green Threads als Prozesse bezeichnet. Allgemein handelt es sich dabei um User-level Threads. Dies bedeutet, dass die Nebenläufigkeit nicht vom unterliegenden System bereitgestellt wird, sondern durch die Ausführungsumgebung selbst. In virtuellen Maschinen werden sie als Green Threads bezeichnet, aber auch in Betriebssystemen gibt es das Konzept von leichtgewichtigen Prozessen bzw. *Kernel Threads* oder gar leichtgewichtigen *User Threads* [SGG02]. Unter Windows werden leichtgewichtige Fibers [SGG02, Mic08b] auf Kernel Threads aufgesetzt und durch die Anwendung selbst verwaltet, da diese weniger Ressourcen verbrauchen als Kernel Threads. Letztendlich geht es dabei darum, auf ein bestehendes Konzept eine leichtgewichtigere Form von Nebenläufigkeit aufzusetzen, die durch die Integration in die Laufzeitumgebung weniger hohe Ressourcenanforderung als unterliegende Mechanismen hat und sich bspw. auch flexibel in die Sprache integrieren lässt.

Zur Implementierung der Green-Thread-Funktionalität in CSOM wurde, so weit es möglich ist, versucht die Implementierung direkt in Smalltalk zu schreiben. Dadurch wird nur ein kleiner Teil zur C-Ebene der VM hinzugefügt. Der größte Teil steht als Smalltalk-Klasse zur Verfügung, so auch einige kleinere Änderungen an der Klasse `Block` in der Smalltalk-Klassenbibliothek.

Auf der C-Ebene wurden Änderungen am Interpreter vorgenommen, um die gerade ausgeführte Methode und den dazugehörigen Frame austauschen zu können. Dazu wird der Scheduler in bestimmten Situationen vom Interpreter aufgerufen. Damit der Scheduler den aktuellen Frame ändern kann, wurde eine entsprechende Funktion eingeführt. Außerdem wurde die Möglichkeit für ein Kennzeichnen von kritischen Bereichen implementiert, sodass der Scheduler in diesen Bereichen den aktuell ausgeführten Thread nicht unterbricht.

Die `Shell` muss, wie die Smalltalk-Klasse `System`, für die Green Threads vor dem Ausführen der ersten Anweisung einen entsprechenden Thread erstellen, auf dem sie ausgeführt werden kann. Im `Universe`-Modul ist ein globales Objekt für die Klasse `Scheduler` hinzugekommen.

4.3.3.6 Native Threads

Im Gegensatz zu den Green Threads ist es bei der Implementierung der nativen Threads das Ziel, die Unterstützung für Nebenläufigkeit des unterliegenden Systems und damit z. B. auch die Unterstützung für Mehrkernprozessoren zu nutzen. Dies ist allein mit Green Threads nicht möglich. Der Nachteil dieses Ansatzes ist jedoch der entsprechend höhere Ressourcenbedarf durch die Verwaltung durch das Betriebssystem und z. B. einen höheren Aufwand für Kontextwechsel zwischen Threads auf einem Rechenkern.

Die Unterstützung für Betriebssystem-Threads wird vom *Native-Threads*-Service-Modul durch die Verwendung der Pthreads-Bibliothek [But97] erreicht. Diese ist damit Voraussetzung für die Verwendung dieses Moduls. Zudem muss der Build-Vorgang entsprechend angepasst werden.

Der Interpreter wurde mithilfe Thread-lokaler Speicher für native Nebenläufigkeit vorbereitet. Letztendlich läuft somit in jedem Thread eine eigene Interpreter-Instanz. Zusätzlich wird der Interpreter so modifiziert, dass er feststellen kann, ob der ausgeführte Thread pausiert werden soll. Das `Universe` wurde um einen Aufruf der Initialisierungsroutine für den Interpreter und die Terminierung der nativen Threads erweitert. Zusätzlich wurden Factory-Funktionen für die Klassen `VMThread`, `VMMutex` und `VMSignal` erweitert, die wie die `VMObjects`-Klassen auf der C-Ebene verfügbar sind. Zusätzlich gibt es für deren Klassenobjekte globale Zeiger, damit sie unter anderem in den Primitiven der Implementierung verwendet werden können.

4.3.3.7 Eins-markierte Integer

Bei markierten Integern handelt es sich um eine Optimierung, die es ermöglicht auf die Erstellung von Objekten mit dem damit verbundenen Speicherverbrauch für einfache Ganzzahlen zu verzichten. Gezeigt wurde ein derartiges Verfahren unter anderem in Smalltalk [GR83] und Self [CUL89]. Der Wert der Ganzzahl bzw. Integer wird mit diesem Verfahren direkt anstelle eines Objektzeiger im Speicher abgelegt und zur Unterscheidung zwischen Objektzeiger und Ganzzahl wird ein einzelnes Bit verwendet. Bei Eins-markierten Integern wird das niedrigstwertige Bit auf Eins gesetzt. Damit bleiben alle Zeiger unberührt und können wie vorher auch verwendet werden. Vor Integer-Operationen muss hingegen der korrekte Wert erst durch eine Verschiebung um ein Bit ermittelt werden. Dies ist möglich, da in den meisten Systemen für die Optimierung der Speicherzugriffe ein Alignment verwendet wird, wodurch die letzten Bits eines Pointers zumeist nicht verwendet werden müssen, da Objekte an bestimmten Grenzen im Speicher beginnen.

Für die Implementierung der mit Eins markierten Integer wurden einige Optimierungen an CSOM vorgenommen, die für eine konsequentere Verwendung der OOP-Nachahmung sorgen. Dies ermöglicht eine Änderung der Makros für die OOP-Nachahmung, um Methodenaufrufe bei Integern an eine globale Instanz mit der passenden Methodentabelle weiterleiten zu können. Die eigentliche Implementierung ist sehr stark lokalisiert. Neben den genannten Konsistenzänderungen gibt es nur in den Integer-Primitiven eine zusätzliche Überprüfung, die sicherstellt, dass tatsächlich alle Parameter bei einem solchen Aufruf Integer sind.

4.3.3.8 Null-markierte Integer

Wie bei den Eins-markierten Integern wird auch hier das niedrigstwertige Bit eines Objektzeigers verwendet, um zwischen Integer und Zeiger unterscheiden zu können. Durch die Markierung der Zeiger mit einem Eins-Bit müssen jedoch alle Dereferenzierungen angepasst werden. Die Integer-Operationen können hingegen direkt durchgeführt werden, da sich das Null-Bit am Ende neutral verhält.

Die Implementierung der mit Null markierten Integer lässt sich nicht so umstrukturieren, dass sie in einem Modul lokalisiert werden kann, da an allen Stellen im Quelltext, an denen Zeiger dereferenziert werden, ein Makro eingeführt wurde und keine geeigneten Sprachmittel zur Verfügung stehen, die Dereferenzierung anderweitig abzuändern. Da die Anpassung der Dereferenzierung jedoch ausschließlich für Null-

markierte Integer verwendet wird, ist es nicht sinnvoll, diese Änderung als so allgemein zu betrachten, dass sie nicht modularisiert werden müsste.

4.3.3.9 Einordnung der Änderungen gegenüber der Ausgangsversion

Die Implementierungen der verschiedenen Features durch die Erweiterung von CSOM haben teils sehr unterschiedliche Änderungen am Kernsystem verursacht. Darunter sind nicht ausschließlich sogenannte *Crosscutting Concerns*, die direkt ihrer Funktionalität zugeschrieben werden können.

Einige Änderungen fügten bisher nicht implementierte Basisfunktionalität zu bestehenden Klassen hinzu. Dazu zählen Getter und Setter, aber auch die Methode `contains` der `Hashmap`. Diese Funktionalität wurde zwar von einer bestimmten Feature-Implementierung beigetragen, ist jedoch nicht damit verbunden, sondern allgemeingültig verwendbar.

Das Umstrukturieren und Beheben von Inkonsistenzen in anderen Modulen ist ebenso allgemeingültig und trägt direkt zur Qualität des Quelltextes bei, auch wenn es durch eine bestimmte Anforderung motiviert wurde. Beispiele für diese Arten von Änderungen war die durch den Mark/Sweep-GC motivierte Korrektur der Semantik für die Anzahl der Klassenfelder, die für Smalltalk-Klassen erreichbar ist, sowie die Aufteilung des Initialisierungsprozesses in geeignete Phasen.

Neu eingeführte Implementierungsmodule, zusätzliche Primitiven, Änderungen am Objektlayout, wie sie von den GCs vorgenommen wurden, Änderungen am Objektmodell bzw. den dazugehörigen Makros, neue Methoden an bereits vorhandenen Klassen und auch zusätzliche Abhängigkeiten zu externen Bibliotheken, können hingegen eindeutig der Funktionalität zugeordnet werden. Daher sollten diese, soweit es praktikabel ist, direkt in den entsprechenden Service-Modulen implementiert werden. Dazu gehören außerdem Änderungen an der Smalltalk-Klassenbibliothek. Für die Thread-Implementierungen wurden verschiedene Klassen direkt in Smalltalk implementiert. Zusätzlich wurden bspw. für die Virtual-Image-Implementierung und die Garbage-Collections bestehende Klassen verändert. So musste die Klasse `System` um die Methode `hibernate` erweitert werden. Für die GCs musste wiederum die `TestSuite` als Teil der Smalltalk-Klassenbibliothek angepasst werden, um auf die veränderte Objektgröße Rücksicht zu nehmen. Für eine Modularisierung auf dieser Ebene müssten geeignete Werkzeuge bzw. Sprachmechanismen wie AOP oder FOP bereitgestellt werden. Da sie jedoch nicht Teil der eigentlichen VM sind, wird dies in dieser Arbeit nicht weiter betrachtet.

Als Ergebnis der Analyse der Feature-Implementierungen kann festgestellt werden, dass jede von ihnen eine Spezialanfertigung für genau einen Anwendungsfall darstellt. Die Art und Weise, auf die sie implementiert wurden, erlaubt es mit den Sprachmitteln von C nicht, ohne erheblichen Aufwand die verschiedenen Features zusammenzuführen und somit eine VM mit mehreren dieser Features zu erhalten. Prinzipiell wäre dies für einige Kombinationen zwar mit relativ überschaubarem Aufwand realisierbar, aber das Resultat wäre eine neue Spezialanfertigung, die für einen weiteren Anwendungsbereich optimiert ist. Theoretisch setzen all diese Spezialanfertigungen auf eine gemeinsame Quelltextbasis auf, können jedoch mit normalen Werkzeugen nicht gemeinsam gepflegt werden, da sie letztendlich eigenständige Kopien darstellen. So müssen bspw. Fehlerbehebungen an einzelnen Modulen stets auch auf alle einzelnen Spezialanfertigungen per Hand übertragen werden.

Außerdem ist es problematisch, dass die Implementierungen der Features die bestehende Architektur nur unzureichend respektieren und in den meisten Fällen ver-

ändern. So kommen neue Interaktionen zwischen bestehenden Modulen hinzu und es werden Feature-spezifische Änderungen an einem allgemeinen Modul vorgenommen. Diese Auswirkungen wurden bereits in Abbildung 9 veranschaulicht. Im Vergleich zur ursprünglichen Abbildung 8 werden dort zusätzlich die durch die Implementierung eines Features an CSOM vorgenommen Änderungen und hinzugekommen Service-Modul-Interaktionen dargestellt.

Um diese Situation zu verbessern, wird in den folgenden beiden Abschnitten beschrieben, wie die Implementierung umstrukturiert wurde, um anschließend VMADL anwenden zu können. Dadurch soll einerseits die Architektur von CSOM klarer dargestellt werden und andererseits die Kombination von Features durch die Entwicklung einer Produktfamilie vereinfacht werden.

4.4 Überarbeitungsschritte und Anwendung von AspectC++

Um von diesen schlecht zu wartenden Spezialanfertigungen zu einer Produktfamilie mit gemeinsamer Quelltextbasis zukommen und mit VMADL eine vollständige Modularisierung der untersuchten Features zu erreichen, wurden die einzelnen CSOM-Erweiterungen in mehreren Schritten überarbeitet.

Die oben vorgestellte Analyse der Features stellt den ersten Schritt dar. Da die Implementierungen für die Reference-Counting-GC und Green bzw. Native Threads auf eine alte CSOM-Version aufsetzten, mussten diese als erstes auf die neue CSOM portiert werden. Im Anschluss daran wurde die Mark/Sweep-GC in ein eigenes Modul ausgelagert und eine einfache auf `malloc` basierende Speicherverwaltung hinzugefügt. Dadurch ist es möglich, für die Speicherverwaltung zwischen einem rein allozierenden Modul und verschiedenen GC-Implementierungen wählen zu können. Das rein allozierende Modul hat insbesondere bei Performanceuntersuchungen einen Vorteil, da bei der Verwendung dieses Moduls kein störender Einfluss durch eine GC besteht.

Damit waren neun verschiedene CSOM-Versionen vorhanden, die auf einer gemeinsamen Quelltextbasis aufsetzten. Abbildung 10 gibt einen Überblick über das für die Entwicklung verwendete Quelltextverzeichnis[3], die verschiedenen CSOM-Varianten und ihre Beziehungen zu einander. Die Pfeile zeigen die Beziehung zwischen den Varianten. Die Semantik ist dabei ähnlich einer Vererbung. Die Variante, auf die der Pfeil zeigt, ist die Ursprungsversion, die ihre Haupteigenschaften an die davon abgeleitete Version weitergibt.

Die Ausgangsversion von CSOM mit der Mark/Sweep-GC bildet die Grundlage für die lokale `trunk`-Version. Diese CSOM enthält alle Änderungen, die mögliche Verbesserungen für den Hauptzweig darstellen. Sie ist die Version, die am nächsten an den offiziellen Hauptentwicklungszweig reicht, und wird z. B. für Vergleichsmessungen herangezogen.

Von dieser Grundlage ausgehend, wurden, beginnend bei der `feature-gc`-CSOM, also der Mark/Sweep-GC-Implementierung aus dem `features`-Verzeichnis, die Arbeiten schrittweise durchgeführt, um die Modularisierung mit AOP bzw. AspectC++ zu erhöhen. Der erste Schritt dazu war die Portierung des CSOM-Quelltextes von C99 auf C++, um ihn mit dem GNU GCC `g++` übersetzen zu können. Zu den dafür nötigen Änderungen gehört im Wesentlichen die Abänderung aller Feldnamen, die das Schlüsselwort `class` verwenden und die Anpassung der Primitiven, sodass sie vom Compiler weiterhin als normale C-Funktionen exportiert werden. Erreicht wurde dies

[3] Zu finden auf der CD unter `/source/CSOM`

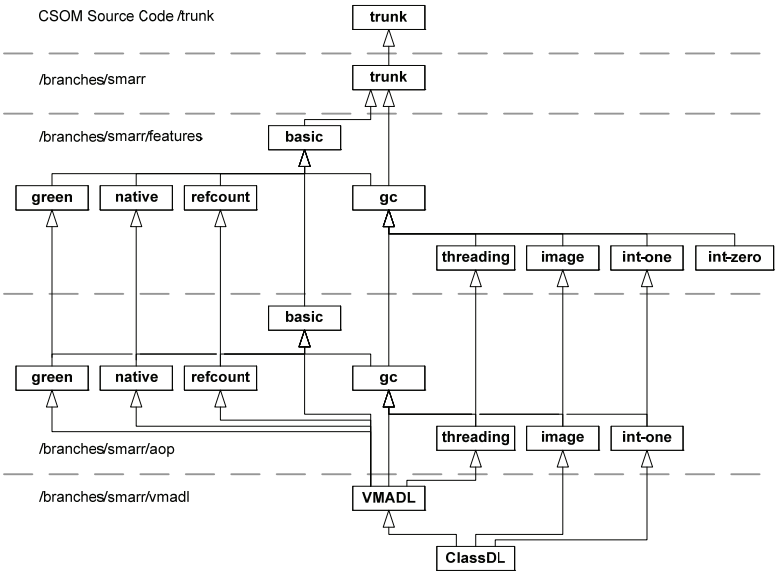


Abbildung 10 Beziehungen zwischen den CSOM-Varianten

über die von C++ angebotene `extern "C" {…}` Notation. Diese Änderungen wurden entsprechend auf alle CSOM-VMs im `features`-Verzeichnis übertragen.

Im zweiten Schritt wurde für die `feature-gc`-CSOM das Problem der inkonsistenten Initialisierung der Feldanzahl von Objekten behoben. Im Anschluss daran waren die Vorbereitungen für eine Anwendung von AspectC++ abgeschlossen und es wurde ein Entwicklungszweig `aop-gc` angelegt, indem die Modularisierung mithilfe der aspektorientierten Programmierung implementiert wurde. Im selben Schritt wurde auch eine entsprechende Version `aop-basic` angelegt, damit die Änderungen zwischen den Basisversionen einfach verfolgt werden konnten. Die wesentliche Änderung im `aop`-Zweig ist die Umstellung der Initialisierung von Objekten von der Verwendung variabler Argumentlisten auf eine feste Signatur mit zwei `void`-Zeigern. Für die aktuell implementierten Features ist dies vollkommen ausreichend und ermöglicht die Verwendung von AspectC++-Advices auf den Initialisierungsmethoden. Diese Änderung war notwendig, da AspectC++ in der verwendeten Version keine variablen Argumentlisten für die verwendeten Advice-Typen unterstützt.

Nachdem die Mark/Sweep-GC mit AOP vollständig modularisiert werden konnte, wurden dieselben Schritte ebenfalls für die Reference-Counting-GC sowie die Green und Native Threads durchgeführt. In der Arbeit von Haupt et al. [HAT+08] wurden bereits die entsprechenden Vorarbeiten geleistet, sodass hier keine größeren Probleme auftraten. Kleinere konzeptuelle Änderungen an CSOM, die während der Arbeit an den Feature-Implementierungen sinnvoll erschienen, wurden jeweils in die entsprechende Basis-Version übertragen und soweit notwendig und sinnvoll auf die anderen CSOM-VMs im selben Entwicklungszweig übertragen. Wenn es angebracht erschien, wurden sie aber auch zurück in den lokalen `trunk` übernommen, um dort alle wesentlichen Änderungen zum späteren Vergleich vorliegen zu haben.

Nachdem die ersten vier Features komplett modularisiert waren, wurde die Entwicklung der VMADL-CSOM im Zweig `vmadl-main` begonnen. Nach der Entwicklung des VMADL-Compilers konnten die VMADL-Service-Modul-Beschreibungen erstellt und CSOM mithilfe eines angepassten Makefiles in Form einer Produktfamilie kompiliert werden. Das Makefile ermöglichte auch bereits die Auswahl der zu erstellenden

Konfiguration, sodass bspw. Green Threads und die Mark/Sweep-GC gemeinsam genutzt werden konnten.

Die ersten vier Features stellten jedoch kaum Ansprüche an die Modularisierungstechniken. Die Modularisierung war sehr gradlinig durchführbar und AspectC++ stellte die nötigen Sprachmittel vollständig zur Verfügung. Einzig Änderungen auf Smalltalk-Ebene und Änderungen an den Makefiles z. B. durch zusätzliche Compiler-Einstellungen mussten im Makefile gesondert berücksichtigt werden.

Besondere Anforderungen an die Modularisierungstechniken stellten erst die letzten vier Features. Besonders die Implementierung der Virtual Images und der markierten Integer stellten neue Anforderungen an die Modularisierung. Die Null-markierten Integer ließen sich letztendlich aufgrund der beschränkten Sprachmittel von C nicht geeignet modularisieren. Für die Eins-markierten Integer musste ein Mechanismus zum Ändern von Makros in VMADL eingeführt und für die Virtual Images sogar eine Definitionssprache für das in CSOM verwendete Objektmodell entworfen werden, um die Features geeignet modularisieren zu können. Dies führte letztendlich zum letzten und finalen Zweig `csom-classdl`. Die Details zur Anwendung von VMADL und zur zusätzlich entwickelten ClassDL werden im folgenden Abschnitt beschrieben.

4.5 Anwendung von VMADL auf CSOM

4.5.1 Herangehensweise und Verwendung von VMADL

Wie im letzten Abschnitt beschrieben, wurden die Feature-Implementierungen zuerst mithilfe der aspektorientierten Programmierung bzw. speziell AspectC++ überarbeitet, bevor sie in ein Service-Modul überführt wurden. AspectC++ ermöglichte zwar mit wenigen Einschränkungen bezogen auf CSOM-Besonderheiten den gewünschten Grad an Modularität und erlaubte es Features zum System hinzuzufügen, ohne bestehende Quelltext-Dateien zu verändern, aber die Architekturwahrnehmung ließ sich damit noch nicht wie gewünscht verbessern.

Daher wurden die einzelnen Features anschließend in Service-Module mit einer vollständigen, bidirektionalen Schnittstellenbeschreibung überführt. Mit den Sprachmitteln von VMADL war es nicht nur möglich die Schnittstellen geeignet zu erfassen, sondern außerdem die Interaktionen zwischen den Service-Modulen explizit darzustellen. Zusätzlich zur reinen Modularisierung wurden damit die Architekturaspekte der Service-Module bzw. der durch sie implementierten Features erfasst.

Ein wesentliches der für diese Fallstudie benannten Ziele ist die Entwicklung einer CSOM-Produktfamilie, mit der es möglich ist, verschiedene Features zu kombinieren und aus solch einer Konfiguration eine lauffähige Version von CSOM generieren zu können. Mit den auf AspectC++ basierenden CSOM-Implementierungen allein wäre dies nicht ohne Weiteres möglich gewesen, da in diesen nicht direkt angegeben wird, welche Advices sich auf welche Module beziehen. Mit VMADL wird dies mithilfe der `combine`-Konstrukte explizit. Die Interaktionen finden immer zwischen zwei benannten Service-Modulen statt. So ist es möglich, die für eine Konfiguration relevanten Interaktionen direkt zu bestimmen und in die Kompilierung einfließen zu lassen.

Wie oben erwähnt, war es darüber hinaus jedoch nötig, eine zusätzliche Abstraktion für die in CSOM verwendete OOP-Nachahmung zu definieren, um eine vollständige Modularisierung aller Features erreichen zu können. Die ClassDL genannte Sprache ist jedoch spezifisch für diese OOP-Nachahmung gedacht und kann bei der Verwendung von VMADL im Zusammenhang mit anderen Implementierungssprachen

durch diese ersetzt werden, wenn sie die von ClassDL bereitgestellten Intertyp-Definitionen ermöglichen.

Bevor die Modularisierung der Features mit VMADL vorgestellt wird, stellt der nächste Abschnitt ClassDL vor und gibt eine Einführung in ihre Konzepte, Syntax und Implementierung.

4.5.2 Featureorientierte Programmierung für CSOM

4.5.2.1 Motivation für eine Klassendefinitionssprache

Im Bereich, der featureorientierte Programmierung ist in einigen Sprachen [ALRS05b, Bat06] die Möglichkeit vorgesehen, Klassen nachträglich um Felder und Methoden zu erweitern, dabei aber die Implementierung für ein Feature lokal zu halten. Bei den verfügbaren Sprachen handelt es sich jedoch um Erweiterungen für objektorientierte Sprachen wie Java und C++. Für die Nachahmung der Objektorientierung in C lassen sich diese Werkzeuge nicht verwenden. Um eine vollständige Modularisierung in Service-Module zu erreichen, ist die Lokalität von nur für ein bestimmtes Service-Modul relevanter Funktionalität jedoch eine wichtige Voraussetzung.

Die *Class Definition Language* (ClassDL) bietet eine Notation zur Definition von Klassen und ersetzt damit die Makro-Konstruktionen, die in CSOM verwendet werden. Dies hat zwei Vorteile für die Implementierung von CSOM. Einerseits ermöglicht die Sprache eine größere Modularität und andererseits können viele komplexe Konstruktionen, die für die Nachahmung objektorientierter Programmierung mit C nötig sind, generiert werden, anstatt sie aufwendig und fehleranfällig von Hand schreiben zu müssen.

ClassDL wurde jedoch nur als zusätzliches Hilfsmittel entworfen, da die vorhandenen Werkzeuge mit dem genutzten Programmiermodell nicht ihren vollen Funktionsumfang entfalten können. Sollte VMADL in einer Umgebung eingesetzt werden, in der bereits entsprechende Sprachmittel für Modularisierung vorhanden sind, sollte diesen im Allgemeinen der Vorzug vor ClassDL gegeben werden, und nur wenn sich herausstellt, dass die vorhandenen Sprachmittel unzureichend sind, sollte eine Portierung von ClassDL in Erwägung gezogen werden.

4.5.2.2 Sprachumfang der ClassDL

ClassDL bietet ein Klassenmodell mit klassischer Einfachvererbung. Zusätzlich unterstützt es ein eingeschränktes Traits-Modell [CUL89, SDNB03] für horizontale Wiederverwendung von Verhalten außerhalb der Klassenhierarchie.

Die Definition von Klassen und Traits erfolgt wie im Quelltext 2 gezeigt. Die Syntax für die Definition von Feldern und Methoden entspricht der normalen C-Syntax, wie sie für die Definition einer C-Struktur oder Funktion verwendet wird. Für die Implementierung werden aus diesen Beschreibungen die C-Header-Dateien generiert und die Definitionen im Wesentlichen direkt übernommen. Somit können alle C-Datentypen und auch Schlüsselwörter wie `restrict` oder `const` verwendet werden. Nicht angegeben werden muss der Zeiger auf das Objekt, dieser wird automatisch generiert. Für Traits steht jedoch nur rudimentäre Unterstützung bereit. Die ursprünglich vorgeschlagenen Mechanismen zur expliziten Konfliktlösung [SDNB03] und bspw. die Definition von zusätzlichen Zuständen in den Traits [BDNW07] werden nicht unterstützt.

Da das nachgeahmte Objektmodell in C keine Sichtbarkeiten unterstützt, ist dies in ClassDL ebenfalls nicht berücksichtigt.

```
01      trait MyTrait {
02           pFoo               foo
03           pBar               bar
04
05           pComb              combine()
06      }
07
08      class MyClass : BaseClass, MyTrait {
09           pFoo               foo
10           pBar               bar
11           pIntObj    count
12
13           void               inc(int byValue)
14           void               dec(int byValue)
15           pIntObj            getCount()
16      }
17
18      refine MyClass {
19           pMyClass   next    { before count }
20           void               printList(const char* title)
21      }
```

Quelltext 2 Beispiel für eine Klassendefinition mit ClassDL

Um die von der featureorientierten Programmierung bekannte Modularität zu erreichen, ist es über das im Beispiel in Zeile 18 verwendete `refine`-Konstrukt möglich, Felder oder Methoden zu Klassen bzw. Traits hinzuzufügen. Da die generierte Implementierung aus Strukturen besteht und das Speicherlayout nicht zur Laufzeit ermittelt wird, kann es nötig sein, direkten Einfluss auf die Anordnung von Feldern nehmen zu müssen. Das Beispiel zeigt in Zeile 19, wie dies mithilfe eines Prädikats möglich ist. Darüber kann angegeben werden, dass ein Feld vor einem bestimmten anderen Feld angeordnet sein muss.

Eine Besonderheit von ClassDL ist die Handhabung von in abgeleiteten Klassen überschriebenen Methoden. In Sprachen wie C++ ist es üblich, die in der Implementierung überschriebene Methode noch einmal mit in der Header-Datei der abgeleiteten Klasse aufzuführen. Da dies jedoch aus Sicht einer Schnittstellendefinition redundant ist und nur ein Implementierungsdetail darstellt, wurde in ClassDL darauf verzichtet, diese Methoden noch einmal aufführen zu müssen. Eine ClassDL-Klassendefinition beschreibt somit nur die öffentlichen Methoden und Felder, die die Schnittstelle einer Klasse darstellen, ohne Implementierungsdetails preiszugeben.

4.5.2.3 Implementierung

Als eine typische externe domänenspezifische Sprache (DSL) [Fow05] wird aus den ClassDL-Definitionen Quelltext generiert, in diesem Fall C, der dann in das normale Programm eingebunden wird. Der ClassDL-Compiler selbst ist in den VMADL-Compiler eingebettet und ebenfalls mit ANTLR und Java realisiert.

Für die Implementierung werden C-Strukturdefinitionen zum einen für das Speicherlayout der Felder und zum anderen für die Funktionszeiger der virtuellen Methoden (die VTable) generiert. Der Code zum Initialisieren der VTable wird ebenfalls generiert. Hierbei besteht jedoch die Schwierigkeit, die korrekte Initialisierung zu generieren.

Wie bereits erwähnt, beschreibt eine Klassendefinition nur die Schnittstellensicht auf eine Klasse und in abgeleiteten Klassen müssen Methoden, die überschrieben werden, nicht noch einmal aufgeführt werden. Dies hat für die Implementierung des ClassDL-Compilers jedoch den Nachteil, dass die C-Implementierung analysiert werden muss, um die tatsächlich implementierten Funktionen zu ermitteln. Dazu wird ein rudimentärer C-Parser verwendet, der einen Teil der C-Grammatik implementiert und darauf spezialisiert ist, die Funktionsdefinitionen zu extrahieren.

Für die Implementierung der Traits mussten ebenfalls einige Einschränkungen vorgenommen werden. Um einen direkten Zugriff auf die Felder des Objekts zu ermöglichen, ist es Voraussetzung, dass alle Klassen, die einen Trait benutzen, ein identisches Speicherlayout für den Bereich haben, der vom Trait genutzt werden soll. Andernfalls ist nur die Verwendung von Zugriffsmethoden möglich.

Ein weiteres Problem sind zirkuläre Abhängigkeiten zwischen Klassendefinitionen. Diese können auftreten, wenn eine Klasse Felder oder Methodenparameter vom Typ einer anderen Klasse hat. Dieses Problem lässt sich nur über die Verwendung von Vorausdeklarationen für die Typen lösen. Der Nachteil hierbei ist jedoch, dass dies eine noch engere Verzahnung mit dem VMADL-Compiler erfordert, um eine zyklusfreie Include-Kette zu erreichen. Für jedes Service-Modul werden vom ClassDL-Compiler zwei Header-Dateien generiert, eine mit den Vorausdeklarationen und eine mit den C-Strukturdefinitionen für die Klassen. Zusätzlich wird eine Implementierungsdatei für die VTable-Initialisierung angelegt.

4.5.3 Modularisierung und Modulinteraktionen

Dieser Abschnitt stellt die einzelnen Features in ihrer VMADL-Implementierung vor. Es wird dabei jeweils auf die besonderen Anforderungen an die Modularisierung eingegangen und gezeigt, wie VMADL für die Realisierung verwendet wurde. Hervorgehoben werden dabei vor allem die Besonderheiten der einzelnen Features und der verwendeten Implementierungsstrategie.

4.5.3.1 Mark/Sweep-Garbage-Collection

Die Modularisierung der Mark/Sweep-GC war bereits mit den Mitteln der aspektorientierten Programmierung nahezu vollständig möglich. Im Abschnitt 4.3.3.1 wurde sogar davon gesprochen, dass die Modularisierung vollständig war. Bei genauer Betrachtung stellt sich jedoch heraus, dass die Implementierung der GC für eine vollständige Modularisierung weiter geändert werden müsste. Das benötigte Markierungsfeld ist ursprünglich in den Objekten selbst abgelegt, sodass das *VMObjects*-Service-Modul geändert werden musste, um die Felddefinition hinzuzufügen. Es gäbe jedoch auch alternative Implementierungsstrategien, die es ermöglichen würden, dieses Feld aus dem Objektlayout herauszunehmen und als unabhängiges Feld den Objekten im Speicher voranzustellen. So wäre dieses Feld nur in den Verwaltungsstrukturen der GC bekannt und der Rest des Systems wäre nicht beeinflusst.

Um jedoch die Implementierung der GC nicht ändern zu müssen, wurde stattdessen die von ClassDL angebotene `refine`-Definition verwendet, um die bestehende Klasse durch das *GC*-Service-Modul beeinflussen zu können, ohne den Quelltext ändern zu müssen. In der im Quelltext 3 dargestellten Implementierung äußert sich dies durch die Definition in den Zeilen 22-24. Dort wird angegeben, dass die Klasse `VMObject` das zusätzliche Feld `gc_field` bekommt. Zusätzlich ist angegeben, dass dieses Feld vor dem `fields[0]`-Array eingefügt werden soll. Dies ist notwendig, da alle Felder, die danach angeordnet sind, von der Smalltalk-Ebene angesprochen werden können. Dies ist jedoch für das `gc_field` nicht erwünscht und würde zu Problemen führen, da der Wert kein Smalltalk-Objekt ist. Die mit `expose` angebotenen Pointcuts wurden für das *Native-Threads*-Service-Modul und das Service-Modul für Eins-markierte Integer hinzugefügt. Diese Module benötigen diese Ereignisse, um ihre Funktionalität implementieren zu können. Die Pointcuts selbst werden jedoch vom *Mark/Sweep-GC*-Service-Modul definiert. Im Falle einer Änderung in der GC-Implementierung müssten die Pointcut-Definitionen entsprechend angepasst werden, um sicherzustellen, dass

```
01      service GCMarkSweep {
02           require Memory;
03           require VMObjects;
04           require VM;
05
06           void*      gc_allocate(size_t size);
07           void       gc_free(void* ptr);
08           void       gc_collect();
09           void       gc_mark_reachable_stack_objects();
10           void       gc_start_uninterruptable_allocation();
11           void       gc_end_uninterruptable_allocation();
12
13           expose {
14              pointcut mark_object(void* _self) =
15                      execution("void gc_mark_object(...)") && args(_self);
16              pointcut split_and_reserve_entry() =
17                      execution("% split_and_reserve_entry(...)");
18              pointcut reserve_and_get_entry() =
19                      execution("% reserve_and_get_entry(...)");
20           }
21
22           refine VMObject {
23              int     gc_field { before fields[0] }
24           }
25      }
26
27      combine GCMarkSweep, Memory {
28           advice execution("void* memory_allocate(unsigned int)") && args(size)
29            : around(unsigned int size) {
30              *tjp->result() = gc_allocate(size);
31           }
32
33           advice execution("void* memory_allocate_object(unsigned int)")
34            && args(size) : around(unsigned int size) {
35              *tjp->result() = gc_allocate_object(size);
36           }
37
38           advice execution("void memory_free(void*)") && args(ptr)
39            : around(void* ptr) {
40              gc_free(ptr);
41           }
42      }
43
44      combine GCMarkSweep, VMObjects {
45           advice execution(VMObjects::initializer()) : around() {
46              gc_start_uninterruptable_allocation();
47              tjp->proceed();
48              gc_end_uninterruptable_allocation();
49           }
50      }
```

Quelltext 3 Definition und Interaktionen des Service-Moduls Mark/Sweep

ihre Semantik erhalten bleibt. Die davon abhängigen Module müssen hingegen nicht angepasst werden, da sie sich weiterhin auf die definierte Schnittstelle verlassen können.

Zusätzlich zur Beschreibung der Schnittstelle des Service-Moduls werden außerdem die Interaktionen mit anderen Service-Modulen dargestellt. Wesentlich sind hier die Advices zum Überschreiben der Methoden des *Memory-Management*-Service-Moduls und die Absicherung der Objektinitialisierung im *VMObjects*-Service-Modul. Sie betten das *GC*-Service-Modul erst in die VM ein und beschreiben die Punkte, an denen eine direkte Interaktion mit anderen Modulen stattfindet. Darüber hinaus gibt es noch Interaktionen mit dem *VM*-Service-Modul, um die Verarbeitung von Kommandozeilenparametern zu realisieren. Dies ist hier jedoch nicht dargestellt.

4.5.3.2 Reference-Counting-Garbage-Collection

Ähnlich wie die Mark/Sweep-GC verwendet die Reference-Counting-GC die ClassDL dazu ein Feld zur Klasse `VMObject` hinzuzufügen. Der Referenzzähler hätte dabei theoretisch auch direkt im *GC*-Service-Modul implementiert werden können, indem er

```
01   service InterpreterThreaded {
02
03        ...
04
05        refine VMMethod {
06             intptr_t  get_number_of_tcodes()
07             void*     get_tcode(size_t)
08             void      set_tcode(size_t, void*)
09        }
10   }
11
12   combine InterpreterThreaded, VMObjects {
13
14        ...
15
16        advice execution("_VMMethod*
17                  VMMethod_assemble(_method_generation_context*)")
18             && args(mgenc) : after(_method_generation_context* mgenc) {
19             tcode_dynamic_translate(*(pVMMethod*)tjp->result());
20        }
21   }
```

Quelltext 4 Redefinition von VMMethod für das Threaded-Interpretation-Service-Modul

jeweils vor dem Objekt angeordnet wird und dann nur im *GC*-Service-Modul bekannt ist. Dies hätte allerdings ebenfalls eine Änderung an der GC-Implementierung erfordert.

Bei einem Blick auf die VMADL-Definition für die Reference-Counting-GC lässt sich sehr schnell ein Nachteil des Verfahrens gegenüber der Mark/Sweep-GC erkennen. Es sind deutlich mehr Interaktionen mit anderen Service-Modulen notwendig, da bei allen Zuweisungen die Referenzzähler aktualisiert werden müssen. Betroffen sind davon letztendlich alle Service-Module, die mit Objekten arbeiten, die durch die GC verwaltet werden. Solange nur das Standardsystem und die Reference-Counting-GC betrachtet werden, betrifft dies die Service-Module *Interpreter*, *VM* und *VMObjects*. Daran ist jedoch auch gleich der große Vorteil von VMADL zu erkennen. Da die Interaktionen als eigenes Konzept im Quelltext einen eigenen Platz haben, lassen sie sich sehr schnell auffinden und analysieren.

Abgesehen von den Interaktionen zur Aktualisierung des Referenzzählers verhält sich die Reference-Counting-GC jedoch wie die Mark/Sweep-GC und überschreibt die Funktionen zur Speicherallokation des *Memory-Management*-Service-Moduls mit den eigenen Routinen.

4.5.3.3 Threaded Interpretation

Das Service-Modul zur Implementierung der Threaded Interpretation nutzt VMADL und AOP auf vielfältigere Weise, als dies die GC-Implementierungen bisher taten. Auch hier gibt es Interaktionen mit dem *VM*-Service-Modul, um eine Verarbeitung der Kommandozeilenparameter zu erreichen. Ähnlich wie die Memory-Management-Funktionen wird auch hier die Funktionalität der Funktion `Interpreter_start` überschrieben. Darüber hinaus werden jedoch noch verschiedene andere Mechanismen ausgenutzt, um die optimierte Form der Interpretation ermöglichen zu können.

Im Quelltext 4 wird dargestellt, wie die Klasse `VMMethod` für die Integration des Threaded Codes erweitert wird. Es wird außerdem gezeigt, wie die der Threaded Code automatisch nach dem Zusammensetzen der Methode erzeugt wird. Die erzeugte Code wird direkt in der Methode ablegen und lässt sich über die mit ClassDL definierten Methoden abfragen und setzen.

Nicht dargestellt werden dagegen die Interaktionen mit den Service-Modulen *Interpreter*, *Bytecode* und *VMObjects*. Dort werden alle Zugriffe mit Bezug auf den ursprünglichen Bytecode so abgeändert, dass sie den Threaded Code bzw. die speziell angepassten Implementierungen verwenden. So wird bspw. im Interpreter die Funktion

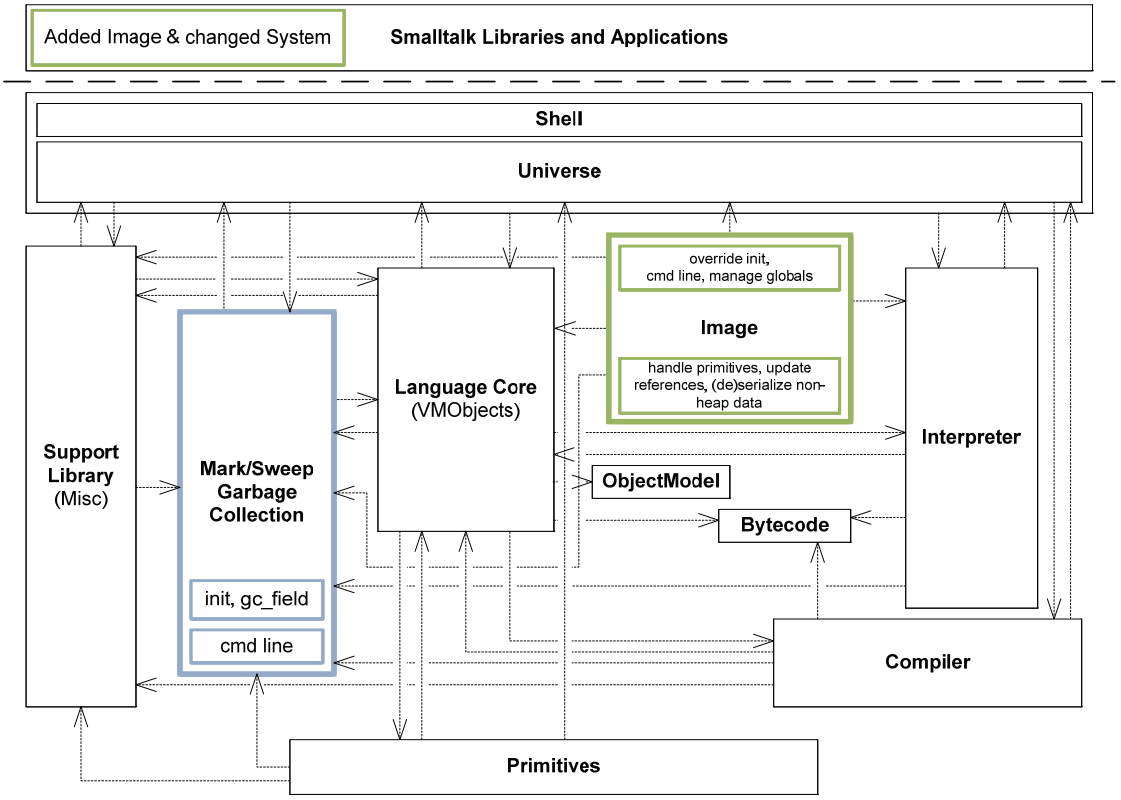


Abbildung 11 Architektur der mit VMADL modularisierten CSOM in einer Konfiguration mit Mark/Sweep-GC und Virtual Images (UML-Klassendiagramm mit Veranschaulichung von Änderungen)

`Interpreter_push_new_frame` durch eine spezielle Routine ersetzt und nach der Kompilierung einer Methode die dynamische Übersetzung des Bytecodes in Threaded Code angestoßen.

4.5.3.4 Smalltalk Virtual Images

Die Schnittstelle dieses Service-Moduls ist im Gegensatz zu den anderen Modulen sehr umfangreich. Die bereitgestellte Schnittstelle wird jedoch nicht von anderen Service-Modulen genutzt. An dieser Stelle sollte bei einer Überarbeitung überlegt werden, ob Teile der Schnittstellenbeschreibung nicht besser in Form von Implementierungsschnittstellen, also .h-Dateien, beschrieben werden sollten. Dies würde den Schnittstellenumfang deutlich reduzieren. Übrig bleiben würden nur solche Definitionen, die für die Interaktion mit anderen Service-Modulen relevant wären.

Ein Teil des Schnittstellenumfangs ist durch die Definition einer eigenen Klasse und der Anpassung von `VMObject`, `Hashmap` und `HashmapElement` verursacht. Die Anpassungen wurden nötig, um einerseits Daten, die nicht auf dem GC-Heap liegen, serialisieren zu können und andererseits, um nach dem Laden des Speicherabbilds die Objektreferenzen in allen Objekten zu korrigieren. Dies ist nötig, da die Implementierung direkt auf der Mark/Sweep-GC aufsetzt und letztendlich den von ihr verwalteten Heap in einer Datei sichert, um ihn später daraus wieder herzustellen.

Für den Ladevorgang wird über eine Interaktionsbeschreibung mit dem *VM*-Service-Modul der komplette Initialisierungsvorgang abgeändert. Wenn ein Abbild geladen werden soll, wird nach der Initialisierung einiger grundlegender Strukturen der Ladevorgang gestartet, anstatt den normalen Initialisierungsprozess fortfahren zu lassen. Nach dem Laden wird dann auch gleich die Ausführung des Smalltalk-Programms an der Stelle, an der es vorher beendet wurde, fortgesetzt. Dazu ist es

jedoch nötig Symbole global zu speichern, damit sie serialisiert werden können. Klassen mit Primitiven müssen registriert werden, damit eine Anpassung der Funktionszeiger auf die Primitivroutinen möglich ist.

Der größte Teil der Komplexität verbirgt sich jedoch in der Implementierung selbst. Es sind eher wenig Interaktionen mit anderen Service-Modulen notwendig, da auch zur Laufzeit kaum spezielle Anpassungen nötig werden und sich die wesentliche Funktionalität auf den Startprozess bezieht. Die Fähigkeit das laufende System zu persistieren wird hingegen über eine Änderung an der Smalltalk-Klassenbibliothek verfügbar gemacht. Diese wird hier jedoch nicht weiter betrachtet, da sie nicht mit VMADL erfasst werden kann.

Abbildung 11 zeigt die entstandene Konfiguration der mit VMADL implementierten CSOM mit der Mark/Sweep-GC und dem *Image*-Service-Modul. Im Vergleich mit Abbildung 9 ist zusehen, dass das *Memory-Management*-Service-Modul komlett durch das Service-Modul der GC ersetzt wurde und alle Änderungen, die in der ursprünglichen Implementierung über das System verstreut waren, im Service-Modul implementiert werden konnten. Dies ist ebenfalls bei der Darstellung des *Image*-Service-Moduls zu sehen, welches ebenfalls die ursprünglich über das System verteilten Änderungen darstellt. Im Unterschied zur Abbildung 9 sind auch *ObjectModel* und *Bytecode* als eigenständige Service-Module repräsentiert, da dies die Modularisierung der Threaded Interpretation und der Eins-markierten Integer vereinfacht.

4.5.3.5 Green Threads

Wie bereits erwähnt, wurde ein Großteil der Green-Threads-Implementierung in Smalltalk realisiert, um den C-Anteil der Implementierung so gering wie möglich halten zu können. Dies hat für das Service-Modul den Nebeneffekt, dass es relativ kompakt ist. Zudem zeigt es, wie Smalltalk-Klassen mit Primitiven erweitert werden, um Aufgaben auf der C-Ebene realisieren zu können. Speziell für die Modularisierung ist hier die Registrierung der Primitiven über eine Interaktion mit dem *Primitives*-Service-Modul realisiert.

Der Teil des Service-Moduls mit Bezug auf andere Service-Module ist sehr einfach gehalten. Die Shell wird so manipuliert, dass sie Green Threading nutzt und der Interpreter wird über ein von ihm angebotenes Event mit dem Scheduler verknüpft, damit es möglich wird, die Ausführung regelmäßig zu unterbrechen und eine Scheduling-Entscheidung zu treffen.

Etwas mehr Komplexität kommt im Zusammenhang mit den verschiedenen GC-Implementierungen hinzu. Für die Mark/Sweep-GC wird bspw. nach der Mark-Phase zusätzlich noch das lokale Root-Set des Moduls verarbeitet. Da der Scheduler eine Liste mit den Threads hält, die momentan im System sind, diese aber sonst nicht referenziert werden, müssen sie extra markiert werden, damit die GC sie nicht in der Sweep-Phase freigibt. Für die Reference-Counting-GC müssen im Gegenzug die verschiedenen Zuweisungen zur Liste der Threads und zum aktuellen Thread Pointer erfasst werden, damit der Referenzzähler immer den korrekten Wert hat.

4.5.3.6 Native Threads

Im Vergleich zur Green-Threading-Implementierung ist das *Native-Thread*-Service-Modul deutlich umfangreicher. Hier wurde der größte Teil der Funktionalität direkt in C implementiert, da die Pthreads-Bibliothek zur Verwendung von Betriebssystem-Threads eingesetzt wurde. Entsprechend wurden mithilfe von ClassDL Klassen für Mutexe, Signale und Threads definiert. Für die Smalltalk-Ebene gibt es eine passende

```
01  service TaggedIntOne {
02       require VM;
03       require VMObjects;
04
05       ...
06
07       replace ObjectModel.Basics {
08
09            #define INSTANCE_POINTER_ACCESS(O) \
10                 ({ (VMINTEGER_IS_TAGGED(O) ? VMInteger_Global_Box() : O); })
11
12            #define SEND(O,M,...) \
13                 ({ typeof(O)_Org = (typeof(O))(O); \
14                 typeof(_Org)_O = (typeof(_Org))(INSTANCE_POINTER_ACCESS(_Org));\
15                 (_O->_vtable->M(_Org, ##__VA_ARGS__)); })
16
17            ...
18
19            pVMInteger VMInteger_Global_Box(void);
20
21            #define VMINTEGER_TAG_BIT 1
22            #define VMINTEGER_IS_TAGGED(V) ( ((int32_t)V & VMINTEGER_TAG_BIT) )
23
24       }
25  }
```

Quelltext 5 Redefinition einer Service-Modul-Schnittstelle

Klassenbibliothek und die Verbindung zwischen den beiden Ebenen wird durch Primitiven bewerkstelligt. Daher findet sich auch in diesem Service-Modul ein `combine`, welches die Interaktionen mit dem *Primitives*-Service-Modul definiert.

Die Implementierungsidee für die nativen Threads war es, den Interpreter in jedem Thread lokal auszuführen, um echte Nebenläufigkeit zu erreichen. Dementsprechend finden sich in den Interaktionsdefinitionen Advices, die den Zugriff auf verschiedene globale Variablen durch Thread-lokale Zugriffe ersetzen.

Eine deutlich erhöhte Komplexität kommt jedoch erst durch die Kombinationsmöglichkeit mit der Mark/Sweep-GC hinzu. Um diese threadsicher machen zu können, muss sichergestellt werden, dass es nicht zu konkurrierenden Änderungen im Speicher kommt. Die dazu gewählte Implementierungsstrategie ist ein einfacher *Stop-The-World*-Ansatz. Zusätzlich zu den eigentlichen Anwendungsthreads wird ein Thread für die Ausführung der Mark- und Sweep-Phasen verwendet. Diese werden ausgeführt, nachdem alle Anwendungsthreads einen sicheren Haltepunkt erreicht haben und auf die Beendigung der GC-Phasen warten. Implementiert ist dies momentan komplett im Interaktionsteil zwischen `NativeThreads` und `GCMarkSweep`. Für eine Verbesserung des Architekturverständnisses beim Lesen dieser Definitionen sollten die Implementierungsdetails jedoch besser in ein eigenes Implementierungsmodul ausgelagert werden, damit in den Interaktionen vorwiegend einzelne Funktionsaufrufe sichtbar sind.

4.5.3.7 Eins-markierte Integer

Für die Implementierung dieses Moduls musste ein besonderer Sprachmechanismus von VMADL eingesetzt werden. Da in C keine Möglichkeit vorgesehen ist, Präprozessoranweisungen nachträglich zu redefinieren, musste von der in Quelltext 5 dargestellten Konstruktion zum Ersetzen einer Schnittstellendefinition Gebrauch gemacht werden. Wie in Abschnitt 3.2 beschrieben, können Schnittstellendefinitionen durch benannte Bereiche strukturiert werden. Diese dienen neben der Erhöhung der Übersichtlichkeit auch als Ziele von Ersetzungen durch andere Service-Module. Das *TaggedIntOne*-Service-Modul nutzt dies wie in Zeile 7 dargestellt, um unter anderem das `SEND`-Makro so anzupassen, dass der Nachrichtenversand auch an markierte

Integer möglich ist, indem eine globale Instanz eines Integer-Objekts mit passender VTable bereitgehalten wird.

Die Interaktionen mit dem *Mark/Sweep-GC*- und dem *VMObjects*-Service-Modul beschränken sich auf kleinere Anpassungen. In der Klasse `VMFrame` wird bspw. eine Optimierung für das Setzen des Stack-Pointers verwendet, die nicht mit markierten Integern kompatibel ist und daher überschrieben werden muss. Weiterhin darf eine der vorhandenen GC-Implementierungen natürlich nicht versuchen, Integer als Speicheradressen zu interpretieren, da dies zu fehlerhaftem Verhalten führen würde.

4.5.3.8 Null-markierte Integer

Wie bereits in Abschnitt 4.3.3.8 diskutiert, ist die Implementierung der Null-markierten Integer mit den vorhandenen Sprachmitteln nicht sinnvoll möglich, da in C kein Sprachmittel zur Verfügung steht, um die Dereferenzierungsaktion für Pointer zu überschreiben. In C++ wäre hier eventuell die Möglichkeit gegeben, mittels einer Operatorüberladung das Dereferenzieren anzupassen, aber in der vorliegenden Implementierung gibt es keine Möglichkeit die über die verschiedenen Module verstreuten Änderungen zu modularisieren. Die verwendeten Makros lassen sich auch nicht durch die Sprachmittel von AspectC++ in ein einzelnes Modul lokalisieren und so musste auf eine Implementierung Null-markierter Integer in Form eines Service-Moduls verzichtet werden.

Es gibt zwar Implementierungsstrategien, die ein Anpassen der Derefenzierung vermeiden würden, aber diese wurden nicht verwendet, da einerseits die Implementierung hätte geändert werden müssen, was jedoch nicht gewünscht war und andererseits diese Strategien vorher noch nicht getestet wurden und eventuell mit Performancenachteilen verbunden sind. Zur Vermeidung der Dereferenzierung wäre es z. B. möglich entweder komplett auf das Alignment im Speicher zu verzichten und die Objekte so abzulegen, dass der mit Eins markierte Zeiger an die korrekte Stelle zeigt. Eine andere Alternative wäre die eigentlichen Objektdaten korrekt im Speicher anzuordnen, aber einen Alignmentheader vor das Objekt zu legen, sodass der Objektzeiger korrekt ist und die Programmierung in C weiter möglich ist. Dies hätte aber entweder Speicherverschwendung zur Folge oder würde komplexe Konstruktionen erfordern, um den Alignmentspeicher sinnvoll nutzen zu können. Insgesamt scheint jedoch eine Lösung des Problems in C nicht praktikabel. Eine Realisierung durch Operator-Überladung in C++ erscheint hier sinnvoller, war aber mit der vorliegenden CSOM nicht implementierbar.

4.5.4 Konventionen und Compiler-Integration

Da CSOM in C geschrieben ist, wird die Kombination aus `.c`- und `.h`-Datei als Implementierungsmodul bezeichnet. Die Verwendung einer `.h`-Datei ist dabei jedoch optional. Die VMADL-Service-Module bestehen aus einem Ordner, der die Implementierungsmodule enthält und der dazugehörigen `.vmadl`-Datei. Die `.vmadl`-Datei muss dabei denselben Namen wie der Ordner haben, damit der VMADL-Compiler die Implementierung finden kann.

Wie bereits in Abschnitt 3.4.2 erwähnt, wurden einige Festlegungen für die Integration des C-Quelltextes mit VMADL getroffen. Da für die Kompilierung ein Standard-Compiler verwendet werden sollte, war es notwendig, die Implementierungsmodule in geeigneter Weise mit den Service-Modulen zu verknüpfen. Aus den VMADL-Dateien werden unter anderem C-Header-Dateien generiert. Die Header-Dateien erhalten den Namen „`module.ModuleName`“. Somit können sie in den

Implementierungsmodulen über bspw. `#include <module.VM>` referenziert werden. Darüber hinaus gilt, dass Implementierungsmodule keine Service-Module referenzieren dürfen, die nicht auch auf Architekturebene als Voraussetzung für das Service-Modul angegeben wurden. Um Konsistenzprobleme zu vermeiden, sollte sogar darauf verzichtet werden andere Module zu referenzieren. Es dürfen nur das eigene Service-Modul sowie dazugehörige Implementierungsmodule direkt referenziert werden.

Die gewünschte Werkzeugunterstützung für die Konfiguration einer Instanz der CSOM-Produktfamilie wird mit Makefiles bereitgestellt. Die für die Kompilierung verwendbaren Service-Module werden direkt in den Makefiles definiert.[4] Dort ist einerseits eine Liste aller Service-Module enthalten, die für eine lauffähige Konfiguration benötigt werden, andererseits sind dort auch die Definitionen für die optionalen Service-Module zu finden. Für jedes ist genau aufgeführt, welche zusätzlichen Angaben zur Kompilierung verwendet werden müssen, um diese automatisch in der ausgewählten Konfiguration berücksichtigen zu können.

4.5.5 ERGEBNISSE

Bei der Verwendung von VMADL hat sich besonders die Trennung der Interaktionsdefinitionen zwischen den Modulen von der Implementierung als vorteilhaft erwiesen. Einerseits wird damit die Architektur klarer, aber andererseits lassen sich somit die Lösungen für Probleme, die durch die Kombination von Modulen entstehen, sehr gezielt beschreiben und im Quelltext lokalisieren. Dies hat unter anderem bei der Kombination der nativen Threads mit der Mark/Sweep-GC sehr gut funktioniert.

Andererseits sind durch die Probleme, die während der Implementierung von CSOM mit VMADL aufgetreten sind, verschiedene Mechanismen zur Sprache hinzugekommen. Dazu zählen die Möglichkeit zur Umdefinition von Schnittstellenteilen und die extra für CSOM entworfene ClassDL. Wiederholt anzumerken ist jedoch, dass eine Umdefinition von Schnittstellen nur sehr eingeschränkt verwendet werden sollte, da dies im Allgemeinen problematisch ist, aber durch die Einschränkungen von C notwendig wurde. Außerdem kam die explizite Beschreibung von Service-Modulkombinationen hinzu, die angelehnt an andere ADLs eine von der Implementierung getrennte Beschreibung der Modulinteraktionen erlaubt. Mit ClassDL kamen die notwendigen Sprachmittel für eine Modularisierung der mithilfe der OOP-Nachahmung implementierten Klassen hinzu. ClassDL selbst kann zwar auch in anderem Kontext als zusammen mit CSOM verwendet werden, aber die angebotenen Mechanismen sind auch in anderen Sprachen wie FeatureC++ [ALRS05b] und teilweise auch in AspectJ [GL03] verfügbar, sodass es sinnvoll erscheint, diesen abhängig vom Anwendungsfall den Vorzug zu geben.

Im Rahmen der Fallstudie wurde ein Problem identifiziert, dass aber im Rahmen dieser Arbeit nicht gelöst wurde. Dabei handelt es sich um die Inkompatibilität von Teilen verschiedener Service-Modulkombinationen, die die Interaktion mit einem weiteren Service-Modul beschreiben. Für die Beschreibung einige Interaktionen wurden `around`-Advices ohne `proceed()` verwendet. Dies ist natürlich problematisch, wenn mehrere Service-Modulkombinationen einen solchen Advice auf den selben Joinpoints anwenden wollen. Prehofer hat im Rahmen der featureorientierten Programmierung für diesen Fall die Verwendung eines Adapters vorgeschlagen

[4] Das Makefile ist auf der CD unter `/source/CSOM/branches/smarr/vmadl/classdl/Makefile` abgelegt.

Make/Test/Bench	✓	✓	✓	✓	✓	MT	✓	✓	✓	✓	M	✓	✓	M	✓	✓	M	✓	✓	M	✓	✓	M	✓	✓	M	✓	✓	M	✓	✓	✓	✓
RefCounting-GC		✗			✗			✗			✗			✗			✗			✗			✗			✗			✗				
Mark/Sweep-GC			✗			✗			✗			✗			✗			✗			✗			✗			✗			✗	✗	✗	✗	✗
Native Threads				✗	✗	✗							✗	✗	✗							✗	✗	✗				✗	✗	✗				
Green Threads							✗	✗	✗							✗	✗	✗															
Eins-markierte Integer										✗	✗	✗	✗	✗	✗	✗	✗	✗							✗	✗	✗	✗	✗	✗		✗		✗
Threaded-Interpr.																			✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗			✗	✗
Virtual Images																														✗	✗	✗	✗

Tabelle 1 Übersicht der getesteten Kombinationen und ihrers Funktionsstatus

[Pre97]. In verschiedenen Aspektsprachen wäre auch eine Priorisierung der Advices möglich, welche jedoch nicht das eigentliche Problem löst. Daher müsste für VMADL überlegt werden, ob bspw. das `combine` so angepasst wird, dass mehr als zwei Service-Module an einer Service-Modulkombination teilnehmen können und in einem solchen Fall diese erweiteren Service-Modulkombinationen, die einfach Kombination ersetzen, um als Adapter zu dienen. Die vollständige Lösung dieses Problems soll jedoch nicht Teil dieser Arbeit sein. Einige dieser Advices ohne `proceed()` lassen sich jedoch auch durch eine Erweiterung der Service-Module um hierarchische Beziehungen im Sinne einer Vererbung vermeiden, aber auch dies ist nicht im Rahmen dieser Arbeit implementiert und bleibt ein Ausblick für Folgearbeiten.

Das Ziel die Architektur besser herauszuarbeiten und den Entwicklern direkt im Quelltext verfügbar zu machen, ist aber zumindest aus rein subjektiver Wahrnehmung gelungen. Die objektive Evaluierung wird in Abschnitt 5.3 betrachtet.

Neben diesem Ziel gab es aber auch die Anforderung, dass CSOM letztendlich als Produktfamilie betrachtet werden kann und mithilfe eines Konfigurationsmechanismus speziell für ein Einsatzgebiet optimierte Exemplare erstellt werden können. Die gewünschte Variabilität einer solchen Produktfamilie wurde bereits in Abbildung 5 auf Seite 26 mit einem Feature-Diagramm gezeigt. Tabelle 1 stellt die momentan tatsächliche realisierte Variabilität dar. Abgebildet sind die Konfigurationen und ihr Zustand in Bezug auf Kompilierbarkeit, erfolgreiche Testläufe und Ausführbarkeit der Benchmarks. Eine Konfiguration wird stets über eine Spalte angegeben. Die in den Zeilen angegebenen Service-Module sind Teil einer Konfiguration, wenn in der Spalte ein Kreuz gesetzt ist. Als Beispiel beschreibt die sechste Konfiguration eine CSOM-VM bestehend aus der Menge der Komponenten, die die Grundfunktionalität bereitstellen und den Komponenten Mark/Sweep-GC sowie Native-Threads. Diese Konfiguration kann, angegeben durch ein M im Spaltenkopf erfolgreich kompiliert werden. Zusätzlich laufen auch die Tests, was durch ein T vermerkt ist. Die Benchmarks laufen hingegen nicht erfolgreich. Wenn alle drei Aktionen mit einer Konfiguration möglich sind, wird ein einzelnes Häkchen angegeben. Die Konfigurationen, bei denen bereits die Kompilierung fehlschlägt, wurden nicht mit in die Tabelle aufgenommen.

Prinzipiell kann festgestellt werden, dass alle durch das Feature-Diagramm beschriebenen Konfigurationen möglich wären. Es müssten nur alle Kombination von jeweils zwei Service-Modulen mittels der Service-Modulekombinationen implementiert werden. Im Fall der Virtual Images müsste aber darüber hinaus die Implementierung von der Mark/Sweep-GC getrennt werden, was durchaus möglich wäre, um sie bspw. mit der Reference-Counting-GC verwenden zu können.

5 Evaluierung

5.1 Übersicht

Die an der CSOM-VM durchgeführte Fallstudie hat bereits demonstriert, dass VMADL Vorteile für die Entwicklung und Wartung einer VM bietet. In diesem Abschnitt sollen die bisher getroffenen Einschätzungen durch zusätzliche Untersuchungen untermauert werden. Dazu werden im ersten Teil der Untersuchung verschiedene Metriken auf die Implementierung angewendet. Ziel ist es, dabei die Einflüsse von VMADL auf die Wartbarkeit abschätzen zu können. Darüber hinaus soll ermittelt werden, wie die Modularität beeinflusst wird. Zu diesem Zweck werden die verschiedenen Stufen der Überarbeitung miteinander verglichen.

Im zweiten Teil der Untersuchung werden die Laufzeiteigenschaften der VM untersucht und ebenfalls mit den Eigenschaften der VMs der verschiedenen Überarbeitungsstufen verglichen, um ermitteln zu können, welche Änderungen für welche Einflüsse verantwortlich sind.

Der dritte Teil der Untersuchung umfasst den Entwurf und die Durchführung eines Experiments mit Studenten, um ermitteln zu können, wie sich VMADL auf die Wahrnehmung und Wartbarkeit einer VM auswirkt.

5.2 Untersuchung mittels verschiedener Metriken

5.2.1 Analyse des Quelltexts

5.2.1.1 Verwendete Metriken

Für die folgenden Betrachtungen gilt grundsätzlich, dass nur der C-Anteil der CSOM-VM betrachtet wurde. Es gibt zwar ebenfalls Änderungen an den Smalltalk-Teilen durch die implementierten Features, diese wurden jedoch nicht von der Verwendung von VMADL beeinflusst und sind somit nicht relevant. Es wurden daher bei allen Messungen nur die C-, AspectC++- und VMADL-Dateien untersucht. Als Grundlage für den Vergleich wurde die in allen Implementierungsvarianten vorhandene CSOM-Konfiguration mit Mark/Sweep-GC verwendet.

In Tabelle 2 werden die verwendeten Metriken definiert. Als wichtigste Metriken sind für diese Untersuchung LoC und LoCwS [Par92] anzusehen. Sie erfassen zwar nicht die Komplexität des Systems, sind aber ein gutes Maß, um die Änderungen zwischen den Implementierungsvarianten zu beschreiben und geben einen abstrakten Eindruck von dem Aufwand, der mit der Verwendung von VMADL verbunden ist. Darüber hinaus sind aber auch die Metriken auf den Systemstrukturen interessant. Daher werden die Änderungen an Dateien als Repräsentationen der Implementierungsmodule und Änderungen auf Funktions- bzw. Strukturebene betrachtet.[5]

5.2.1.2 Auswirkungen der Überarbeitungen und von VMADL

Wie bereits in Abschnitt 4.4 genauer betrachtet wurde, sind im Laufe der Entwicklung verschiedene Änderungen an CSOM vorgenommen worden, die auch die ursprünglichen Module betreffen. Abbildung 12 zeigt die Auswirkungen dieser Änderungen in Form der Änderungen in den Metriken LoC und LoCwS. Die absoluten Werte sind für die LoC im oberen Bereich und für die LoCwS im unteren Bereich des Diagramms mit an die Balken angetragen. Die Balken selbst sind immer bezogen auf die originale

[5] Die Rohdaten sind auf der CD unter `/Arbeit/Evaluierung/loc-stats/` zu finden.

LoC	**Lines of Code** Dies ist die einfachste Metrik und ist definiert über die Anzahl der Zeilen einer Datei. Berechnet wird sie über die Anzahl der Zeilenumbrüche einer Datei plus eins. Sie schließt damit ausdrücklich sowohl Kommentare, als auch Leerzeilen ein. Die Betrachtung von Kommentaren ist sinnvoll, da sie ein wesentlicher Bestandteil des Programmes sind und entsprechend auch Wartungsaufwand bedeuten. Ähnliches gilt für Leerzeilen. Diese dienen einerseits der visuellen Unterstützung des Programmierers und sind in ihrer Gestaltung oft in Richtlinien vorgegeben. Damit haben auch sie Auswirkungen auf den Wartungsaufwand einer Software.
LoCwS	**Lines of Code with Statements** Diese Messgröße wird auch als *Physical Source Lines of Code* [Par92] bezeichnet. Die hier verwendete Definition stimmt mit der von Robert E. Park überein. Berechnet wird sie hier als Teilmenge von LoC. Es werden dabei alle Zeilen in die Zählung eingeschlossen, die keine Leerzeilen sind und nicht ausschließlich aus Kommentar bestehen.
Dateien	**Neue und geänderte Dateien** Diese Größe repräsentiert die beeinflussten Implementierungsmodule und erlaubt damit eine Rückschluss auf die Modularität bzw. Lokalität von Änderungen und implementierten Features.
Funktionen und Strukturen	**Geänderte Funktionen und Strukturen** Mit dieser Metrik kann die Modularität auf Submodul-Ebene bestimmt und bewertet werden, indem überprüft wird, ob eine Implementierung bestehende Funktionalität oder Definitionen anpassen muss. Dazu wird ermittelt, ob Änderungen am Quelltext vorgenommen wurden.

Tabelle 2 Definition der verwendeten Metriken

CSOM. Mit dieser Darstellung soll ein Bild vom Umfang der Änderungen zwischen den Versionen vermittelt werden. So wurde mit der ersten Überarbeitungsphase die Anzahl der Quelltextzeilen etwas reduziert, aber mit der Umstellung auf C++-Kompatibilität und anschließend auf AspectC++, sind jeweils wieder einige hinzugekommen.

Insgesamt können die Änderungen als minimal in Bezug auf die Komplexität des Gesamtsystems angesehen werden. Allerdings zeigt diese Darstellung das Haupt-

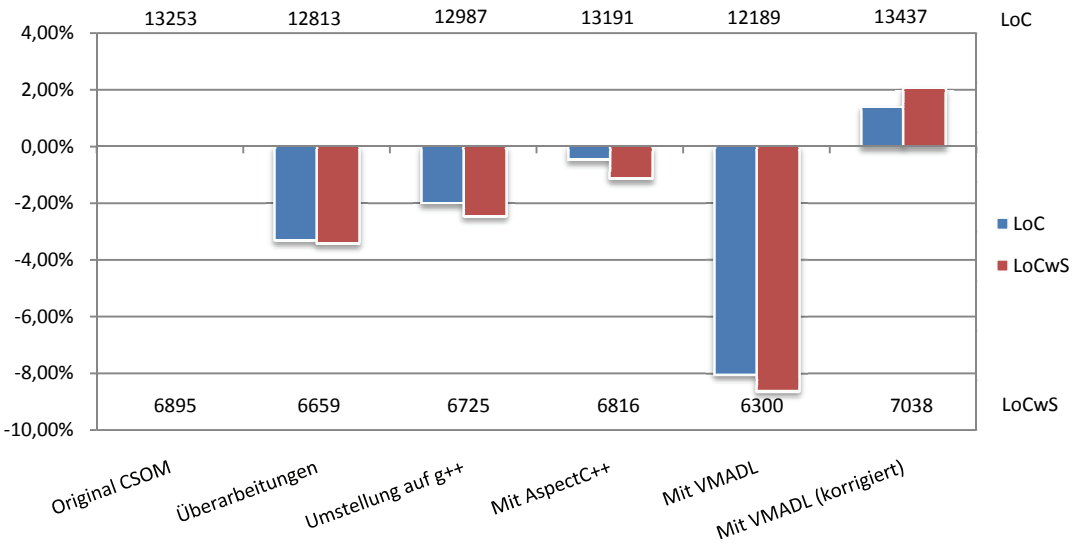


Abbildung 12 Auswirkungen der Überarbeitung auf die Lines of Code

problem, bei der Bewertung der Änderungen im Zug von VMADL, auf. Da VMADL für die Modularisierung auf das Vorhandensein von FOP-Ansätzen in der Implementierungssprache vertraut, musste mit ClassDL ein entsprechender Sprachaufsatz für die in CSOM verwendete OOP-Nachahmung eingeführt werden. Dadurch werden jedoch die tatsächlichen Auswirkungen von VMADL auf die Gesamtgröße des Quelltexts verschleiert. Um abschätzen zu können, welcher Aufwand mit der Verwendung von VMADL verbunden ist, muss der Einfluss, der durch ClassDL verursacht wird, herausgerechnet werden. Das Ergebnis ist zwar rein hypothetisch, erlaubt aber eine bessere Einschätzung von VMADL. Dies ist relevant, da die Unabhängigkeit von einer speziellen Implementierungssprache ein Ziel beim Entwurf von VMADL war.

Durch die Verwendung von ClassDL wurde es, wie in Abschnitt 4.5.2.3 beschrieben, möglich die nötigen Strukturbeschreibungen für das Objektlayout und die Methodentabellen sowie die Initialisierung der Methodentabellen zu generieren. Die eigentliche Beschreibung dieser Strukturen mit ClassDL benötigt im Vergleich mit der C-Implementierung deutlich weniger Quelltext. Um ein korrektes Maß für die durch VMADL zusätzlich verursachten Quelltextzeilen zu bekommen, muss der Unterschied den ClassDL verursacht genau ermittelt werden. Dazu wurde die Anzahl der Quelltextzeilen, die Strukturbeschreibungen und die Initialisierungsroutinen in der AspectC++-Variante benötigen ermittelt, da dies genau der Quelltext ist, der durch die Verwendung von ClassDL eingespart wurde. Von dieser Zahl wurde anschließend die Anzahl der Zeilen der ClassDL-Definition in der VMADL-Variante abgezogen, um genau den Unterschied den ClassDL verursacht zu erhalten. Nach der Anwendung dieses Korrekturwerts ergibt sich, dass durch die Verwendung von ClassDL 1248 LoC bzw. 738 LoCwS eingespart wurden. Dies bedeutet im Umkehrschluss, dass VMADL durch die strukturierenden Sprachmittel und die erhöhte Modularisierung zusätzliche 184 LoC (1,4%) bzw. 143 LoCwS (2,1%) verursacht.

5.2.1.3 Bewertung der Modularität

Bei dieser Untersuchung wurden die Änderungen, die an bestehenden Modulen zur Implementierung eines Features vorgenommen werden müssen, untersucht. Dazu wurden die verschiedenen Implementierungsstrategien einzeln betrachtet, um diese Modifikationen zu ermitteln. Für die Untersuchung wurden jeweils die Basis-Version von CSOM mit den Versionen mit Mark/Sweep-GC, Reference-Counting-GC, Green Threading und Native Threading sowie die Version mit Mark/Sweep-GC mit den Versionen mit zusätzlicher Unterstützung für Virtual Images, Eins-markierte Integer und Threaded Interpretation verglichen. Somit konnte jeweils ermittelt werden, wie viele Quelltextzeilen für ein spezielles Feature bearbeitet wurden und auf wie viele Dateien und Funktionen bzw. Strukturdefinitionen sich dies auswirkte.

In Abbildung 13 sind die Ergebnisse dargestellt. Die Daten der einzelnen Features wurden dabei aufsummiert und getrennt für die verwendeten Implementierungsstrategien dargestellt. Die Daten für die einzelnen Features variieren durch die unterschiedlichen Charakteristiken der einzelnen Implementierungen zwar leicht, aber zeigen im Wesentlichen die gleichen Eigenschaften. Auf der Primärachse links ist die Summe der neu für die Features hinzugefügten Quelltextzeilen abgetragen. Die dazugehörenden Balken sind im Hintergrund dunkelblau dargestellt. Im Vordergrund sind die restlichen Daten dargestellt, die entsprechend auf der Sekundärachse rechts abgetragen wurden. Gut zu erkennen ist der Anstieg der neu hinzugefügten Quelltextzeilen bei der Verwendung von AspectC++. Dies ist mit der Verlagerung der Implementierung aus den Modulen, in denen sie ursprünglich implementiert waren, in die

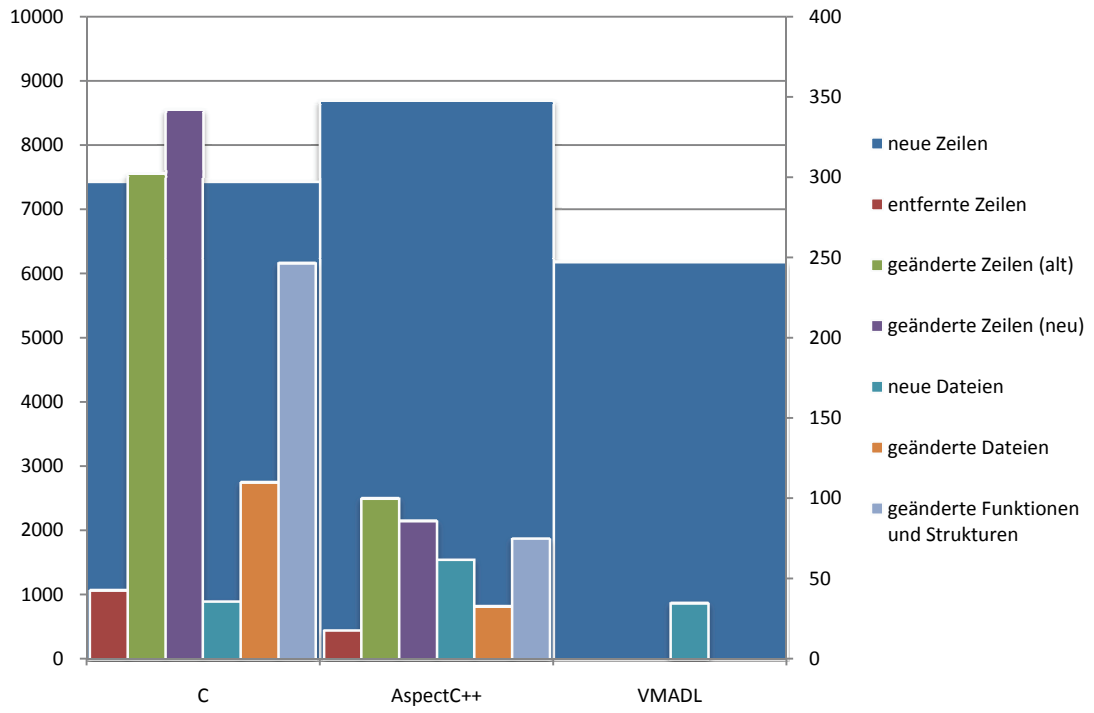


Abbildung 13 Änderungen durch Features am Quelltext

speziell für die Implementierung gedachten Module zu erklären. Bei VMADL ist dieser Anstieg nicht zu sehen, da hier die Vorteile von ClassDL ebenfalls noch hineinspielen, welche den Umfang des benötigten Quelltexts, wie bereits im vorhergehenden Abschnitt gezeigt, insgesamt reduzieren. Der hypothetische Wert von VMADL ohne ClassDL würde etwas über dem AspectC++-Werts liegen, da die strukturierenden Elemente von VMADL etwas umfangreicher sind, als die von AspectC++.

Auch sehr gut zu erkennen ist, wie sich durch den Einsatz von AspectC++ die Anzahl der bereits vorhandenen und durch die Features geänderten Quelltextzeilen reduziert. Dies wirkt sich in etwa demselben Verhältnis auch auf die Anzahl der geänderten Funktionen und Strukturen aus. An dieser Stelle zeigen sich allerdings auch die Beschränkungen von AspectC++ für die Modularisierung des vorliegenden CSOM-Quelltexts. Da einige der verwendeten Sprachmittel, wie bspw. Präprozessoranweisungen oder Struktur- bzw. Typendefinitionen nicht mit AspectC++ modularisiert werden können, ist es weiterhin nötig im Quelltext anderer Module Änderungen vorzunehmen, die nur für die Implementierung des speziellen Features notwendig sind. Ein Beispiel sind hier die Felder für die Garbage-Collection-Implementierungen oder die Methoden, die durch die Implementierung der Virtual Images zu einigen der mit C nachgeahmten Klassen hinzugefügt wurden.

Mit der Verwendung von VMADL und ClassDL lassen sich tatsächlich alle Features vollständig modularisieren. Somit lässt sich VMADL als Grundlage für die Implementierung einer VM-Produktfamilie verwenden. Bei der Implementierung einzelner Service-Module kann es zwar vorkommen, dass dies Auswirkungen auf bestehende Service-Module hat, da diese z. B. ihre Schnittstelle erweitern und zusätzlich relevante Ereignisse bereitstellen müssen, doch da diese auch von anderen Service-Modulen verwendet werden können, lässt sich wie in Abschnitt 4.3.3.9 argumentieren, dass diese Änderungen nicht dem Feature selbst zugerechnet werden müssen, sondern als Beitrag zur Überarbeitung und Qualitätssteigerung der anderen Service-Module be-

trachtet werden können. Zusätzlich ist damit implizit sichergestellt, dass die Architektur des Systems nicht verändert wird und die bestehenden Modulbeziehungen im System nicht verändert werden.

5.2.2 Analyse der Ausführungsgeschwindigkeit

5.2.2.1 Benchmark-Durchführung und Messfehlerbewertung

Für die Implementierung von virtuellen Maschinen werden auch heute noch, obwohl Ansätze wie metazirkuläre VMs [AAC+99, Mat08, USA05] und PyPy [RP06] praktikabel werden, sehr systemnahen Sprachen wie C oder C++ verwendet, da die Verfechter dieser Techniken die Möglichkeiten zur Performanceoptimierung in diesen Sprachen schätzen. Daher wird an dieser Stelle versucht, die Auswirkungen der aspektorientierten Programmierung bzw. VMADL auf die Performance zu ermitteln.

Für die Benchmarks wurden die jeweils aktuellsten Versionen verwendet. Für die originale CSOM bedeutet dies, dass die Revision 705 vom `trunk`-Entwicklungszweig verwendet wurde. Für alle durchgeführten Benchmarks gilt, dass die mit CSOM mitgelieferten Benchmarks, inklusive des `IntegerLoop`-Benchmarks, der von der Implementierung der Eins-markierten Integer stammt, verwendet wurden. Alle Benchmarks wurden einmal mittels der Klasse `All.som` ausgeführt und anschließend jeweils zehnmal in einer eigenen CSOM-Instanz mittels eines Shell-Skripts[6], um anschließend den Lauf mit der minimalen Laufzeit bestimmen zu können. Die minimale Laufzeit wird verwendet, um alle Auswirkungen von parallel auf demselben Prozessor laufenden Prozessen ausschließen zu können. Der ermittelte Wert stellt damit die beste erzielbare Laufzeit mit einer möglichst geringen Anzahl von Störungen dar.

Die Benchmarks wurden auf einem Rechner mit zwei Intel Xeon E5420 CPUs mit 2,50GHz, also insgesamt vier Kernen, 6MB Cache und 8GB RAM auf einem Debian 4.0 r3 mit einem 64-Bit-Kernel Version 2.6.18, aber 32-Bit-Userland durchgeführt. Die verwendeten Compiler sind aus der *GNU Compiler Collection* Version 4.1.2. Der verwendete AspectC++-Compiler wurde am 11. Juni 2008 direkt aus den Quellen des Projekt-SVNs[7] erstellt und trägt die Versionen `ac++ 1.0pre4` bzw. `ag++ 0.7`. Soweit nicht explizit angemerkt, werden zum Kompilieren von CSOM die Standardeinstellungen und die Optimierung `-O3` verwendet.

Um den Bereich der Messungenauigkeit abschätzen zu können, wurden Benchmark-Durchläufe mit einer ausgewählten in VMADL implementierten CSOM durchgeführt. Ziel war es dabei den Bereich zu ermitteln, in dem die Messergebnisse bewegen, ohne dass einer der betrachteten Parameter geändert wird. Neben den Grundbestandteilen für eine lauffähige VM waren das *Mark/Sweep-GC*-Service-Modul und das Service-Modul der Eins-markierten Integern Teil dieser Konfiguration. Es wurden die gleichen Skripte wie auch bei den anderen Benchmarks verwendet, die aber dieses Mal für diese Konfiguration zehnmal ausgeführt wurden. Dadurch ergeben sich letztendlich hundert Einzeldurchläufe für jeden einzelnen Benchmark. Anschließend wurde für jeden dieser zehn Durchläufe die Summe aus den minimal ermittelten Einzelwerten der verschiedenen Benchmarks gebildet und mit den anderen Durchläufen in Relation gesetzt. Der kleinste ermittelte Wert war eine Gesamtdurchlaufzeit für alle verschiedenen Benchmarks von 5130 ms, der größte Wert betrug 5155 ms. Damit ist die maximale Abweichung bei ca. 0,5% für die Summe über die Benchmarks. Die Standardabweichung über die ermittelten Werte lag bei 9 ms. Damit

[6] Auf der CD `/source/CSOM/branches/smarr/bench.php`
[7] AspectC++-Repository: `https://svn.aspectc.org/repos/AspectC++-Project/trunk`

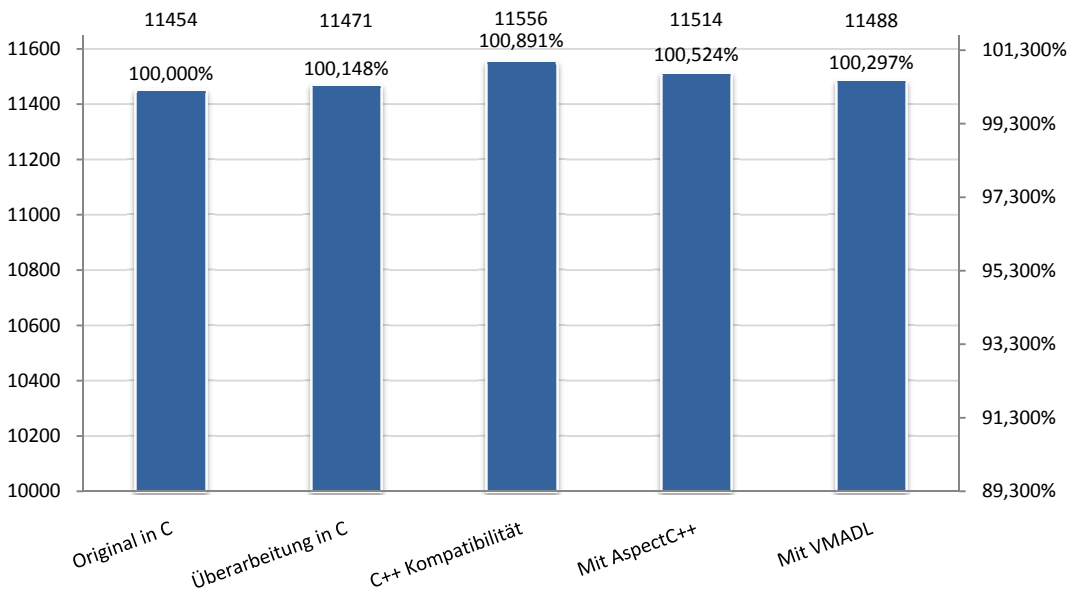


Abbildung 14 Auswirkungen der Überarbeitung auf die Performance

haben die ermittelten Werte in den durchgeführten Benchmarks eine Genauigkeit von ± 0,2%. Daraus lässt sich für die Ergebnisse sagen, dass ein Laufzeitunterschied von 25 ms durchaus im Rahmen von Messfehlern liegen kann und bei der Bewertung entsprechend berücksichtigt werden muss. Ursachen für die Messfehler sind einerseits die Multiprozessumgebung und die Möglichkeit, dass ein Prozess von einem anderen unterbrochen werden kann, aber andererseits auch die begrenzte Genauigkeit bei der Zeitermittlung durch das System.

5.2.2.2 Auswirkungen der Überarbeitung

Vor der Untersuchung der Benchmark-Ergebnisse für die einzelnen Implementierungen wird an dieser Stelle ein Blick auf die Auswirkungen der Überarbeitung, die im Laufe der Entwicklung von der originalen CSOM zur VMADL-CSOM durchgeführt wurden, geworfen. Die vorgenommenen Änderungen wurden bereits in Abschnitt 4.4 genauer betrachtet.

Abbildung 14 zeigt die Auswirkungen auf die Laufzeit aller Benchmarks zusammengenommen. Die originale CSOM ist hier die Referenz und hat damit 100%. Die Überarbeitungen für den Entwicklungszweig `smarr-trunk` verursachen laut der Benchmarks eine minimal höhere Laufzeit von 100,148%. Dies liegt jedoch innerhalb der Standardabweichung von 0,2% und kann damit auf Messfehler zurückgeführt werden. Mit dem Wechsel des Compilers `gcc` zu `g++` im Entwicklungszweig `features-gc` gibt es einen minimalen Anstieg der Laufzeit, der über der Messtoleranz liegt und auf den Compiler zurückzuführen ist. Die Messungen bei der Verwendung von AspectC++ im Zweig `aop-gc` und VMADL im Zweig `vmadl-classdl` zeigen einen geringeren Leistungsabfall. Hier ist zu vermuten, dass abgesehen von den Messungenauigkeiten vernachlässigbar kleine Auswirkungen durch die Überarbeitung zu sehen sind, da z. B. bei der Objektinitialisierung keine Variablenargumentlisten mehr verwendet werden. Insgesamt können die Auswirkungen der Überarbeitung jedoch als sehr klein betrachtet werden, sodass die Benchmark-Ergebnisse durch sie nicht verfälscht werden sollten.

Aus dieser Abbildung geht jedoch nicht die Problematik beim Einsatz von aspektorientierter Programmierung im Allgemeinen hervor, da in der eingesetzten

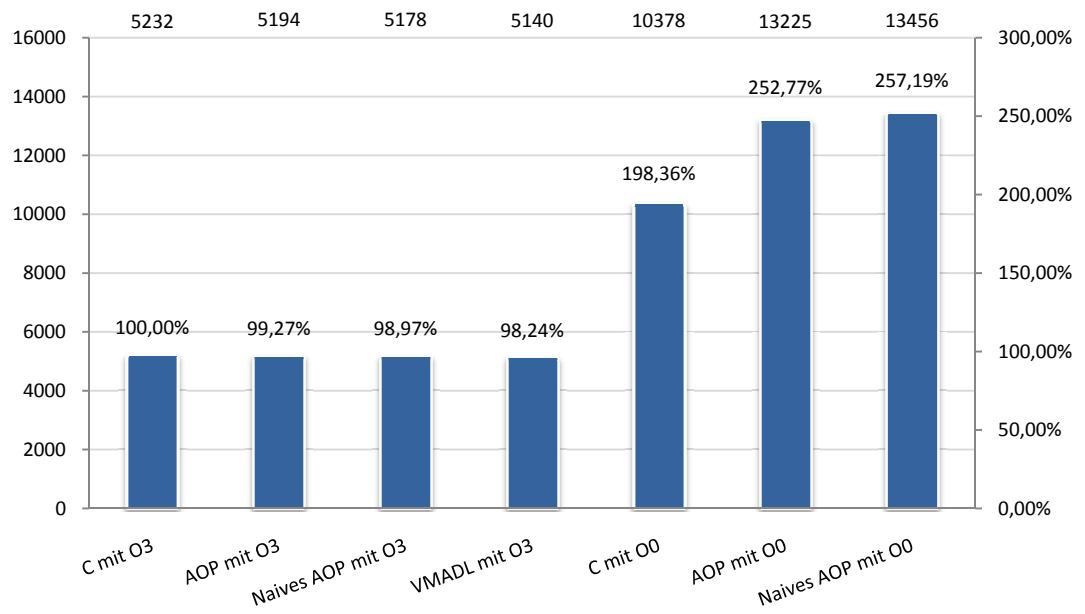


Abbildung 15 AspectC++-Performanceeinflüsse und Compiler-Optimierungen

Konfiguration der CSOM-VM nur sehr einfache `around`-Advices verwendet werden, die der Compiler wegoptimiert. Daher wird im nächsten Abschnitt eine andere Konfiguration betrachtet, die eine Bewertung der Auswirkungen der Verwendung von AspectC++ erlaubt.

5.2.2.3 Auswirkungen von AspectC++ auf die Performance

Die Performanceeinflüsse der Verwendung von AOP sind direkt davon abhängig, welche Sprachmechanismen verwendet werden. Sind die verwendeten Mechanismen statisch auflösbar, können sie vom Compiler eliminiert werden. Sobald sie jedoch abhängig vom Laufzeitverhalten sind, wie bspw. Cflow-Konstruktionen oder dynamische Abfragen von Parametern in einem Advice, kann es zu entsprechenden Performanceeinbußen kommen. Daher sollte bei performancekritischen Anwendungen genau darauf geachtet werden, welche Sprachmechanismen verwendet werden.

Die Auswirkungen von unterschiedlicher AOP-Verwendung, aber vor allem der Optimierung durch den Compiler sind in Abbildung 15 dargestellt. Untersucht wurden dafür die unterschiedlichen Implementierungen von CSOM mit Eins-markierten Integern, die jeweils auch die Mark/Sweep-GC verwenden. Bei der naiven AOP-Implementierung handelte es sich um den ersten Versuch der Modularisierung mittels AspectC++. Im Unterschied zu der letztendlich verwendeten AOP-Implementierung ist hier nur auf die Verwendung des Aspekt-Objekts zum Zugriff auf die Parameter eines Advices verzichtet und durch die Verwendung der AspectC++-Notation für Argumente ersetzt worden. In diesem Test wurden jedoch keine Cflow-Konstruktionen untersucht, da diese konzeptbedingt immer einen Performanceeinfluss haben. Als Referenz diente für diese Untersuchungen die reine C-Implementierung aus dem `features-tagged-int-one`-Entwicklungszweig. Die Unterschiede in den ersten vier betrachteten CSOM-Implementierungen sind auf unterschiedliche Compiler-Optimierungen zurückzuführen und gehen nur geringfügig über den Bereich der Messfehler hinaus. Wie sich hier herausstellt, kann sogar die Verwendung des Aspekt-Objekts zur Abfrage der Argumente als nicht performancebeeinflussend angenommen werden, wenn wie in diesem Beispiel stets mit konstanten Indizes auf die Argumente zugegriffen wird. Dies

liegt vermutlich an der Möglichkeit die Compiler-Optimierungen voll auszuschöpfen, da nur statische Argumentenindizes verwendet werden. Inwieweit diese Optimierung auch bei Verwendung von variablen Argumentenzugriffen möglich ist, wird hier nicht betrachtet.

Interessanter sind hingegen die Auswirkungen der Compiler-Optimierung insgesamt. Die Referenz-CSOM-VM ist in diesem Beispiel ohne Optimierung nur noch halb so schnell, behauptet sich aber sehr stark gegenüber den unoptimierten AOP-Versionen. Diese benötigen etwa 2,52 bzw. 2,57 mal so lange für die Ausführung der Benchmarks. Hier ist ebenfalls der Einfluss der unterschiedlichen Argumentabfrage zu sehen. So bleibt festzustellen, dass AOP einerseits in performancekritischen Bereichen abhängig von einer guten Compiler-Optimierung ist und andererseits die Sprachmittel voll ausgeschöpft werden sollten, wenn keine Compiler-Optimierung vorhanden ist.

Der deutliche Unterschied zwischen der unoptimierten CSOM-VM, die in C geschrieben wurde, und der unoptimierten AOP-CSOM-VM lässt sich auf die Implementierung von AspectC++ zurückführen. Hier kommen verschiedene Techniken zum Einsatz, die eine Anwendung von AOP auf eine statisch getypte Sprache wie C++ ermöglichen. Der Prozess des Aspektwebens wird auf Sprachebene durchgeführt und es wird vom Aspektweber C++-Quelltext generiert [SGSP02], der anschließend von einem Standard-C++-Compiler verarbeitet werden kann. Advices auf Funktionen führen so zu einer deutlichen Erhöhung der Funktionsaufrufschachtelung im Programm, da verschiedene Proxy-Mechanismen eingesetzt werden, um diese Funktionalität zu ermöglichen. Zusätzlich wird über Template-Konstruktionen die Parameter-Übergabe an die Advices und auch wieder an die originalen Funktionen realisiert. Im Normalfall lassen sich, wie die Benchmark-Ergebnisse belegen, diese zusätzlichen Funktionsaufrufe und Speicherstrukturen komplett wegoptimieren. Ohne Optimierung bleibt jedoch der erhebliche Performanceeinbruch.

Wie bereits angemerkt, bedeutet die Verwendung von Cflow-Konstruktionen konzeptbedingt immer einen Performanceeinfluss, da zur Laufzeit ermittelt werden muss, ob eine bestimmte Methode gerade ausgeführt wird. Für die Implementierung von CSOM wurde daher im Nachhinein bei allen Verwendungen von Cflow-Konstrukten überprüft, ob diese notwendig sind, oder ob sie entfernt werden können. Letztendlich war es möglich alle zu entfernen. Hier bleiben jedoch immer verschiedene Aspekte abzuwägen. Cflow-Konstruktionen können vor allem sicherstellen, dass Advices nicht auf Stellen angewendet werden, die nicht beabsichtigt werden. Um dies sicherzustellen, ist eine gute Unterstützung in der IDE sehr hilfreich, die die Stellen visualisiert, an denen Advices wirken. Wird kein `cflow` verwendet, um die Advices abzusichern, steigt der Wartungsaufwand entsprechend, da mit jeder Änderung überprüft werden muss, ob die Advices auch nur die richtigen Stellen beeinflussen.

Für die CSOM-Implementierung mit VMADL war es, wie bereits erwähnt, möglich alle Cflow-Konstrukte zu entfernen. Dies gelang aber auch nur, da einige Funktionen so in mehrere Funktionen aufgeteilt werden konnten, dass eindeutige Ziele für die Advices entstanden sind. Dies hat jedoch auch immer einen Einfluss auf die Struktur des Programms, und man muss daher genau abwägen, ob die neue Struktur semantisch immer noch für das Programm geeignet ist, oder ob andernfalls der Performancegewinn groß genug ist, um eine eventuell schlechtere Wartbarkeit in Kauf nehmen zu dürfen.

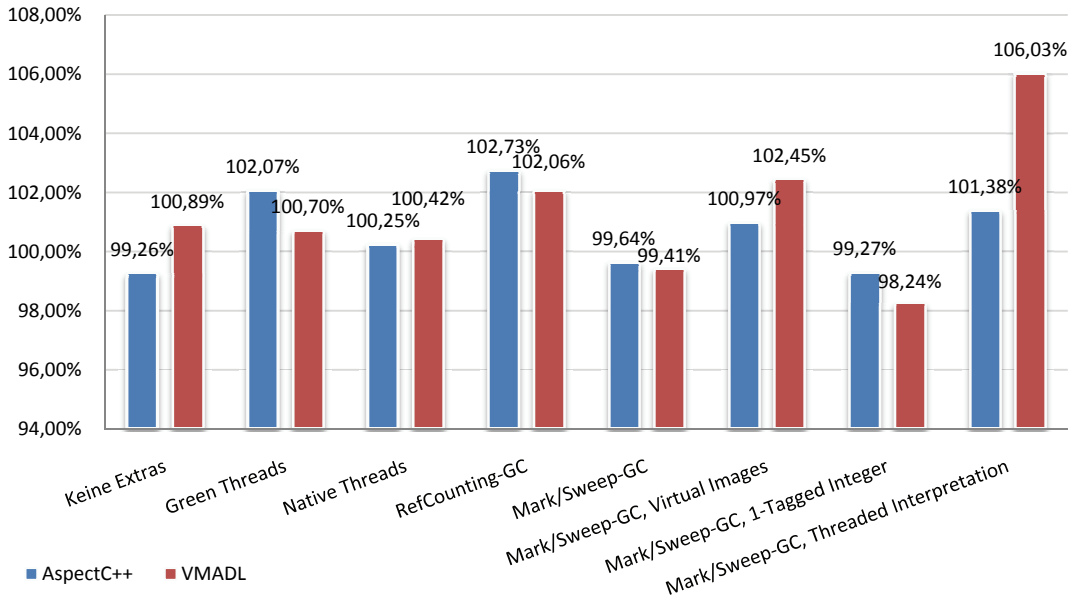


Abbildung 16 Laufzeitvergleich mit reinen C-Implementierungen

5.2.2.4 Performancevergleich der verschiedenen Implementierungen

Für diese Untersuchung wurden jeweils die reinen C-Implementierungen aus den `features`-Entwicklungszweigen mit den AspectC++-Implementierungen aus den `aop`-Entwicklungszweigen und der VMADL-Implementierung bei Verwendung der entsprechenden Konfiguration aus dem `vmadl-classdl`-Entwicklungszweig verglichen.

Die Ergebnisse zeigen nur sehr geringe Performanceunterschiede zwischen den Implementierungsvarianten. Lässt man den Wert für die VMADL-Variante von CSOM mit der Mark/Sweep-GC und Threaded-Interpretation einmal außen vor, liegt der Mittelwert der Abweichung bei 0,6%, wobei der Mittelwert der absoluten Abweichung bei 1,2% liegt. Zieht man davon noch die Messtoleranz von 0,2 Prozentpunkten ab, liegt die mittlere absolute Abweichung für die verschiedenen Implementierungsvarianten bei 1% und ist damit vernachlässigbar klein.

Ein Großteil der Schwankungen in den Ergebnissen ist auf Messfehler zurückführbar. Der Rest lässt sich mit den Überarbeitungen zwischen den einzelnen Implementierungsvarianten begründen. Die Implementierungen unterscheiden sich jeweils nur in der Verwendung von AOP und VMADL. Für die AOP- und die VMADL-Version werden auch jeweils nahezu identische Quelltexte zum Kompilieren verwendet, da der VMADL-Generator nahezu identische AspectC++-Implementierungsdateien generiert und durch die Verwendung des Konfigurationssystems jeweils nur die für die konkrete Konfiguration benötigten Advices generiert werden. Unterschiede können hier eventuell noch durch eine unterschiedliche Reihenfolge bei der Advice-Anwendung durch AspectC++ zustande kommen.

Ein Unterschied, der jedoch noch anzumerken bleibt, ist bedingt durch die Verwendung von ClassDL. Dies führt zu minimalen Abweichungen im Objektmodell, da die von ClassDL generierten Implementierungsdateien sich minimal von den handgeschriebenen unterscheiden. Eventuell kann es daher zu minimal anderen Compiler-Optimierungen kommen.

Die starke Abweichung der Threaded-Interpretation-Implementierung ist jedoch auf den ersten Blick nicht nachvollziehbar, da der Quelltext nachweislich nahezu

identisch ist. Ein Ansatzpunkt für die Erklärung wäre hier nur die Reihenfolge mit der AspectC++ die Advices anwendet, sodass in der VMADL-Version eventuell ein Programmfehler zum Tragen kommt, der eine Optimierung verhindert oder zusätzlichen Code ausführt, der in den anderen beiden Versionen nicht vorhanden ist. Dies ließ sich bisher jedoch nicht abschließend klären. Es kann jedoch festgestellt werden, das die 6% Performanceeinbuße auf einen Implementierungsfehler zurückzuführen sind, der sich theoretisch beheben lassen würde. Dazu müsste der vom AspectC++-Compiler generierte Quelltext der VMADL-CSOM-VM genau mit dem für die AOP-CSOM-VM generierte Version verglichen werden. Der Fehler liegt jedoch nicht in der Verwendung von AOP im Allgemeinen, da er in der AOP-Version nicht auftritt. Dort ist die implizit verwendete Reihenfolge der Advices vermutlich anders als in der VMADL-CSOM-VM und versteckt diesen Fehler. Die Änderung der Reihenfolge der Advice-Anwendung lässt sich durch die zwischen den Varianten geänderten Dateinamen erklären.

Abschließend lässt sind jedoch feststellen, dass die Performanceeinbußen insgesamt sehr gering bzw. nicht messbar sind und durch Optimierung, die durch ein besseres Verständnis des Zusammenspiels in CSOM möglich sind, mehr als ausgeglichen werden können.

5.2.3 EINSCHÄTZUNG

Die Untersuchung der Unterschiede zwischen den verschiedenen CSOM-VM Implementierungen hat gezeigt, dass VMADL zu einer minimalen Anzahl zusätzlicher Quelltextzeilen führt. Diese entsteht nachvollziehbar durch die zusätzlichen Strukturierungsmittel des Quelltexts, die vorher nur implizit vorhanden waren, mit VMADL aber explizit ausgedrückt werden können. Für CSOM ergab sich letztendlich ein Zuwachs des Quelltexts von etwa 2% durch die neuen Sprachmittel. Da diese zusätzlichen Zeilen aber direkt das Systemverständnis der Entwickler verbessern sollen und zusätzlich Verbesserungen in der Modularität sowie damit einhergehend Flexibilität im Sinne einer Produktfamilie bieten, sind sie als positiv anzusehen. Erst durch diese Sprachmittel wird es möglich, Features so in Service-Module zu kapseln, dass ihre Implementierung keine Änderungen an anderen Service-Modulen erfordert.

Bei der Untersuchung des Einflusses auf die Performance konnte festgestellt werden, dass eine performanceneutrale Verwendung der AOP-Sprachmittel möglich ist, ohne durch den Verzicht auf kritische Sprachkonstruktionen relevante Nachteile in Kauf nehmen zu müssen. Für CSOM hat sich gezeigt, dass die Verwendung von Cflow-Konstruktionen nicht notwendig ist, um die gewünschte Ausdrucksstärke der Pointcut-Definition zu erreichen. So konnte die Implementierung von CSOM mit VMADL dieselbe Leistung erreichen wie die reine C-Implementierung. Die einzelnen Werte lagen im Mittel etwa 1% über den Werten der C-Implementierung, sind jedoch auch durch nötige Überarbeitungen beeinflusst. Daher kann insgesamt gesagt werden, dass keine signifikante Änderung im Laufzeitverhalten verursacht wird.

Insgesamt sind die Ergebnisse damit sehr positiv und zeigen außer den ausdrücklich gewünschten zusätzlichen Strukturbeschreibungen keine Auswirkung auf das System. Im nächsten Abschnitt soll daher untersucht werden, ob diese zusätzlichen Sprachmittel die gewünschten positiven Auswirkungen auf die Entwicklung und den damit verbundenen Aufwand haben.

5.3 Auswirkungen auf Architekturwahrnehmung und Wartbarkeit

5.3.1 Ziel der Untersuchung

Nachdem die Untersuchung mithilfe von Metriken bereits einen Eindruck der Vorteile von VMADL geben konnte, soll ein Experiment die Möglichkeiten, die VMADL für die Softwareentwicklung bietet, besser bewertbar machen. Die Untersuchung wird dabei unter folgender Fragestellung durchgeführt:

Ermöglicht es die Verwendung von VMADL den Entwicklern, die Architektur einer VM genauer und schneller zu erfassen und dadurch effizienter Änderungen an ihr vornehmen zu können, ohne eine Verschlechterung der Architektur durch kontinuierlichen Verlust der ursprünglichen Architektureigenschaften in Kauf zu nehmen?

Um diese Fragestellung untersuchen zu können, werden Null-Hypothesen verwendet, deren Aussagen sich widerlegen lassen. Diese Vorgehensweise ist üblich, da sich für die hier betrachte Problematik keine Hypothesen bestimmen lassen, die exakt quantifizierbare Verbesserungen beschreiben und anschließend direkt überprüft werden könnten. Mit Null-Hypothesen kann hingegen die direkte Beziehung der zu untersuchenden Variablen verneint werden [Chr04]. Die in dieser Untersuchung zu widerlegenden Null-Hypothesen zur Fragestellung lauten wie folgt:

1. Die Verwendung von VMADL führt nicht zu einer konsistenten und gleichmäßigen Wahrnehmung der Architektur des Softwaresystems. Die Probanden kommen zu Ergebnissen, die stark voneinander und der erwarteten wahrzunehmenden Architektur abweichen.
2. Die Verwendung von VMADL hat keine positive Auswirkung auf die Dauer, die benötigt wird, die Architektur wahrzunehmen und zu dokumentieren.
3. Die Verwendung von VMADL hat keine positive Auswirkung auf die Zeitdauer, die benötigt wird, um eine Strategie zu entwickeln, mit der sich das Softwaresystem um eine neue Funktionalität erweitern lässt.
4. Die Verwendung von VMADL hat keine positive Auswirkung auf die Umsetzbarkeit und damit Korrektheit der entwickelten Strategie.

Im Rahmen dieser Arbeit bietet sich für die Untersuchung der Fragestellung ein kontrolliertes Experiment an, wie es bspw. von Easterbrook und Aranda [SEP06] und Christensen [Chr04] beschrieben wird. Im folgenden Abschnitt wird daher auf der Grundlage dieser Quellen ein solches Experiment mit Labor-Charakter entworfen und es werden die problematischen Einflüsse diskutiert, die zu einer Verfälschung der Ergebnisse führen können.

5.3.2 Entwurf eines Experiments mit anschließender Befragung

5.3.2.1 Probanden

Für die Durchführung des Experiments wurden Studenten herangezogen, die auf freiwilliger Basis teilnahmen und auch keine Aufwandsentschädigung erhielten. Die Probanden wurden dahin gehend ausgewählt, dass sie bereits Erfahrung mit der Arbeit an virtuellen Maschinen hatten. Dazu ist das Auswahlkriterium die erfolgreiche Teilnahme

an der Vorlesung *Virtuelle Maschinen* von Michael Haupt in den Sommersemestern 2007 oder 2008.

Damit haben die Probanden einen ausreichend gleichen Wissensstand und auch praktische Erfahrung in der Entwicklung von VMs, da die praktische Lösung von Implementierungsaufgaben Teil dieser Vorlesung war. Um eine zu starke Beeinflussung der Ergebnisse durch ein auf VMs fokussiertes Interesse zu vermeiden, wurden alle Studenten, die an der Vorlesung teilgenommen haben und noch verfügbar waren, gebeten als Probanden für das Experiment zur Verfügung zu stehen. Das Interesse der Einzelnen an diesem Thema variiert damit relativ stark. Für einige war diese Vorlesung das letzte Mal, dass sie mit dem Thema in Berührung gekommen sind. Andere haben sich jedoch darüber hinaus in anderen Projekten weiter in diesem Feld engagiert. Damit kann davon ausgegangen werden, dass die Probanden einerseits über genug Wissen in diesem Feld verfügen, andererseits aber keine reinen System- bzw. Plattformentwickler sind und die Ergebnisse zu einem ausreichend hohen Grad auf die Allgemeinheit von Softwareentwicklern übertragen werden können.

5.3.2.2 Arbeitssituation und Aufgabenstellung

Für das Experiment stand nur eine begrenzte Anzahl von geeigneten Studenten als Probanden zur Verfügung. Um mit diesen Probanden trotzdem zu möglichst aussagekräftigen Ergebnissen zu kommen, wurde eine Standard-Experimentsituation mit einer Experimentgruppe und einer Kontrollgruppe mit anschließender Ergebnisüberprüfung und Befragung gewählt.

Um trotz des Laborcharakters des Experiments eine möglichst typische Arbeitssituation für die Probanden zu schaffen, sollten sie in Teams, bestehend aus zwei Personen, arbeiten. Die Verteilung der Probanden auf die Gruppen und Teams erfolgte dabei einzeln durch eine zufällige Zuordnung im Vorfeld. Die Probanden wurden jedoch erst durch den Durchführenden in ihrer jeweiligen Gruppe über die Details des Experiments informiert. Damit wurde die maximale Anzahl an Freiheitsgraden für die Zuordnung erreicht. Im Allgemeinen kann daher davon ausgegangen werden, dass die Probanden gleichmäßig auf die Gruppen und Teams verteilt wurden, individuelle Unterschiede zwischen ihnen ausgeglichen sind und Gruppen mit annähernd gleicher Qualifikation erreicht wurden.

Da für die Durchführung des Experiments Seminarräume verwendet wurden, um eine möglichst genaue Kontrolle über alle potenziell störenden Faktoren zu haben, unterscheiden sich die Arbeitsweise und die Situation, in der die Probanden arbeiteten, etwas von ihrem sonst üblichen Arbeitsumfeld. Beide Gruppen haben zeitgleich angefangen, um einen direkten Austausch und somit eventuelle unkontrollierte Beeinflussung zu vermeiden. Da die Bewältigung der Aufgaben etwa zwei bis drei Stunden in Anspruch nehmen sollte, durften die Teams jeweils in Eigenregie kleine Pausen machen, damit trotz allem eine zu gewissen Teilen freie Arbeitsweise, wie sie sonst üblich ist, erreicht werden konnte.

Die Aufgabenstellung war für beide Gruppen nahezu identisch. Sie bekamen eine vorbereitete Version des CSOM-Quelltexts und sollten diesen analysieren. Nach dem Analyseschritt war die erste Aufgabe die Dokumentation der vorgefundenen Architektur. Die zweite Aufgabe war es eine Strategie zu entwickeln, wie Green Threading in diese CSOM-Version integriert werden könnte. Die dritte Aufgabe, das Ausfüllen eines Fragebogens, wurde von den Probanden einzeln und nicht mehr im Team ausgeführt. Der Fragebogen sollte einerseits das Vorwissen der Probanden als Selbsteinschätzung erfassen, aber auch die Erfahrungen mit den eingesetzten Technologien dokumen-

tieren, um daraus in der Auswertung Verbesserungen für das VMADL-Design ableiten zu können und Probleme mit dem Experimententwurf und der Durchführung zu erkennen.

Gruppe 1 hat als Kontrollgruppe eine CSOM-Version bearbeitet, die ohne VMADL implementiert wurde. Gruppe 2 bekam als Experimentgruppe eine mit VMADL realisierte CSOM. Der Unterschied zwischen den beiden Quelltext-Versionen besteht allein in der Verwendung von VMADL. Die implementierten Features sind hingegen identisch. Die Quelltexte entsprach dabei im Wesentlichen der Version von CSOM, die für die letzte Vorlesung verwendet wurde, und brachte daher die Mark/Sweep-Garbage-Collection mit.

Für beide Gruppen war somit die Aufgabenstellung nur durch die verwendete Technologie variiert. Die Architektur der Systeme war ebenfalls gleich. Im Fall der Experimentgruppe war sie jedoch durch die Verwendung von VMADL expliziter dargestellt und mit dem Experiment sollte geprüft werden, ob die Probanden die Architektur dadurch auch genauer wahrnehmen. Die Dokumentation der Architektur sollte von den Teams mit einer selbst gewählten Notation auf Papier erfolgen. Empfohlen wurde den Probanden die Verwendung von FMC oder UML. Dies wurde jedoch nicht zur Voraussetzung gemacht. Um einen Einfluss von unterschiedlichen Notationskenntnissen auszuschließen, konnte von den Probanden auch eine selbst gewählte Ad-hoc-Notation verwendet werden.

Die Aufgabenstellung für die Dokumentation der entwickelten Implementierungsstrategie für ein Green-Threading-Modul wurde so gestaltet, dass die Teams auf der Funktionsebene der Implementierung ansetzen sollten. Dies bedeutet, sie sollten alle Funktionen identifizieren, welche verändert werden mussten. Zusätzlich sollten sie exakt beschreiben, in welchen Modulen sie Funktionalität ändern oder hinzufügen würden. Dies sollte bei der anschließenden Bewertung der Architektur eine Identifikation der somit entwickelten Architektur ermöglichen und alle Verletzungen der ursprünglichen Architektur aufdecken. Zusätzlich sollte jeweils beschrieben werden, welche Technik eingesetzt wurde, um die gewünschte Modularität zu erreichen. Hier standen den beiden Gruppen unterschiedliche Möglichkeiten zur Verfügung. Die Experimentgruppe hatte mit VMADL deutlich mächtigere Möglichkeiten als die Kontrollgruppe, die sich z. B. auf Präprozessor-Anweisungen beschränken musste, um zur Kompilierungszeit die gewünschte Modularität zu erreichen. Da jedoch in beiden Gruppen in der Aufgabenstellung nicht ausdrücklich die Modularität als Ziel ausgegeben wurde, konnte davon ausgegangen werden, dass die Implementierung der Kontrollgruppe nicht sehr modular aufgebaut sein würde.

5.3.2.3 Variablen der Untersuchung

Dieses Experiment sollte ermitteln, welchen Einfluss die Verwendung von VMADL auf die verschiedenen Faktoren hat, die über die Null-Hypothesen beschrieben wurden. Daher wird die Arbeitsgrundlage für die beiden Gruppen genau in diesem Faktor variiert. VMADL bildet damit die sogenannte unabhängige Variable des Experiments.

Als sogenannte abhängige Variablen werden die Dauer und Korrektheit für das Erfassen und Dokumentieren der Architektur sowie das Entwickeln einer konkreten Implementierungsstrategie für ein zusätzliches Service-Modul angesehen, da sich auf sie die unabhängige Variable auswirkt. Über die benötigte Dauer für das Erfassen und Dokumentieren der Architektur lässt sich schließen, wie explizit diese im Quelltext zu finden ist und welcher Aufwand nötig ist, um die Beziehungen zwischen den Implementierungsmodulen zu interpretieren. Problematisch ist hingegen die exakte Er-

fassung und Trennung dieses Wertes von der Dauer, die für das Ermitteln einer Implementierungsstrategie notwendig ist, da typischerweise bei der Entwicklung dieser Strategie das Architekturbild noch verfeinert wird. Dies war bei der Durchführung entsprechend zu berücksichtigen.

Zusätzlich zur Dauer wurde die Korrektheit der Lösung bestimmt, um hier ebenfalls Abweichungen, die durch die unabhängige Variable erzeugt wurden, feststellen zu können. Die Korrektheit wurde über ein Punktesystem ermittelt, welches das Vorhandensein oder Fehlen wesentlicher Charakteristika in der Architekturbeschreibung und der Implementierungsstrategie bewertet. Die Bewertungskriterien sind im Abschnitt 5.3.3.1 dokumentiert.

Neben den abhängigen Variablen, die im Fokus der Betrachtung stehen und durch die Null-Hypothesen gegeben sind, wurden noch andere Eigenschaften der Lösungen untersucht und bewertet, damit nach dem Experiment eventuell verfälschende Einflüsse und auch unerwartete Auswirkungen der unabhängigen Variablen festgestellt werden konnten. Zu diesen Eigenschaften gehören die *formale Korrektheit der gewählten Notation* und der *Detailgrad der Lösungsstrategie*, der zwar durch die Aufgabenstellung relativ genau vorgegeben war, aber von den einzelnen Teams unterschiedlich interpretiert werden konnte.

5.3.2.4 Verfälschende Einflüsse auf die abhängigen Variablen

Auch bei einem kontrollierten Labor-Experiment wie diesem lassen sich viele verfälschende Einflüsse auf die Ergebnisse nicht genau steuern. Es gibt jedoch für die meisten Probleme Strategien die Auswirkungen abzumildern. So haben die unterschiedlichen Programmierkenntnisse von Probanden natürlich Auswirkung auf die Geschwindigkeit und Genauigkeit, mit der sie die Systemarchitektur erfassen können. Dies gilt auch für die unterschiedliche Motivation und Aufnahmefähigkeit von Probanden. Solche Faktoren können in so einem Fall nur durch zufällige Verteilung ausgeglichen werden, da sie im Vorfeld nicht geeignet erfasst werden können. Durch eine zufällige Verteilung von Probanden auf Gruppen bzw. Teams entsteht im Allgemeinen jedoch ein sehr angeglichenes Niveau.

Darüber hinaus war es das Ziel beim Entwurf dieses Experiments eine möglichst einfache Konstruktion zu wählen, die nicht durch übermäßige Komplexität zusätzliche schlecht handhabbare Faktoren einbringt, durch welche die Ergebnisse hätten verfälscht werden können.

Eine wichtige Voraussetzung für die geeignete und korrekte Dokumentation der wahrgenommenen Architektur ist eine Kenntnis von Notationen für solche Architekturen. Da diese Aufgabe nicht für alle Probanden alltäglich ist, sind sie unterschiedlich stark geübt. Unsicherheiten im Umgang mit der zu verwendenden Notation hätten daher Auswirkungen auf die zur Lösung der Aufgabe benötigte Zeit. Um diese Auswirkungen zu minimieren, wurde einerseits die Verwendung einer Ad-hoc-Notation erlaubt und andererseits für die am Hasso-Plattner-Institut gebräuchlichen Notationen UML [Obj07] und FMC [KGT06] ein Überblicksblatt bereitgestellt, auf dem die wesentlichen Notationselemente und ihre Verwendung dargestellt waren.

Ein anderer nicht zu unterschätzender Faktor für die Lesegeschwindigkeit bei Quelltext ist die Entwicklungsumgebung, mit der gearbeitet wird. Die Probanden wurden daher im Vorfeld nach ihren Präferenzen befragt und der CSOM-Quelltext in einer passenden Form bereitgestellt, sodass hier keine verfälschenden Auswirkungen entstehen sollten und die Probanden in ihrem gewohnten Umfeld arbeiten konnten.

Für die Beschreibung der Architektur eines Systems gibt es zwei mögliche Blickwinkel, die Auswirkungen auf die Darstellung des Systems haben könnten. Wenn ein System nur im Quelltext ohne weitere Dokumentation vorliegt, kann man einerseits die statische Struktur des Quelltextes als die Architektur des Systems begreifen, aber andererseits kann man dynamische Aspekte höher gewichten und so die Laufzeiteigenschaften eines Systems stärker in die Architekturdarstellung einfließen lassen. Um diesen Freiheitsgrad einzuschränken, wurden die Probanden in der Einführung gebeten, sich auf die statische Architektur, die für VMADL relevanter ist, als wesentliche Charakteristik zu konzentrieren.

Das Vorhandensein der Möglichkeiten der aspektorientierten Programmierung, die mit VMADL für eine bessere Modularisierung genutzt werden sollten, könnte ebenfalls einen über den beabsichtigten Effekt hinausgehenden Einfluss haben. Allein das Wissen zur Verfügbarkeit dieser Sprachmittel könnte die Entwicklung der Implementierungsstrategien beeinflusst haben. Dabei spielen aber auch unterschiedliche AOP-Kenntnisse ein Rollen und es besteht die Möglichkeit, dass allein durch die Beschreibung der Möglichkeiten in der Vorbereitung des Experiments nur in der Experimentgruppe unterschiedliche Strategien angeregt worden sind, wodurch die Kontrollgruppe benachteiligt gewesen wäre. Um diesen Faktor so gering wie möglich zu halten, wurde in der Einführung in beiden Gruppen ein relativ ähnliches Bild von den Anforderungen an die Implementierungsstrategie vermittelt. Durch die Anforderung, dass die Strategie auf Funktionsebene beschrieben und zusätzlich Details zu Funktionen, die verändert wurden, angegeben werden sollten, ist anzunehmen, dass dies ein ähnliches Problembewusstsein in beiden Gruppen erzeugte.

Subtiler sind jedoch andere Einflüsse auf die Probanden. So ist laut Christensen [Chr04] immer auch die Beziehung der Probanden zu den Durchführenden des Experiments zu beachten. Ein Faktor ist hierbei, dass Probanden dazu neigen sich selbst in einem möglichst guten Licht zu präsentieren. Dies hätte bei einer unterschiedlichen hierarchischen Stellung der Durchführenden in den beiden Gruppen verschiedene Auswirkungen haben können, da davon ausgegangen werden kann, dass sich Probanden z. B. gegenüber einem Professor anders als gegenüber einem Mitstudenten verhalten. Daher wurde das Experiment von zwei Masterstudenten durchgeführt. Somit sind sie auf dem gleichen Stand wie die Probanden, die ebenfalls Masterstudenten waren.

Ähnliche Auswirkungen kann das Engagement desjenigen, der die Experimentergebnisse bewertet, auf die Daten haben. Sie könnten subjektiv beeinflusst werden. Wenn bei der Bewertung bekannt wäre, ob eine Lösung aus der Experiment- oder Kontrollgruppe kommt, könnten Unterschiede in der Bewertung gemacht werden. Daher mussten die Lösungen vor einer Bewertung so bearbeitet werden, dass der Korrektor nicht feststellen konnte, aus welcher Gruppe sie stammen, sodass er sie *blind* bewerten konnte.

5.3.2.5 Bewertung der Verallgemeinerbarkeit des Experiments

Zur Bewertung, ob dieses Experiment geeignet ist, Daten zu liefern, die verallgemeinert werden können, gibt es drei wichtige Kriterien [Chr04].

Das erste Kriterium ist die inhaltliche Übereinstimmung, also die Bewertung, zu welchem Grad die Aufgaben tatsächlichen Aufgaben im normalen Entwicklungsalltag entsprechen. Da der Lebenszyklus von Software im Wesentlichen durch den Anteil der Wartung bestimmt wird, erscheint das gewählte Szenario als sinnvoll. In der Literatur wird von einem Kostenanteil für Wartung und Betrieb von 70% bis 90% am IT-Budget

ausgegangen [Ede93, Erl00]. Daher kann man davon ausgehen, dass sich in vielen Fällen neue Entwickler in ein bestehendes System einarbeiten und Änderungen daran vornehmen müssen. Ein großes Problem ist dabei die Wahrnehmung und Erhaltung der Architektur des Systems, da diese meist nicht explizit dokumentiert wurde. Daher kommt die gewählte Aufgabenstellung dem typischen Problem schon sehr nahe, auch wenn sie über die normalerweise erledigten Aufgaben hinaus geht, da nicht alle Entwickler die Zeit investieren die Architektur vor eine Änderung noch einmal explizit darzustellen.

Das zweite Kriterium ist die Beziehung der gewählten abhängigen Variablen zum übergeordneten Kriterium und die damit verbundene Möglichkeit, Vorhersagen für dieses Kriterium abgegeben zu können. Für dieses Experiment ist daher die implizierte Beziehung zwischen der Geschwindigkeit und Korrektheit, mit der die Architektur erfasst und dokumentiert werden kann, und der erwünschten besseren Wahrnehmbarkeit der Architektur zu betrachten. Die Geschwindigkeit, mit der Entwickler bestimmte Aspekte in einem Programm erfassen können, hängt davon ab, wie explizit dieser Aspekt im Quelltext dargestellt wurde. Dieser Zusammenhang ist allgemein anerkannt und drückt sich unter anderem in der Verwendung von domänenspezifischen Sprachen aus, die bestimmte Sachverhalte sehr explizit und mit einem geeigneten Vokabular beschreibbar machen. Neben der Geschwindigkeit beeinflusst dies gleichfalls die Korrektheit der Wahrnehmung, da bei einer erhöhten Wahrnehmungsgeschwindigkeit mehr Zeit auf die korrekte Wahrnehmung verwendet werden kann.

Das letzte und allgemeinste Kriterium ist der Grad, mit dem vom konkreten Experiment auf den allgemeinen Sachverhalt geschlossen werden kann. Für dieses Experiment gibt es einige Eigenschaften, die dagegen sprechen könnten. Einerseits hat das Experiment Labor-Charakter und entspricht damit nicht dem normalen Arbeitsumfeld, andererseits haben die Studenten noch keine jahrelange Erfahrung, wie sie Entwickler im industriellen Umfeld haben können. Zusätzlich ist die untersuchte Software eine speziell auf die Lehre ausgelegte virtuelle Maschine, nicht so komplex wie z. B. die HotSpot-VM von Sun. Diese Eigenschaften haben jedoch keinen so starken Einfluss auf die Verallgemeinerbarkeit der Ergebnisse, dass das Experiment nicht mehr geeignet ist. So beeinflusst der Laborcharakter des Experiments nur minimal die Verallgemeinerbarkeit. Probanden arbeiten in solch einer Situation vermutlich insgesamt etwas konzentrierter, als sie es mit den typischen Unterbrechungen im Arbeitsalltag könnten, aber die grundsätzliche Arbeitsweise wird nicht beeinflusst. Zusätzlich hatten sie die Möglichkeit in den Zweierteams zu kommunizieren, wie sie es aus anderen Projekten gewohnt sind. Damit können die Ergebnisse nicht auf das Prozent genau die Verbesserung durch das eingesetzte Verfahren belegen, aber dies ist bereits durch die begrenzte Anzahl an Probanden ausgeschlossen. Die Ergebnisse sollten jedoch die starke Tendenz belegen können, dass es eine generelle Verbesserung durch den Einsatz einer expliziteren Architekturdarstellung gibt. Dies gilt auch in Bezug auf die Programmiererfahrung, hier jedoch noch etwas abgeschwächter, da die Probanden regelmäßige Programmiererfahrung haben. Im Durchschnitt haben sie bereits 9 Jahre Erfahrung und 3 Jahre als Werkstudenten in der Softwareentwicklung gearbeitet. Die Übertragbarkeit auf ein komplexeres Softwaresystem ist ebenfalls gegeben. Hier bedarf es voraussichtlich noch einiger strukturierender Ergänzungen zum Sprachumfang von VMADL, z. B. in Form eines Submodulkonzepts, jedoch stellt dies nur eine Verfeinerung dar und beeinflusst nicht die allgemeine Idee und die Wahrnehmung durch die Entwickler.

5.3.2.6 Vorbereitung und Durchführung

Aus dem Entwurf des Experiments und den zu beachtenden Einflüssen ergaben sich verschiedene Punkte, die beachtet werden mussten, um verlässliche Daten vom Experiment ableiten zu können.

Ein Ergebnis der Vorbereitung waren die an die Probanden auszuhändigenden Materialien. Diese sind in Anhang 3 beigefügt. Die verwendeten Präsentationsfolien enthalten eine jeweils leicht angepasste Version derselben Einführung. [8] Bei der Kontrollgruppe wurde jedoch statt einer Vorstellung von VMADL, eine kurze Wiederholung zur Struktur des CSOM-Quellpakets gegeben, um hier in beiden Gruppen das vermittelte Wissen auf dem selben Niveau zu halten. Die Durchführenden wurden vor dem Experiment so instruiert, dass die Einführungen bei beiden Gruppen den gleichen zeitlichen und inhaltlichen Rahmen haben sollten.

Beim Eintreffen der Probanden zum Experiment wurde ihnen jeweils mitgeteilt, in welchem Seminarraum sie abhängig ihrer Gruppenzugehörigkeit arbeiten sollten, ohne an dieser Stelle jedoch schon auf Details zum Experiment hinzuweisen. Sobald alle Probanden einer Gruppe anwesend waren, sollte mit der Einführung begonnen werden. Als erstes bekamen sie die den ersten Teil des Arbeitsmaterials ausgehändigt. Dieser besteht aus der Aufgabenstellung und den Blättern, auf denen sie ihre Ergebnisse notieren sollten. Der Fragebogen wurde jedoch erst nach der Durchführung ausgeteilt. Zusätzlich bekam jedes Team einen Stift in einer bestimmten Farbe, mit dem es die Lösungen zu den Aufgaben dokumentieren sollte. Die Farbe des Stifts ist vom Durchführenden auf dem Protokoll vermerkt worden, damit später die unterschiedlichen Farben für die einzelnen Durchführungsphasen zugeordnet werden konnten.

Nachdem die Materialien ausgeteilt wurden, begannen die Durchführenden mit der Einführung. Nach dem diese beendet war, sollten die Probanden mit ihrer Arbeit beginnen. Dazu wurde ihnen der Quelltext der zu untersuchenden CSOM-Version ausgehändigt, mit dem sie auf ihrem eigenen Notebook arbeiten konnten. Um hier den Aufwand für die Probanden so gering wie möglich zu halten, wurden entsprechende Projektdateien für ihre Entwicklungsumgebung vorbereitet, die ebenfalls im Quelltext-Archiv enthalten waren. Die Phase zum Vorbereiten und Einrichten der Arbeitsumgebung war damit minimal und fällt bei der Bewertung der Arbeitsdauer nicht ins Gewicht.

Sobald die Probanden mit der ersten Aufgabe fertig waren, sollten sie sich beim Durchführenden melden, um von ihm einen Stift mit einer neuen Farbe zu bekommen. Damit war es einerseits möglich, die Dauer der einzelnen Arbeitsschritte zu erfassen und andererseits ließ sich nachvollziehen, welche Änderungen an der aufgezeichneten Architektur im zweiten Arbeitsschritt vorgenommen wurden.

Mit Abschluss der zweiten Aufgabe sollten sich die Probanden noch einmal beim Durchführenden melden, um einen neuen Stift einer weiteren Farbe zu erhalten. Hier musste ebenfalls wieder die Zeit notiert werden. Außerdem verteilte der Durchführende den Fragebögen, der von den Probanden ausgefüllt werden sollte. Dem Fragebogen war auch die Referenzarchitektur beigelegt, damit die Probanden bei der Bearbeitung des Fragebogens ihre Architektur mit dieser vergleichen konnten.

[8] Die Folien sind auf der CD unter `/Arbeit/Evaluierung/Experiment/` abgelegt.

5.3.3 Auswertung des Experiments

5.3.3.1 Bewertungskriterien

Für die Bewertung der von den Teams angefertigten Architekturbilder und Implementierungsstrategien wurde die vorhandene Implementierung zugrunde gelegt. Als Referenzarchitektur wurde die im Anhang 3 zu findende Abbildung verwendet. Zur Bewertung der Übereinstimmung mit dieser Architektur wurden die Darstellungen auf die Vollständigkeit der dargestellten Service-Module und deren Interaktionen untersucht. Von den in der Abbildung dargestellten Elementen konnte abgewichen werden, wenn trotzdem eine Wiedererkennung möglich war. So konnte das *VM*-Service-Modul sowohl durch seine Einzelteile als auch in seiner Gesamtheit dargestellt werden. Die Service-Module *Bytecode* und *ObjectModel* mussten nicht erkannt werden. Die in der Referenzarchitektur dargestellten Interaktionen sollten hingegen vollständig erkannt und dokumentiert werden. Für jedes der dargestellten Elemente gab es einen Punkt, sodass bei einer exakten Darstellung sieben Service-Module und zehn Interaktionen erkannt werden mussten, um eine volle Punktzahl zu erreichen.

Als weiteres Kriterium wurde die Vollständigkeit in Bezug auf die für Green Threading relevanten Module betrachtet. Die korrekte Verwendung der Notation und der optische Eindruck in Bezug auf Sorgfalt und Gewissenhaftigkeit bei der Dokumentation wurden ebenfalls betrachtet, wobei letzteres eine rein subjektive Einschätzung ist.

Bei der Auswertung der Strategien wurden insgesamt 16 Punkte definiert, die für eine Implementierung in der Art und in dem Umfang beschrieben werden sollten, wie sie die Referenzimplementierung vorgibt. Für das *VM*-Service-Modul sollte erkennbar gemacht werden, dass eine Initialisierung neu hinzugefügter Klassen und eine Anpassung der Verarbeitung in der Shell nötig sind. Für hinzugefügte Klassen sollte erkennbar sein, dass sie von `VMObject` erben oder allgemeiner eine Abhängigkeit zum *VMObjects*-Service-Modul begründen. Bei der Betrachtung des Interpreters sollte erkannt werden, dass für ein präemptives Scheduling und für das Setzen des aktuellen Frames bzw. die Abfrage des aktuellen Frames eine Änderung notwendig ist. Für eine sinnvolle Verwendung von Green Threading muss in der Klassenbibliothek eine Möglichkeit vorgesehen werden, Blöcke zur Ausführung zu bringen oder Thread-Klassen implementieren zu können. Ebenfalls sollte ein Synchronisationsmechanismus mit Mutexen oder Semaphoren angeboten werden. Hierfür gab es jeweils einen Punkt, wenn die Ansätze für das entsprechende Konzept erkennbar waren. Für eine Implementierung hätte in dem Zusammenhang auch ein Hinweis auf die Realisierung von Primitiven und deren Registrierung gegeben werden sollen. Da alle erwarteten Implementierungen in irgendeiner Form zumindest einzelne globale Zeiger auf einer Verwaltungsstruktur für das Scheduling mitbringen, sollte eine entsprechende Behandlung durch die GC ebenfalls erwähnt werden. Für die Funktionalität des Schedulers wurde eine Erwähnung der Initialisierung, eines Mechanismus zum Erstellen neuer Threads, die Handhabung kritischer Sektionen und eines Scheduling-Algorithmus erwartet.

In die Bewertung sind außerdem die Formulierungen eingeflossen, die genutzt wurden, um anzudeuten, wie eine Modularisierung erreicht werden sollte. Der Umfang der Strategie wurde aufgenommen und es wurde eine subjektive Einschätzung der Qualität in Bezug auf eine Umsetzbarkeit und in dieser Hinsicht Vollständigkeit vorgenommen.

5.3.3.2 Auswertung der Architekturwahrnehmung

Die einfachste Strategie zur Ermittlung der Module der Architektur wäre in beiden Gruppen eine direkte Verwendung der Ordnernamen in der Dateistruktur des Quelltext-Paketes gewesen, in der die Implementierungsmodule organisiert waren. Diese Strategie hat jedoch nur ein einziges Team verwendet. Dieses Team war aus der Kontrollgruppe und hatte somit die reine C-Implementierung vor sich. Es konnte neben den Modulen auch deren Interaktionen wie gewünscht wiedergeben.

Alle anderen Teams haben die Architektur nur unvollständig dargestellt. Es wurden dabei weder alle Module, noch alle Interaktionen dargestellt. Allerdings ließ sich bei der Analyse der Darstellungen erkennen, dass die dokumentierten Architekturen sehr auf die Lösung des gestellten Implementierungsproblems optimiert waren. So wurden die relevanten Aspekte wie die Bytecode-Erzeugung und die Interaktion des Systems mit dem Interpreter sehr ausführlich dargestellt. Die Architekturen lagen damit nicht für alle dargestellten Elemente auf derselben Abstraktionsebene, sondern haben relevante Elemente stärker verfeinert dargestellt. Es kann hier davon ausgegangen werden, dass die dargestellten Aspekte direkt mit dem Verständnis des Systems einhergehen. Einige der aufgenommenen Aspekte gingen sogar soweit, dass sie dynamische Zusammenhänge auf konzeptueller Ebene abstrahierten, die so nicht direkt aus dem Quelltext ersichtlich sind, sondern erst durch ein Verständnis der Zusammenhänge im System erkennbar werden. Dazu gehört bspw. die Interaktion mit den Primitiven.

Bei den VMADL-Teams musste zudem festgestellt werden, dass diese den VMADL-Definitionen nicht die erhoffte Relevanz zugeordnet haben. Aus den Darstellungen kann keine Beeinflussung durch diese Definitionen abgeleitet werden. Weder wurde die Vollständigkeit der Darstellung, noch das Abstraktionsniveau im Vergleich zu der Kontrollgruppe merklich beeinflusst. Hier muss davon ausgegangen werden, dass die Einführung von VMADL nicht das nötige Bewusstsein zu diesen Zusammenhängen geschaffen hat. Einerseits war sie dafür vermutlich zu kurz, andererseits wurden die Architekturen, wie bereits festgestellt, von den Gruppen dazu genutzt, das ihrer Meinung nach nötige Systemverständnis zu erlangen. Es kann davon ausgegangen werden, dass die Probanden einen Großteil der Zeit dafür aufgewandt haben, den Quelltext der Implementierungsmodule zu analysieren, da dies das für sie gewohnte Vorgehen darstellt, um eine Änderung implementieren zu können.

Die Daten zur benötigten Zeit für die Erfassung der Architektur sind in der Experimentgruppe leider nicht verlässlich, wie in Abschnitt 5.3.3.5 erläutert wird, liegt jedoch die Vermutung nahe, dass die Einarbeitungszeit für VMADL unterschätzt wurde und die Teams daher eine gewisse Unsicherheit mit dem Umgang mit dieser Sprache hatten, die sich durch einen höheren Zeitbedarf bemerkbar machte.

Besonders herauszustellen ist außerdem die unerwartet hohe Übereinstimmung der Architekturen der Kontrollgruppe mit der erwarteten Architektur. In Bezug auf die für die Implementierungsstrategie relevanten Elemente waren sie wie gesagt vollständig, aber auch die restlichen Elemente waren auf einem hohen Niveau erkannt worden. Hier kann als eine Ursache die relativ überschaubare Komplexität und gute Strukturierung von CSOM und als eine andere Ursache das bereits vorhandene Vorwissen angenommen werden.

5.3.3.3 Auswertung der Implementierungsstrategien

Qualität und Umfang der Implementierungsstrategien unterscheiden sich stark über die Teams betrachtet, über die Gruppen sind sie jedoch auf ähnlichem Niveau. Der Um-

fang ist in der Experimentgruppe insgesamt etwas geringer. Ein Team hatte anscheinend versucht die nötigen Datenstrukturen und Advices für eine Implementierung zu entwerfen und hat dafür insgesamt zu viel Zeit aufgewendet, sodass am Ende die dokumentierte Strategie sehr lückenhaft war. Statt eines groben Plans wurden Lösungen für Teilprobleme entwickelt.

Auch in der Kontrollgruppe gab es ein Team, dessen Lösung stark von den Erwartungen abwich. Dieses Team lieferte nur eine Liste der Module mit einzelnen Stichworten zu den angedachten Änderungen, nicht jedoch die erwarteten Stichpunkte zum Beschreiben der Strategie in einem Detailgrad, der eine Implementierung erlauben würde. Bei diesem Team handelte es sich aber gleichzeitig um jenes mit der Architektur, die in Bezug auf die erwarteten Elemente vollständig war. Zudem war es das Team, welches die wenigste Zeit insgesamt aufgewendet hat. Hier liegt die Vermutung nahe, dass eines der Teammitglieder bereits zu viel Wissen zum Ziel des Experiments hatte und somit versucht hat, die erwarteten Ergebnisse zu produzieren. Dieser Zusammenhang liegt nahe, da der betreffende Proband im selben Büro wie einer der Durchführende des Experiments arbeitete und bereits vorher Einblick in verschiedene Überlegungen zum Experimententwurf hatte. Dies ist jedoch nur eine mögliche Erklärung. Sie erscheint allerdings plausibel, da die anderen Teams dieser Gruppe sehr umfangreiche und vollständige Implementierungsstrategien entwickelt haben und somit von einer ausreichenden Erklärung der Aufgabenstellung im Vorfeld ausgegangen werden kann. Andere Faktoren wie fehlende Motivation, Aufnahme- und Konzentrationsfähigkeit zur Zeit des Experiments können nicht ausgeschlossen werden, sollten jedoch durch die Arbeit in Teams reduziert worden sein. Eine weitere Interpretationsmöglichkeit ist im Zusammenhang mit der Architektur, die nicht auf die Problemstellung ausgerichtet war, zu sehen. Alle anderen Teams haben bereits auf dieser Ebene eine höhere Problemorientierung erkennen lassen.

Der Aspekt der Modularisierung wurde von den VMADL-Teams wie erwartet besonders gewürdigt und durch Andeutungen in der Strategie auf die anzulegenden Module und der Verwendung von AOP dokumentiert. Bei diesen Teams wird daher die Modularisierung als vollständig betrachtet. In den Beschreibungen der Kontrollgruppe fanden sich wesentlich weniger Modularitätsaspekte. Sie haben keine über die Möglichkeiten von C hinausgehende Modularisierung beschrieben und viele Änderungen direkt für die bestehenden Module vorgeschlagen. Jedoch muss fairerweise angemerkt werden, dass zwei Teams auch die Verwendung von AOP vorgeschlagen haben, nachdem im Fragebogen danach gefragt wurde. Aber die intuitive Implementierung hätte zu einer typischen, wenig modularen Spezialanfertigung geführt. Es kann jedoch davon ausgegangen werden, dass eine anders gestellte Aufgabenstellung hier entsprechend andere Strategien motiviert hätte.

Das Team, welches am schnellsten mit der Dokumentierung seiner Implementierungsstrategie fertig war, arbeitete in der Experimentgruppe und gehörte zu den beiden Teams mit den meisten Punkten in Bezug auf die erwarteten Bestandteile der Strategie. Trotzdem war der Gesamteindruck der Strategie, auch bezogen auf den recht geringen Umfang von einer halben Seite, eher im Mittelfeld. Es fehlten einige Details, die für eine Implementierung wesentlich gewesen wären, wie bspw. die Anpassung der Garbage-Collection.

Drei der erwarteten Elemente wurden von keinem Team erkannt. Dies war die Registrierung der Primitiven, damit sie von CSOM verwendet werden können, ein Mechanismus für kritische Code-Segmente, in denen kein Kontextwechsel durchgeführt

werden darf und ein Bezug auf die Block-Klasse, die für Smalltalk im Zusammenhang mit Threads hätte betrachtet werden sollen. Nur von jeweils einer Gruppe wurde die Initialisierung von globalen Variablen zum Start der VM, eine Anpassung der Verarbeitung von Anweisungen auf der Shell, die Änderung des Root-Sets der GC, die Initialisierung des Schedulers und Mechanismen für Synchronisierung mit bspw. Mutexen oder Semaphoren, angedeutet.

5.3.3.4 Auswertung der Fragebögen

Im ersten Teil des Fragebogens wurden die Probanden nach ihren allgemeinen Programmierkenntnissen gefragt. Im Durchschnitt gaben sie dabei an, seit mehr als 9 Jahren zu programmieren und mehr als 3 Jahre davon in der Wirtschaft z. B. als Werkstudent gearbeitet zu haben. Als vorherrschendes Programmierparadigma wurde die imperative, objektorientierte Programmierung angegeben. Kein Proband verwendet nach eigener Auskunft funktionale Programmiersprachen. Im Rahmen dieser Arbeit sind natürlich die Kenntnisse zu Techniken zur *Separation of Concerns* interessant. Hier hatte ein großer Teil bereits von aspekt- und kontextorientierter Programmierung [HCN08] gehört, aber bis auf drei Probanden keine praktische Erfahrung damit gesammelt.

Ihre Fähigkeit, C-Quelltexte zu verstehen, schätzen die Probanden insgesamt eher gut ein. Ein Proband entwickelt auch regelmäßig mit C und sieht sich selbst als Experte auf dem Gebiet.

Auf die Frage, wie schwierig es war, die Architektur zu ermitteln, wurde dies im Durchschnitt als normal schwer bewertet. Die Probanden aus der Kontrollgruppe waren jedoch nicht so überzeugt vom Detailgrad ihrer Architektur wie die Personen aus der Experimentgruppe, die VMADL verwendet haben.

Als hilfreich erwies sich laut den Probanden die Tatsache, dass sie bereits früher mit dem Quelltext von CSOM in Berührung kamen und auf Vorwissen aus der Vorlesung zur Bewältigung der Aufgabe zurückgreifen konnten. Auf die Frage, wie sehr sich die von ihnen wahrgenommene Architektur von der Strukturierung im Quelltext unterscheidet, antwortete der Durchschnitt im Bereich von *wenig*, bis *etwas*.

Die Probanden der Kontrollgruppe waren überwiegend der Meinung, dass C nicht genug Sprachmittel bereitstellt, um eine geeignete Modularisierung zu erreichen. Daher wünschten sich die meisten die Verwendung einer objektorientierten Sprache zur Implementierung bzw. schlugen Sprachmittel wie Vererbung, Klassen und Polymorphie vor. Gleichzeitig kam aber auch der Kommentar, dass dies nicht die Modularisierung auf Architekturebene verbessern würde.

Die nach den Sprachmitteln von VMADL befragten Probanden waren überwiegend der Meinung, dass diese geeignet sind, um Modularisierung zu erreichen, aber kreuzten nicht den Punkt *sehr geeignet* an. Es muss also davon ausgegangen werden kann, dass hier noch Überzeugungsarbeit geleistet bzw. Erweiterungen vorgenommen werden sollten, für die es jedoch keine konkreten Vorschläge gab.

Zur Durchführung des Experiments befragt, bewerteten die Probanden sowohl Zeitrahmen, Vorbereitung durch den Durchführenden, als auch Vorwissen und zur Verfügung gestelltes Material als passend bzw. gut. Die Aufgabenstellung wurde als klar verständlich bewertet und das eigene Vorwissen im Großen und Ganzen ebenfalls als geeignet angesehen. Die Möglichkeit, die ihnen bekannte Entwicklungsumgebung zu verwenden, wurde ebenfalls als positiv empfunden und sie wurden beim Lesen des Quelltexts nicht durch sie behindert. Im Nachhinein kamen hier jedoch auch Kommentare zur fehlenden Sprachintegration von VMADL in die jeweilige IDE. Für den

gesetzten Zeitrahmen war ihrer Meinung nach die Teamarbeit förderlich und sie nehmen an, dass sie ohne sie nicht so gut zurechtgekommen wären.

5.3.3.5 Bei der Durchführung aufgetretene Probleme

Das größte Problem bei der Durchführung des Experiments war das Fehlen eines konkreten Plans für den Fall, dass Probanden nicht wie vereinbart zum Experiment erscheinen. Ein Proband der Kontrollgruppe kam etwas verspätet, direkt nach dem Einführungsvortrag. Dies kann jedoch als vernachlässigbar eingestuft werden, da er vom Durchführenden nachträglich instruiert wurde und die Ergebnisse des Teams sonst keine weiteren Auffälligkeiten aufwiesen.

Wesentlich kritischer für die gesamte Bewertung der Ergebnisse war jedoch das Fernbleiben von zwei Probanden. Hier gab es keinen konkreten Plan, wie mit dieser Situation umgegangen werden sollte. Zudem sind beide Probanden im Vorfeld der Experimentgruppe zugeordnet worden. Es wurde aufgrund dieser nicht vorhergesehenen Situation die Entscheidung getroffen, die beiden betroffenen Teams zunächst mit nur je einem Probanden arbeiten zu lassen. Dies führte jedoch zu erheblichen Schwierigkeiten.

Wie sich während des Experiments herausstellte, war die vorbereitete Projektdatei für Microsofts Visual Studio nicht für die Aufgabenstellung geeignet, da sie nicht die Verzeichnisstruktur des Projekts abbildete und zusätzlich die VMADL-Dateien nicht mit anzeigte. Von den Teams, die zu zweit arbeiten, wurde dieses Problem schnell erkannt und gelöst. Bei einem der Probanden, der allein arbeiten sollte, fiel dieses Problem jedoch erst nach 30 min auf. Daher sind die aufgenommenen Zeiten sehr wenig aussagekräftig. Zusätzlich wurde erst weitere 20 min später die Entscheidung getroffen, die beiden einzeln arbeitenden Probanden zu einem Team zusammenzusetzen. Zu diesem Zeitpunkt war es klar, dass die Aufgaben einfach nicht allein im angesetzten Zeitrahmen schaffbar waren. Die Entscheidung wurde erst so spät getroffen, da somit ein weiterer Datensatz verloren ging. Wären die Aufgaben allein schaffbar gewesen, hätte zwar die Zeit nicht bewertet werden können, aber die Auswertung im Vergleich mit der anderen Gruppe wäre zumindest auf der Qualitätsebene möglich gewesen. So ist nur die zweite Aufgabe in Ansätzen vergleichbar.

Während der Durchführung zeigte sich noch ein weiterer Faktor, der vorher beim Entwurf des Experiments nicht genügend gewürdigt wurde. Die Gruppendynamik in und zwischen den Teams beeinflusste wesentlich den Abgabezeitpunkt und die Fertigstellung der Aufgaben, da sich die Teams aneinander orientierten. In der Kontrollgruppe war dieser Effekt deutlich stärker zu sehen, da einerseits drei Teams im selben Raum arbeiteten und andererseits zumindest zwei Teams gemeinsam zu Mittag essen wollten.

Letztendlich muss daher festgestellt werden, dass zumindest die Betrachtung der Dauer für die einzelnen Aufgaben keine Schlüsse ermöglicht, da die Werte durch zu viele Störgrößen beeinflusst wurden. Aber auch der Vergleich der Qualität der Lösungen ist nur sehr eingeschränkt möglich, da letztendlich ein Datensatz komplett für die Auswertung fehlt und insgesamt die Anzahl der verfügbaren Datensätze zu gering ist.

In der Experimentgruppe ist zudem aufgefallen, dass der Lernaufwand für VMADL deutlich unterschätzt wurde und der Einführungsvortrag nicht ausgereicht hat, um den Probanden das Gefühl zu geben, dass sie souverän mit VMADL umgehen können. Einerseits waren nur beschränkte Kenntnisse zu AOP vorhanden, wodurch sich die Probanden auch noch mit diesen Konzepten auseinandersetzen mussten und anderer-

seits wurde nicht das Verständnis von VMADL für die Bedeutung der Architektur geweckt, sodass die Probanden auf die ihnen bekannten Strategien zur Systemanalyse zurückgriffen, aber für den Versuch die VMADL-Definitionen zu interpretieren zusätzliche Zeit aufwenden mussten.

5.3.3.6 Interpretation der Ergebnisse

Wie bereits erwähnt, ist die Aussagekraft der gewonnen Daten, durch die während der Durchführung aufgetretenen Probleme, sehr stark beeinträchtigt. Die Ergebnisse der Kontrollgruppe können zwar als einigermaßen akkurat angesehen werden, sind jedoch in sich selbst nicht schlüssig und entsprechen nur in Teilen den Erwartungen.

Im Folgenden werden die Daten mit den in Abschnitt 5.3.1 aufgestellten Null-Hypothesen in Zusammenhang gebracht.

Die erste Hypothese sagte aus, dass die Verwendung von VMADL nicht zu einer konsistenten und gleichmäßigen Wahrnehmung der Architektur führt und dadurch die wahrgenommene Architektur stark zwischen den Teams abweicht. Diese Hypothese impliziert die Annahme, dass die wahrgenommene Architektur generell stark variiert, da sie nur schwer aus dem Quelltext zu entnehmen ist, wenn kein VMADL verwendet wird. Wie sich herausgestellt hat, lassen sich die Daten so interpretieren, dass dieses Problem den Probanden durchaus zumindest unterbewusst klar ist, sie aber eine Strategie haben, um trotzdem zum Ziel zu kommen. Die dokumentierten Architekturen waren klar problemorientiert und speziell auf die Aufgabenstellung zugeschnitten. Man kann so schon sagen, dass Architekturen voneinander abwichen, da sie diese nicht durchweg auf der selben Abstraktionsebene dargestellt wurde und jedes Team andere Schwerpunkte gesetzt hat. Dies hat sich aber auch nicht durch die Verwendung von VMADL verändert. Einerseits lag dies vermutlich an der fehlenden Vertrautheit mit der Sprache, andererseits aber auch an der bewährten Strategie zur Entwicklung einer problemorientierten Architekturdarstellung.

Die zweite Hypothese zielte darauf ab, einen Zusammenhang zwischen der Verwendung von VMADL und der Dauer der Analyse und Dokumentation des Systems herzustellen. Mit den gewonnenen Daten müsste man dem Wortlaut der Null-Hypothese zustimmen und sagen, dass die Verwendung von VMADL tatsächlich keine positive Auswirkung hat. Dies ist jedoch vermutlich der zu geringen Vorbereitung der Probanden auf die Sprache geschuldet und erscheint insgesamt nicht als zwingend.

Die dritte Hypothese bezieht sich auf die Entwicklungsdauer der Implementierungsstrategie. Auch hier müsste festgestellt werden, dass es keine positiven Auswirkungen gibt, die aus den gewonnenen Daten abgeleitet werden dürften. Tatsächlich ist der Mittelwert für die Dauer bei der Experimentgruppe mit 35 min knapp 8 min geringer als bei der Kontrollgruppe mit 43 min. Jedoch sind die gewonnenen Daten zu stark durch die verschiedenen Störgrößen verfälscht und erlauben diesen Schluss nicht.

Mit der vierten Hypothese soll die Auswirkung auf die Umsetzbarkeit und Korrektheit der Strategie bewertet werden. Wenn zur Betrachtung der dazu subjektiv gebildete Wert zur Einschätzung dieses Kriteriums herangezogen wird, gibt es keinerlei Auswirkung und auch diese Null-Hypothese wäre zu bejahen. Insgesamt konnte keine der entwickelten Strategien die erwartete Vollständigkeit erreichen und die meisten Strategien waren auch an kritischen Punkten, wie der Interaktion mit der GC, unvollständig. Auf der Basis der gewonnen und qualitativ nicht ausreichenden Daten sollen hier daher auch keine weiteren Schlüsse gezogen werden.

Ein einziger Punkt, der nachvollziehbar durch VMADL motiviert wurde, ist die Modularisierung, was durchaus als qualitative Verbesserung begriffen werden kann. Ein nicht zu unterschätzender Einfluss geht bei diesem Aspekt aber auch von der Aufgabenstellung aus, die absichtlich keinen Verweis auf Modularität enthielt. Dadurch war die Kontrollgruppe in gewisser Weise benachteiligt, da hier nicht die Sprachmittel und somit nicht die Motivation zur Modularisierung gegeben waren.

5.3.3.7 Schlussfolgerungen für weitere Experimente

Ein wesentlicher Schluss kann aus dem durchgeführten Experiment auf jeden Fall gezogen werden. Die aufgetretenen Probleme sind vermeidbar und können durch Anpassungen am Experimententwurf vermieden werden. Jedoch ist es gleichzeitig sehr schwer, geeignet qualifizierte Probanden für solch ein Experiment im Bereich virtueller Maschinen zu finden.

Für Experimente sollte im Allgemeinen ein Katalog von möglichen Ereignissen, die während der Durchführung auftreten können, angelegt werden. Dieser Katalog sollte ebenso die Gegenmaßnahmen beinhalten, damit der Durchführende bei auftretenden Problemen geeignet reagieren kann und die Reaktionen in allen Gruppen konsistent sind.

Des Weiteren sollte der Lernaufwand für eine neue Technik nicht unterschätzt werden. Es muss eine Möglichkeit gefunden werden, die Probanden mit der Technik so vertraut zu machen, dass sie souverän damit umgehen können, für die Kontrollgruppe aber keinen Nachteil durch den Wissensunterschied entsteht.

Abgesehen davon, dass es unter Umständen problematisch ist, geeignete Probanden zu finden, war die Anzahl der Datensätze, die durch dieses Experiment gewonnen wurden, zu gering. Das Experiment wurde so entworfen, dass einerseits genügend Freiwillige gefunden werden konnten, aber andererseits der Zeitaufwand, den diese investieren mussten, möglichst gering ist. Die daraus resultierende Teamarbeit hat aber natürlich die Anzahl der Datensätze halbiert. Hier muss vermutlich ein anderer Ansatz gewählt werden. So wäre eine Integration in den Vorlesungsprozess denkbar. Wie solch ein Experiment aussehen könnte und welche Probleme dabei auftreten können, wurde in einer Artikelreihe zur Durchführung empirischer Studien sehr umfangreich diskutiert [SC02, BT03, SC03].

Für Folgeexperimente im Bereich der Architekturwahrnehmung sollte der Aspekt der problemorientierten Architekturdarstellung noch genauer untersucht werden. Ein Zusammenhang mit der verwendeten Notation ist eigentlich nicht zu vermuten, aber eventuell mit der Methodik, die im Zusammenhang mit der Notation vermittelt wurde. Die Gruppe mit der am wenigsten problemorientierten, aber dafür in Bezug auf die Erwartung vollständigen Architektur, setzte UML ein. Hier könnte vermutet werden, dass UML von den Studenten zumeist für im Quelltext klar erkennbare Strukturen, wie Klassen eingesetzt wird und daher weniger problemorientiert ist als FMC, welches als Mittel zur Kommunikation gelehrt wurde und nicht zwingende auf eine Implementierung bzw. Quelltext abgebildet werden kann. Um die gewünschten Ergebnisse zu erhalten, wäre hier also eventuell eine Einschränkung auf UML hilfreich gewesen, da die Studenten sich in diesem Zusammenhang eventuell eher auf die tatsächlichen Quelltextkonstrukte beschränkt hätten. Andererseits hätte dies vermutlich bei der Erstellung der Implementierungsstrategie hinderlich sein können, da die aufbereitete Architektur dann zu wenig detaillierte Informationen enthalten würde.

5.4 GESAMTEINDRUCK

Die Analyse der Fallstudie mit verschiedenen Metriken hat ergeben, dass VMADL keinen negativen Einfluss auf die Gesamtkomplexität des Systems und dessen Laufzeiteigenschaften hat. Insgesamt konnte nachgewiesen werden, dass die komplette Modularisierung von Features durch VMADL erreicht werden kann. Daraus ergibt sich die nötige Variabilität für die Entwicklung einer Produktfamilie. Gleichzeitig wurden aber auch die Architekturaspekte expliziter dargestellt. Dies spiegelt sich in einer minimalen Anzahl zusätzlicher Quelltextzeilen wider, hilft den Entwicklern aber beim Verstehen des Systems, da die Module und ihre Interaktionen an klar definierten Stellen im Quelltext abgebildet werden.

Die zusätzliche Komplexität, die durch die Verwendung von AOP- und FOP-Sprachmitteln in VMADL entsteht, wurde hier nicht näher betrachtet, da bspw. die *AspectJ Development Tools* [Ecl08] für die Eclipse-IDE oder die Arbeit von Quitslund et al. [QMHB04] zeigen, wie solche Sprachmittel in einer Entwicklungsumgebung unterstützt werden können und die Komplexität handhabbar machen.

Die Ergebnisse des durchgeführten Experiments haben leider nicht die notwendige Qualität, um daraus Schlüsse ziehen zu können. Es gibt ein nicht verlässliches Anzeichen, dass VMADL die Wartbarkeit erhöht, da die Entwickler durch die explizite Architektur schneller eine Strategie zur Lösung eines Problems entwickeln können. Jedoch sind die Daten durch viele Punkte beeinflusst worden und nicht umfangreich genug, um dies mit Bestimmtheit sagen zu können.

Letztendlich bleibt daher vor allem der subjektive Eindruck, dass die Sprache ein angenehmeres Arbeiten mit dem System ermöglicht, da die Architektur expliziter dargestellt wird und die Modulschnittstellen konkreter sind, als sie das in einer reinen C-Implementierung sein können. Es fällt dadurch leichter ein neues Modul in das System zu integrieren, da die Interaktionen schneller zu finden sind und auch von vornherein klar ist, wo ein Modul auf andere Module abgestimmt werden muss und wie dies implementiert werden kann.

Die Sprache ermöglicht damit auch, ein System aus Service-Modulen zusammenzusetzen und wenn es nötig wird, neue Module hinzuzufügen, ohne die bestehenden Module abändern zu müssen und somit kann die bestehende Systemarchitektur unbeeinflusst bleiben. Die Grundstruktur des Systems kann somit auch über längere Zeit und trotz stetiger Wartung erhalten bleiben, da Erweiterungen bereits auf Sprachebene modularisiert werden können und neue Module das System nur ergänzen, ohne es zu ändern.

6 Verwandte Arbeiten und Technologien

6.1 Separation of Concerns

6.1.1 Aspektorientierte Programmierung

Mit der 1997 vorgeschlagenen Technik der aspektorientierten Programmierung (AOP) [KLM+97] wurde die Forschung auf dem weiten Feld der *Separation of Concerns* neu angeregt.

Im Bereich der Aspekt- bzw. Pointcut-Sprachen sind darauf aufbauend viele verschiedene Ideen entwickelt worden, um auch für eine Vielzahl von besonderen Anwendungsfällen die Mittel der aspektorientierten Programmierung verwenden zu können. Relevanter für diese Arbeit ist jedoch die Forschung zu den Einsatzmöglichkeiten von AOP. So wurde untersucht, inwieweit AOP genutzt werden kann, um die Implementierung von Betriebssystemen zu erleichtern. Spezielle Themen waren dabei die Frage, inwiefern sich Betriebssystemkomponenten mit AOP unabhängig von einer speziellen Kernelarchitektur implementieren lassen [SL04], wie sich die Verwendung von AOP auf die Performance auswirkt [ASMT07] und wie sich die Wartbarkeit und Möglichkeiten zur Weiterentwicklung von Betriebssystemen verbessern lassen [Coa03].

Für die vorliegende Arbeit wurde, wie bereits in der Einleitung erwähnt, auf Open Modules [Ald05] und Crosscutting Interfaces [GSS+06] aufgebaut. Für die eigentliche Implementierung wurde AspectC++ [SGSP02] verwendet. Um keine Performancenachteile in Kauf nehmen zu müssen, wurden jedoch nur statisch auflösbare Sprachkonstruktionen verwendet, sodass letztendlich auf Sprachkonstrukte wie `cflow` für eine kontrollflussbasierte Beschreibung von Pointcuts verzichtet wurde. Die Komplexität der Pointcuts ist damit sehr gering.

Auf eine genaue Untersuchung dieser Komplexität wurde im Rahmen dieser Arbeit jedoch verzichtet. Mit der Arbeit von Bartolomei et al. [BGSF06] gibt es jedoch Ansätze in dieser Richtung weiterzugehen. Letztendlich wäre es interessant eine Möglichkeit zu haben, die Komplexität zu bewerten, die durch die Aufteilung in Aspekte oder im Fall von VMADL in Service-Module entsteht, um abschätzen zu können, inwiefern dies wiederum negative Auswirkung auf die Wartbarkeit hat.

6.1.2 Featureorientierte Programmierung

Die featureorientierte Programmierung [Pre97] ist ebenfalls eine Technik zur *Separation of Concerns*. Der in Bezug auf VMADL und ClassDL wichtige Ansatz ist die vollständige Lokalisierung der Implementierung eines konkreten Features. Damit ist es möglich, die für VMADL gewünschte Modularität zu erreichen und ein Service-Modul als vollständige Einheit der Implementierung eines Features zu betrachten. Da keine Implementierungsfragmente des Features über das System verteilt werden müssen, wird es möglich optionale Service-Module komplett aus dem System zu entfernen, wenn diese nicht benötigt werden.

Die Lokalisierung der Implementierung wird dabei durch die partielle Definition von Klassen im Kontext eines Features erreicht. Dies wird in der Literatur auch als heterogener Crosscut bezeichnet [Bat06, SB02, ALRS05a]. Zu den Sprachen die solche Möglichkeiten bieten gehören unter anderem FeatureC++ [ALRS05b] und CaesarJ [AGMO06].

Auch die in VMADL verwirklichte Kombination von aspektorientierter und featureorientierter Programmierung wurde bereits früher von Apel et al. vorgeschlagen [ALS06]. In VMADL stehen jedoch nicht alle der vorgeschlagenen Möglichkeiten zur Verfügung. So ist das Verfeinern eines Aspekts bzw. eines *Aspectual Mixin Layers* nicht möglich, da es in dem Sinne keine Aspekte als eigene Konstrukte gibt und eine hierarchische Beziehung zwischen den Service-Modulen momentan nicht vorgesehen ist. VMADL ist hier klar auf die Architektursicht beschränkt und bietet stattdessen eine klare Trennung von Schnittstellenbeschreibung, Service-Modul-Interaktion und der Implementierung der Service-Module.

6.1.3 *Kontextorientierte Programmierung*

Mit der von Hirschfeld et al. [HCN08] entwickelten kontextorientierten Programmierung (COP) steht ein weiterer Ansatz zur *Separation of Concerns* bereit. Eines der wichtigsten Ziele bei der Entwicklung dieser Technik war es die Möglichkeit zu schaffen, ein System dynamisch komponieren zu können. Es soll zur Laufzeit die Funktionsweise des Systems abhängig vom aktuellen Anwendungskontext, also aller zur Verfügung stehenden Daten zur Umgebung und zum System, angepasst werden können. Zu diesem Zweck wird das System in Schichten bzw. *Layer* dekomponiert, die dann dynamisch, abhängig vom Kontext, im System aktiviert werden können, um das zusätzliche, vom Kontext abhängige, Verhalten nutzen zu können.

Die verschiedenen Implementierungen wie ContextL [CH05] und ContextR [Sch08] lokalisieren eine Layer-Implementierung in der zugehörigen Klasse. Dieser Ansatz verhindert jedoch eine Lokalisierung aller zu einem Layer gehörenden Teile in ein separates Implementierungsmodul, wie dies bei der featureorientierten Programmierung der Fall ist. Abgesehen von den existierenden Implementierungen wäre aber eine Lokalisierung der Layer in eigenständige Module mit der Idee der kontextorientierten Programmierung vereinbar.

Für die Verwendung zur Modularisierung von virtuellen Maschinen scheint COP jedoch nicht direkt die erste Wahl zu sein. Aus konzeptueller Sicht ist ein reines Schichtenmodell für virtuelle Maschinen nicht vorstellbar, da die engen Verzahnungen eine sinnvolle Aufteilung in Schichten sehr erschweren. Die dynamische Komposition von bestimmten Verhalten könnte zumindest für einige Anwendungsfälle interessant sein. Besonders im Bereich der adaptiven Optimierung könnte zusätzliches Verhalten, das nur lokal in einem Thread verwendet wird, ein mächtiges Werkzeug sein, um bspw. Ausführungspfade für besondere Teile einer Anwendung protokollieren und optimieren zu können.

Für die Modellierung der Architektur auf Quelltextebene scheint hingegen die mit VMADL gewählte Kombination von AOP und FOP geeigneter, da diese beiden Techniken zusammen eine klare Modularisierung der Implementierung nach der Zugehörigkeit zu Service-Modulen erlauben und die Mächtigkeit zur Beschreibung von Service-Modul-Interaktionen bieten, ohne andere Service-Module anpassen zu müssen. Dies ermöglicht auch auf statischer Ebene eine klare Abgrenzung zwischen Funktionalität und kann als Grundlage für die Entwicklung von Softwareproduktlinien genutzt werden.

6.2 Bidirektionale Schnittstellen

Das Hauptanliegen von bidirektionalen Schnittstellen, die Bereitstellung von Ereignissen, lässt sich bereits über althergebrachte Mechanismen realisieren. Callback-Handler mit Funktionszeigern sind ein klassischer Ansatz zur Entkoppelung von Teil-

systemen. In objektorientierten Sprachen wird diese Funktionalität bspw. über das Observer-Entwurfsmuster implementiert. Mit der Idee der bidirektionalen Schnittstellen werden Ereignisse jedoch direkt Teil der Schnittstellendefinition, ohne hinter Implementierungsdetails verborgen zu werden. Sie bieten somit eine höhere Abstraktion auf der Sprachebene, als dies die Implementierung über z. B. Callbacks ermöglicht.

Zu den Sprachen, die direkte Unterstützung für Ereignisse in Interface-Beschreibungen und die dazugehörige Infrastruktur bereitstellen, gehören C# [Mic08a] und im weitesten Sinne auch C++ mit der Signal-Slot-Erweiterung des Qt-Frameworks [Tro08]. Hier wird bereits auf Sprachebene die Verwaltung und Aktivierung von Callbacks in Form von Ereignissen bzw. Signalen bereitgestellt. Bei C# ist dies direkt in die Sprache integriert. Qt realisiert dies durch einen speziellen Präprozessor. Beide Ansätze lassen sich jedoch sehr ähnlich verwenden. Die Klassen definieren neben ihren Methoden zusätzlich Ereignisse. Bei C# kann ein Nutzer einen Callback-Handler zuweisen. In Qt gibt es eine `connect`-Funktion, mit der Callbacks registriert werden, die dann bei Bedarf von der globalen Ereignisverarbeitung aufgerufen werden.

Die bidirektionalen Schnittstellen, definiert über Pointcuts in VMADL, sind jedoch durch die Vorteile von AOP im Allgemeinen mächtiger als die klassischen Ansätze, Ereignisse anderen Modulen zur Verfügung zu stellen, da durch die Verwendung einer Pointcut-Sprache das explizite Auslösen eines Ereignisses entfällt und somit Vorteile für die Implementierung bietet.

6.3 Modularisierung virtueller Maschinen

6.3.1 Überblick über verschiedene Ansätze

Wie bereits im Abschnitt 2.2.3 betrachtet wurde, gibt es verschiedene Implementierungsstrategien für virtuelle Maschinen. Die Forschung in diesem Bereich konzentriert sich jedoch auf die grundlegende Herangehensweise zur Implementierung und der Aspekt Architektur wird dabei nur am Rande betrachtet.

Im Bereich domänenspezifische Sprachen für virtuelle Maschinen, zu denen auch VMADL gezählt werden kann, gibt es mit dem Projekt *Vmgen* einen Ansatz die Implementierung von Interpretern erleichtern soll [EGKP02]. Dazu wird eine DSL bereitgestellt, mit der VM-Instruktionen bzw. Bytecodes spezifiziert werden können. Aus dieser Spezifikation kann im Anschluss ein optimierter Interpreter generiert werden. Die anderen Teile der VM werden von diesem Ansatz nicht beeinflusst und nutzen die klassischen Techniken zur Modularisierung.

Da die Verwendung einer maschinennahen Implementierungssprache und die damit klare Trennung von der implementierten Sprache sehr aufwendig ist, wurde auch die Verwendung von Sprachen, die eine Teilmenge der implementierten Sprache bilden, betrachtet. Ein Überblick zu diesem Ansatz wurde bereits in Abschnitt 2.2.3 gegeben. Das PyPy-Projekt [RP06] hat den Weg einer werkzeuggestützten, schrittweisen Transformation gewählt, an deren Ende aus RPython-Quelltext bspw. C-Quelltext erzeugt wird. Die Verwendung einer schrittweisen Transformation ermöglicht es, Teile der VM auf späteren Stufen in das System einzubinden. Als Beispiel wird die Garbage-Collection genannt. Für die Implementierung eines Interpreters können sich die Entwickler in diesem Fall ganz auf die zu implementierende Sprache konzentrieren und die GC wird als Teil der Werkzeugkette bereitgestellt. GC und Interpreter sind damit voneinander unabhängig, wie dies auch mit VMADL erreicht werden kann. Im Unterschied zu VMADL wird diese Modularisierung jedoch über die Implementierung auf

verschiedenen Abstraktionsebenen, die durch die Werkzeugkette definiert werden, bereitgestellt. Auf diesen Abstraktionsebenen selbst können jedoch nur die Modularisierungskonzepte verwendet werden, die RPython bietet und damit ist eine klare Abgrenzung von Funktionalität aus Architektursicht nicht möglich, da Mechanismen, wie sie AOP oder FOP bieten, nicht vorhanden sind.

Für metazirkuläre VMs trifft dies, abhängig von der zu implementierenden Sprache ebenfalls zu. Im folgenden Abschnitt wird dies am Beispiel der Java-VM Maxine näher betrachtet.

6.3.2 MAXINE – EINE METAZIRKULÄRE JAVA-VM

Die Maxine VM [Mat08] ist ein Forschungsprojekt [Sun08c] von Sun, mit dem Ziel eine in Java geschriebene und möglichst gut für die Forschung an VMs geeignete Java-VM zu schaffen. Durch Ausnutzung aller Möglichkeiten, die objektorientierte Programmierung und die verfügbaren Java-Entwicklungswerkzeuge bieten, soll das Prototyping von VMs vereinfacht werden. Um dieses Ziel möglichst komplett mit Java erreichen zu können, ist die VM metazirkulär konzipiert und integriert sich vollständig mit dem Standard-JDK von Sun. Die Implementierung verwendet an vielen Stellen bekannte objektorientierte Entwurfsmuster und macht so viel Gebrauch wie möglich von den in der aktuellsten Java-Version verfügbaren Sprachkonstrukten, wie z. B. Generics, Annotationen und Enumerationen.

Als Besonderheit wird herausgestellt, dass die VM um den JIT-Compiler herum aufgebaut wurde und auf einen Interpreter verzichtet. Dies ermöglicht es, unter Ausnutzung von Annotationen als Compiler-Hinweise sehr effizienten und optimierten Maschinencode zu erzeugen. Gleichzeitig spielen das metazirkuläre Design und die weitgehende Verwendung von Java, eine große Rolle. Dadurch, dass Assembler-Code und C nur an den Stellen verwendet werden, für die es absolut notwenig ist, werden viele Probleme vermieden, die in klassischen VM-Implementierungen [USA05], wie z. B. auch in CSOM, auftreten. So entfällt z. B. die mehrfache Implementierung verschiedener Systemfunktionalität. Ein Beispiel ist die Implementierung eines Array-Zugriffs. Bei klassischen Ansätzen muss dieser in einer einfachen Version und jeweils in den verschieden optimierten JIT-Compilern implementiert werden. Dies entfällt idealerweise in einer metazirkulären VM. Ein anderes Beispiel sind Werkzeuge zum Debugging, um den aktuellen Zustand der VM zur Laufzeit inspizieren zu können. Diese werden bei herkömmlichen Implementierungen zusätzlich zur VM implementiert und erreichen teilweise eine ähnliche Komplexität bei relativ ähnlich Funktionalität. Hier ist dies jedoch nicht nötig, da die VM-Implementierung für viele dieser Aufgaben direkt wiederverwendet werden kann. Dadurch das die implementierte Sprache und die Implementierungssprache identisch sind, gibt es aber noch weitere Vorteile. So ist es bei Maxine bspw. so, dass die GC sich natürlich integriert und alle in Java geschriebenen Teile automatisch unter ihrer Verwaltung stehen. Zusätzlich ermöglicht die gemeinsame Sprachebene zusätzliche Optimierungen durch den JIT-Compiler, da es möglich ist, den Programmcode der VM direkt in Anwendungscode hinein zu optimieren.

Um trotz dieser Implementierungsstrategie das Standard-JDK verwenden zu können, werden Klassen, die Primitiven enthalten, in Maxime noch einmal abgebildet, wobei die Primitiven direkt in Java implementiert werden und die originalen nativen Methoden aus dem JDK beim Kompilieren ersetzen. Dies wird ähnlich wie die Optimierung über Annotationen gesteuert.

Auf die Verwendung von aspekt- oder featureorientierter Programmierung wird jedoch verzichtet. Stattdessen wird versucht, aus den Konzepten der objektorientierten

Programmierung im Allgemeinen und Java im Besonderen den größtmöglichen Grad an Modularität herauszuholen.

Dabei spielen Interfaces eine große Rolle. Zu den Designprinzipien zählt dabei, dass Features allgemein durch Java-Interfaces zu kapseln sind. Dies gilt besonders für Features, deren Implementierung noch nicht optimal ist und die noch Gegenstand weiterer Forschungsarbeiten sein könnten. Die Interfaces für solche Features werden vom Basisinterface `Scheme` abgeleitet und dann für die bestimmten Features verfeinert. Normale Java-Interfaces haben jedoch gegenüber bidirektionalen Schnittstellen den Nachteil, dass Ereignisse nicht so elegant veröffentlicht werden können. In Maxine wird zumindest für einen Teil solcher Ereignisse ein Monitor verwendet, der ebenfalls zur klassischen Synchronisierung dient. So werden direkt im Interface `MonitorScheme` z. B. Ereignisse für Start und Ende eines Garbage-Collection-Durchgangs vorgesehen. An diesem Interface zeigen sich aber auch bereits andere Mängel in der Modularisierung von Crosscutting Concerns. Neben diesen GC-Ereignissen wird eine Methode `scanReferences` für den GC angeboten, um native Zeiger auf Java-Objekte eines Monitors behandeln zu können. Ein anderes Beispiel ist das `HeapScheme`-Interface, welches die Methode `isGcThread` hat und damit den Threads mit Garbage-Collection vermischt. An dieser Stelle stehen mit Service-Modulkombinationen in VMADL mächtigere Modularisierungsmechanismen bereit.

Insgesamt wird aber die Modularisierung schon sehr weit gebracht und durch die Idee der `Scheme`-Interfaces auch mit den begrenzten Mitteln der OOP ein Eindruck vom Architekturzusammenhang vermittelt. Der Quelltext ist mithilfe von Packages klar strukturiert. Jedes Feature hat ein eigenes Package, wobei Features verschiedene Implementierungen haben können, die dann jeweils in einem Package eine Hierarchieebene tiefer angeordnet werden. Zusätzlich verfügen die Features jeweils über eine eigene Verfeinerung des `Scheme`-Interfaces, um ihre Dienste anderen Komponenten zur Verfügung stellen zu können.

Die Konfiguration, also die Information, welche Implementierung für ein Feature verwendet werden soll, wird über den Java-Classpath erreicht. Beim Start der VM wird eine Instanz der Klasse `VMConfiguration` erzeugt, welche die Instanzen der gewählten Features enthält. Dies hat dann leider den Nachteil, dass diese Klasse für alle neuen Features angepasst werden muss, um diese im System verfügbar zu machen. Eine andere Auswirkung dieses Designs ist eine exzessive Verwendung von *Consultation* [Kni05], um Methodenaufrufe an die richtigen, der aktuellen Konfiguration entsprechenden, Objekte weiterzuleiten. Durch den optimierenden JIT-Compiler ist dies jedoch kein Performanceproblem.

Insgesamt fehlt jedoch ein allgemeiner Architekturüberblick. Hier bietet VMADL deutliche Vorteile, um den Entwicklern ein Verständnis der Zusammenhänge im System zu ermöglichen. In Maxine bleibt letztendlich nur die Möglichkeit, den kompletten Quelltext zu lesen. Jedoch bleibt klar festzustellen, dass der erreichte Grad an Modularität und Codewiederverwendung beachtlich ist. Dies ist einerseits dem metazirkulären Ansatz und der damit verbundenen fast kompletten Umsetzung in Java zu verdanken, andererseits zeigt der konsequente Einsatz von Pattern und der aktuellen Java-Sprachmittel positive Wirkung.

6.4 Architekturbeschreibungssprachen

Wie bereits in Abschnitt 2.4 betrachtet, gibt es ein breites Spektrum an Definitionen für Architektur und ein ebenso breites Spektrum an Architekturbeschreibungssprachen. In

diesem Abschnitt werden daher drei Vertreter aus der großen Fülle von Sprachen herausgegriffen, um VMADL klarer abzugrenzen.

6.4.1 *ArchJava*

Ziel von Aldrich, Chambers und Notkin war es, mit der Entwicklung von ArchJava eine ADL zu schaffen, die nicht die Probleme einer von der Implementierung separierten Sprache aufweist [ACN02]. Sie kritisieren, dass bei den meisten ADLs die Entwickler bestimmte Stilvorgaben einhalten müssen und bestimmte Programmieridiome einfach verboten werden, damit die implementierten Komponenten für die ADL geeignet sind. Zudem geht bei abstrakten ADLs oft die Verbindung zur Implementierung verloren und die Architekturbeschreibung wird zur Implementierung inkonsistent.

Um diese Probleme zu umgehen, ist ArchJava eine Erweiterung von Java. Damit ist die Architektursprache gleichzeitig die Implementierungssprache und die benannten Probleme werden vermieden. Für VMADL gilt dies genauso. Durch die Integration mit einer Implementierungssprache werden die durch eine externe ADL verursachten Probleme vermieden und die Architektur ist stets konsistent mit der Implementierung, da die Implementierung die Architektur explizit mit definiert.

Im Gegensatz zu VMADL berücksichtigt ArchJava jedoch auch dynamische Architekturaspekte eines Systems. So gibt es den Gedanken von Laufzeitinstanzen von Komponenten, die dynamisch miteinander verbunden werden können. Dies ist mit VMADL nicht möglich, da sie eine rein statische Sicht auf die Architektur darstellt.

ArchJava bietet Sprachmittel zur Definition von Komponenten und ihrer Kommunikationsschnittstellen sogenannten Ports. Kompositionen von Komponenten können statisch, aber auch zur Laufzeit erstellt werden. Beschrieben werden sie über eine explizite Verknüpfung von Ports. Im Vergleich zu VMADL hat dies sowohl Vor- als auch Nachteile. Zu den Vorteilen zählen die Ausdrucksstärke und die explizite Definition von Kommunikationskanälen zwischen den Komponenten. Mit AOP und VMADL ist jedoch der Grad der möglichen Modularität und die *Separation of Concerns* besser möglich. Dies ist bei ArchJava nicht gegeben, da die Konfiguration direkt im Quelltext vorgenommen wird und so zumindest bei Verwendung der statischen Komponentenverbindung die Flexibilität, die VMADL bietet, nicht erreicht wird. Zudem sind mit den Sprachmitteln der AOP die bekannten Vorteile verbunden.

6.4.2 *Rapide*

Luckham und Vera haben beim Entwurf ihrer Architektursprache besonderen Wert darauf gelegt, dass die Sprache dafür genutzt werden kann, Prototypen bereits früh auf ihre Kommunikationseigenschaften und ihr Verhalten im Zusammenspiel mit anderen Systemteilen hin untersuchen zu können [LV95]. Rapide ist als ausführbare ADL entworfen worden, um sogenannte *Interface Connection Architectures* zu beschreiben.

Ihr Ziel war es mit möglichst wenig Aufwand Prototypen implementieren zu können, um Analysen zu ermöglichen, die eine Vorhersage des Verhaltens vor einer Integration in das Gesamtsystem erlauben. Die Sprache stellt dazu die Möglichkeit bereit, Schnittstellen und Nachrichtenaustausch modellieren zu können. Dieses Modell ist letztendlich eine Menge von Einschränkungen, die eine Menge von Systemen beschreibt, die diesen Einschränkungen entsprechen würden. Ähnlich wie ArchJava werden hier auch dynamische Aspekte betrachtet und die Anzahl und Verbindungen von Komponenten können zur Laufzeit variieren.

Im Vergleich zu VMADL ist Rapide jedoch nicht mit der Implementierung verbunden. Im besten Fall soll man aus einem beschriebenen Modell ein System

synthetisieren können, aber die grundlegenden Probleme bleiben damit trotzdem bestehen, da die Architekturbeschreibung auf lange Sicht nicht mit der Implementierung mitgepflegt wird.

6.4.3 DAOP-ADL

Die *Dynamic Aspect-Oriented Platform ADL* (DAOP-ADL) wurde als Sprache zur Konfiguration dynamischer Komponentensysteme entworfen. Nach Pinto, Fuentes und Troya ist sie dazu gedacht mit ihrer DAOP zusammen verwendet zu werden und die Verbindungen zwischen den einzelnen Komponenten, die auf dieser Plattform laufen, zu beschreiben [PFT03]. Sie soll damit die Lücke zwischen Entwurf und Implementierung von komponenten- und aspektorientierten Anwendungen schließen.

Die Ausführungsplattform für die Komponenten bietet eine Infrastruktur für einen Nachrichtenaustausch zwischen den Komponenten. Dieses Konzept scheint die wesentliche Grundlage für das System zu sein und umfasst sowohl asynchronen als auch synchronen Nachrichtenversand. Die in der ADL formulierten Komponentenschnittstellen beschreiben sowohl die Nachrichten, die verstanden werden, d. h. die angebotene Schnittstelle, als auch die Nachrichten, die versendet werden, also die benötigte Schnittstelle.

Die Implementierung dieser Schnittstellen wird dann durch Komponenten bereitgestellt, wobei ihre Implementierungen nicht mit der ADL zusammenhängen. Sie werden zur Laufzeit durch die DAOP mithilfe der DOAP-ADL untereinander verbunden. Die DAOP-ADL bietet auch die Möglichkeit für die Nachrichten, zwischen den Komponenten in aspektorientierter Form Transformationen zu definieren, um die Komposition anpassen zu können.

Die grundsätzlichen sprachlichen Möglichkeiten scheinen daher ähnlich zu sein, auch wenn der Ansatz wieder losgelöst vom Quelltext ist und die Architektur des Gesamtsystems den Entwickler der Komponenten verborgen bleiben kann. Interessant sind die Hierarchiebeziehungen zwischen Modulen. Diese könnten auch für VMADL eine relevante Erweiterung für komplexe Systeme sein. In einfachen Systemen wie CSOM lässt sich das Fehlen dieses Konzepts jedoch auch durch eine geeignete Verwendung von AOP kompensieren.

6.5 KOMPONENTENSYSTEME

Die oben beschriebene DAOP-ADL ist eine ADL für ein spezielles Komponentensystem. Zu dieser Art von Systemen könnten in einer sehr weit gefassten Interpretation auch CORBA und Web Services gehören, die letztendlich Anwendungen bestehend aus einzelnen Komponenten oder Services bereitstellen sollen. In solchen Systemen kann man jedoch im Allgemeinen von einer anderen Granularität der Komponenten ausgehen.

Etwas vergleichbarer erscheint das OSGi-Framework, welches eine Java-basierte Service Plattform bereitstellt [Nor07]. Relevant in Bezug auf diese Arbeit ist die Absicht, eine Plattform zu schaffen, auf der beliebige Komponenten miteinander kombiniert werden können. Die selbe Absicht wird letztendlich mit VMADL verfolgt, da es das Ziel ist, VMs als Produktfamilien zu implementieren. Im Unterschied zu VMADL, versucht das OSGi-Framework dies jedoch zur Laufzeit zu erreichen. Das standardisierte Framework stellt dazu verschiedenste Funktionalität bereit, die es auch erlaubt Komponenten in verschiedenen Versionen miteinander zu kombinieren oder zur Laufzeit zu ersetzen. VMADL ist da deutlich beschränkter, da es die Flexibilität nur zur Entwurfszeit bietet.

Problematisch am OSGi-Framework ist jedoch die nur implizite Verwirklichung einer Architektur. In diesem Framework gibt es keine Möglichkeit eine Architektur zu beschreiben. Die Mechanismen zum Kombinieren von Komponenten stehen so nur zur Laufzeit zur Verfügung und die Komponentenimplementierungen sind zur Entwicklungszeit einfach nur unabhängige Artefakte, deren tatsächliche Interaktionen erst zur Laufzeit sichtbar werden. Die Architektur müsste in solchen Systemen erst nachträglich wieder ermittelt werden. Antkiewicz hat in dieser Richtung geforscht und versucht die Modelle hinter solchen Frameworks aus dem Anwendungsquelltext zu extrahieren [ABC07]. Funktionierende Ansätze, die Architektur automatisch zu ermitteln, scheint es jedoch nicht zu geben und daher sind Ansätze wie bspw. VMADL der momentan besser geeignete Weg eine mit der Implementierung konsistente Architekturbeschreibung zu erhalten.

7 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es die Sprachmittel der *Virtual Machine Architecture Definition Language* zu untersuchen und soweit notwendig zu erweitern, um einerseits die Darstellung der Systemarchitektur in der Implementierung zu verbessern und somit das Systemverständnis und die Wartbarkeit zu erhöhen. Andererseits sollte es möglich werden, die Modularisierung in virtuellen Maschinen soweit zu verbessern, dass eine VM-Produktfamilie entwickelt werden kann. Zu diesem Zweck wurden ein VMADL-Compiler implementiert, VMADL auf CSOM mit verschiedenen Feature-Implementierungen angewendet und anschließend die Ergebnisse untersucht.

Der erste Schritt dieser Arbeit bestand in der Betrachtung der für die Implementierung von virtuellen Maschinen relevanten Aspekte. Die typische hohe Komplexität wird von den vielfältigen Anforderungen an VMs verursacht und führt letztendlich zur Implementierung von Spezialanfertigungen, die auf einzelne Einsatzgebiete wie eingebettete Systeme oder Serveranwendungen optimiert sind. Dies führt jedoch zu einem hohen Entwicklungsaufwand und wenig Wiederverwendung. Die bisher vorgeschlagenen unterschiedlichen Implementierungsstrategien konnten dies nicht auffangen. Die Ursachen werden durch die Betrachtung der allgemeinen Architektur einer virtuellen Maschine deutlich. Die Analyse der Bestandteile und ihrer Interaktionen zeigt, dass die Modularisierung von VMs im Allgemeinen nur gering ist. Die verwendeten Implementierungssprachen für VMs bieten bisher nicht die notwendigen Konzepte, wie sie unter anderem in der aspekt- und featureorientierten Programmierung zur Verfügung stehen. Ohne den Einsatz dieser Sprachmittel sind die gebildeten Module zu stark miteinander verflochten. Dies wird durch Optimierungen z. B. auf Speicherbedarf für eingebettete Systeme oder Ausführungsgeschwindigkeit bei Desktopanwendungen noch weiter verstärkt, sodass die Wartung sehr aufwendig ist, da die Struktur und die Interaktionen zwischen einzelnen Modulen nur schwer von den Entwicklern erfasst werden können.

Diese Situation motiviert den Bedarf für eine Beschreibungssprache für die Architektur virtueller Maschinen. Da die Begriffe Architektur und Architekturbeschreibungssprachen sehr verschieden definiert werden, gibt es unterschiedliche Ansichten, welche Eigenschaften eine ADL ausmachen sollten. Diese reichen von der Kommunikationssicht, also der anschaulichen Beschreibung von Systemen z. B. mithilfe einer grafischen Syntax, bis hin zur Ansicht, dass eine formale Semantik und Werkzeugunterstützung zur Analyse oder Code-Synthese das Wesen einer ADL ausmachen. Für virtuelle Maschinen wurden aus diesem Spektrum die klare Abgrenzung von Modulen auf der Architekturebene und die gesonderte Beschreibung der Modulinteraktionen als wichtigste Eigenschaften identifiziert. Zusätzlich ist eine Einbettung in die Implementierung und eine an die Implementierungssprache angepasste Syntax und Compiler-Unterstützung sinnvoll, um die Entwickler direkt in ihrem gewohnten Umfeld unterstützen zu können.

Nach dieser Vorbetrachtung folgt im zweiten Schritt die Vorstellung der *Virtual Machine Architecture Definition Language*, die diese Eigenschaften realisiert. Die Grundkonzepte der Sprache sind daher Service-Module mit bidirektionalen Schnittstellen, die sowohl typische Funktionen als auch Ereignisse bereitstellen, und Service-Modulkombinationen, welche auf der Basis der Service-Modul-Schnittstellen die Interaktionen zwischen den Modulen beschreiben. VMADL bildet dazu einen strukturierenden Rahmen für die Implementierung und setzt auf vorhandene Implementierungssprachen auf. Der für diese Arbeit implementierte VMADL-Compiler

ermöglicht eine Verwendung von VMADL zusammen mit C und AspectC++ für die im nächsten Schritt durchgeführte Fallstudie.

Ziel der Fallstudie war die Anwendung von VMADL auf CSOM, um durch eine bessere Modularisierung die Systemarchitektur deutlicher im Quelltext erkennbar zu machen und CSOM in Form einer Produktfamilie entwickeln zu können. Zuerst wurde dazu die Architektur von CSOM durch Reverse-Engineering ermittelt. Im Anschluss daran wurden acht verschiedene CSOM-Erweiterungen analysiert, um die Änderungen im Vergleich zur Ausgangsversion von CSOM zu bestimmen und einzuordnen. Die Änderungen, die nicht als allgemeine Überarbeitung bzw. Verbesserung eingeordnet wurden, sind in den nächsten Schritten mit den durch VMADL zur Verfügung gestellten Modularisierungsmechanismen in Service-Module modularisiert worden. Bevor die eigentliche Modularisierung durchgeführt werden konnte, mussten jedoch verschiedene Änderungen an CSOM vorgenommen werden, damit VMADL mit AspectC++ verwendet werden konnte. Relevant waren hier insbesondere die Anpassungen um CSOM mit einem C++-Compiler übersetzen zu können. Während der Modularisierung der einzelnen Features hat es sich herausgestellt, dass für die in CSOM verwendete OOP-Nachahmung eine eigene Klassenbeschreibungssprache mit featureorientierten Modularisierungstechniken benötigt wurde. Die entworfene ClassDL ermöglicht es, Klassenimplementierungen mit Bezug zu den implementierten Features zu modularisieren. Die sogenannten heterogenen Crosscuts erlauben es Service-Modulen, Felder und Methoden zu bestehenden Klassen hinzuzufügen. Mit dieser zusätzlichen Sprache, die in anderem Kontext durch andere featureorientierte Sprachen ersetzt werden kann, war es im Zusammenspiel mit VMADL möglich, bis auf die Null-basierten Integer alle Features in Service-Module zu modularisieren. Für die Null-basierten Integer stehen in C und den eingesetzten Sprachen nicht die notwendigen Mittel bereit, um die Zeiger-Dereferenzierungsoperation geeignet anpassen zu können. Letztendlich wurde aber das Ziel erreicht, die Modularisierung durch VMADL soweit zu verbessern, dass eine CSOM-Produktfamilie zur Verfügung steht und somit geeignete Konfigurationen gewählt werden können, um ein Exemplar von CSOM, optimiert für ein spezielles Einsatzgebiet, erzeugen zu können. Als besonders vorteilhaft hat sich dabei die getrennte Implementierung der Modulinteraktion erwiesen, da sie so für jede gewünschte Kombination speziell angepasst werden können, ohne die Module selbst verändern zu müssen. Insgesamt wird damit auch die Architektur von CSOM klarer, die Module und ihre Abhängigkeiten bzw. Interaktionen direkt beschrieben werden können.

Der vierte Schritt dieser Arbeit war die ausführliche Evaluierung der in der Fallstudie erreichten Ergebnisse. Zuerst wurden verschiedene Metriken genutzt, um zu zeigen, dass der zusätzliche Quelltext durch VMADL minimal ist und allein durch die strukturierenden Konstrukte entsteht. Diese sind wiederum gewünscht, um das Systemverständnis zu fördern und die gewünschte Variabilität zu erreichen. Die sich daran anschließende Betrachtung der Performanceeigenschaften des Systems konnte belegen, dass die Verwendung von VMADL bzw. AOP keinen negativen Einfluss auf die Performance hat. Insgesamt wurde damit gezeigt, dass VMADL mit den aspekt- und featureorientierten Ansätzen ein geeignetes Mittel ist, um virtuelle Maschinen zu modularisieren und die interne Struktur klarer zu machen, ohne z. B. Performancenachteile in Kauf nehmen zu müssen.

Als zweiter wesentlicher Teil der Evaluierung wurde ein Experiment entworfen, mit dem die Auswirkungen auf die Wahrnehmung der Architektur und die Wartbarkeit

des Systems von VMADL ermittelt werden sollten. Die erhobenen Daten ließen jedoch keine Schlüsse in Bezug auf die betrachteten Hypothesen zu, da sie durch verschiedene Einflüsse kompromittiert wurden. Aus den Ergebnissen des Experiments lässt sich dennoch auf eine unerwartet starke problemorientierte Wahrnehmung der Architektur durch die Probanden schließen, die ihnen eine erfolgreiche Bearbeitung der gestellten Aufgabe erleichterte. Ein Einfluss von VMADL konnte aus den gewonnenen Daten jedoch nicht abgeleitet werden. Allerdings lassen sich die gemachten Erfahrungen für die Vorbereitungen von Folgeexperimenten verwenden, um einerseits den Vorbereitungsaufwand zu reduzieren, aber andererseits auch die aufgetretenen Probleme zu vermeiden.

Zum Abschluss der Arbeit wurden verschiedene verwandte Arbeiten betrachtet und in Bezug zu VMADL gesetzt. Nach einem kurzen Blick auf die Verbindungen von VMADL zur aspekt-, feature- und kontextorientierten Programmierung, wurde ein Blick auf vorhandene Sprachen mit bidirektionalen Schnittstellen geworfen. Daran schließt sich eine Betrachtung von Modularisierungsansätzen für VMs und insbesondere Maxine, eine metazirkuläre Java-VM, an. Hier hat sich gezeigt, das VMADL die gewünschten Vorteile gegenüber reiner objektorientierter Programmierung hat, was auch bei einem kurzen Blick auf die Techniken von Komponentensystemen bestätigt wurde.

Letztendlich konnte mit dieser Arbeit gezeigt werden, dass es mit einer direkt in den Quelltext eines Systems eingebetteten Architektursprache möglich ist, komplexe Systeme wie virtuelle Maschinen geeignet zu modularisieren. Dies bietet die gewünschten Vorteile einer expliziten Architekturdarstellung nicht nur in Bezug auf die Entwicklung, sondern auch in Form einer erhöhten Variabilität. Mit den durch VMADL zur Verfügung gestellten Sprachmitteln lassen sich, wie durch die Fallstudie demonstriert wurden, virtuelle Maschinen in Form von Produktfamilie entwickeln.

Für VMADL und die Compiler-Implementierung gibt es über diese Arbeit hinaus jedoch noch weitere Ideen, die umgesetzt werden könnten. Für einen produktiven Einsatz des Compilers sollte die Konsistenzprüfung deutlich ausgebaut werden. Um den Entwicklern geeignete Hinweise auf Probleme zu geben, sollte der Compiler soweit auf die verwendeten Implementierungssprachen abgestimmt sein, dass er feststellen kann, dass in den `combine`-Konstrukten nur Funktionen und Pointcuts verwendet werden, die in den Schnittstellen der Service-Module beschieben wurden. Dadurch könnten Inkonsistenzen in der Schnittstellenbeschreibung vermieden werden, die momentan noch nicht durch die Werkzeuge aufgedeckt werden.

Als ein weiteres sinnvolles Werkzeug wäre für die Arbeit mit VMADL, neben einer Unterstützung der Sprache in einer IDE, die Navigation im Quelltext mit einer grafischen Repräsentation denkbar. In Abschnitt 3.2 wurde auf Abbildung 3 bereits eine Möglichkeit dargestellt, wie VMADL-Sprachkonstrukte auf grafische Elemente abgebildet werden könnten. Eine solche Navigation würde erlauben eine Übersicht über die Architektur zu bekommen und diese gleichzeitig zur Bewegung im Quelltext zu verwenden. Über einen Mechanismus, der den Entwicklern bei der Verwendung eines solchen Werkzeugs ermöglicht, besondere Teile detaillierter darzustellen, wäre es eventuell sogar möglich die im Experiment aufgefallene problemorientierte Architekturwahrnehmung zu unterstützen.

Über die Erweiterung der Werkzeugunterstützung hinaus erscheinen auch einige Erweiterungen an der Sprache VMADL als sinnvoll. Besonders für komplexe Systeme ist die bisher nicht umgesetzte Verwendung eines automatisierten Initialisierungs-

prozesses sinnvoll. Bisher ist keine Unterstützung für die Startup- und Shutdown-Teile der Schnittstellenbeschreibung implementiert. In komplexen Systemen würde hier die Unterstützung eines Phasenkonzepts jedoch verschiedene Vorteile bieten. In CSOM werden zur Initialisierung verschiedene Pointcuts angeboten, die jedoch semantisch nicht an die Verwendung für die Initialisierung gebunden sind und damit konzeptuell schwächer sind. Zusätzlich entstehen durch die Pointcuts Abhängigkeiten zum *VM*-Service-Modul, die durch ein Phasenkonzept umgangen werden könnten.

In Abschnitt 4.5.5 musste außerdem festgestellt werden, dass VMADL bisher kein Sprachmittel zur Behandlung inkompatibler Service-Module hat. Eine denkbare Möglichkeit dieses Problem zu lösen ist es, die Service-Modulkombinationen auf mehr als zwei Service-Module beziehen zu können, womit es möglich würde, eine Anpassung der Modul-Interaktionen vorzunehmen, welche die Inkompatibilität behebt.

Ein weiteres Konzept, das bisher nicht in VMADL verfügbar ist, aber nützlich wäre, um komplexe Systeme besser beschreiben zu können, ist die hierarchische Verfeinerung von Service-Modulen. Einerseits wäre eine klassische Enthaltensein-Beziehung hilfreich, die beschreibt, dass ein Modul ein Bestandteil eines anderen Moduls ist. Andererseits wäre auch eine Vererbungsbeziehung auf Schnittstellenebene für z. B. das *Memory-Management*-Service-Modul und die verschiedenen GC-Implementierungen denkbar. Eine Enthaltensein-Beziehung würde bedeuten, dass nicht alle Module auf derselben Abstraktionsebene liegen, aber sie würde ein konsistentes Verständnis von Modularisierung ermöglichen. Zudem wäre es dadurch möglich, über alle Granularitätsebenen hinweg von den angebotenen AOP und FOP-Mechanismen zu profitieren. In dem Zusammenhang würde sich das erwähnte Konzept einer grafischen Visualisierung in der IDE vermutlich auch sehr natürlich einpassen.

Unabhängig von den möglichen Erweiterungen kann aber festgestellt werden, dass VMADL in der vorliegenden Form verschiedene Konzepte vereint, die einen hohen Grad an Modularisierung und die Entwicklung von Produktfamilien ermöglichen und gleichzeitig die Systemarchitektur in einer Art und Weise beschreiben, die es Programmierern ermöglicht, die Systemstrukturen einfacher zu erfassen und damit die Wartung von VMs erleichtert.

LITERATURVERZEICHNIS

[AAC+99] Alpern, Bowen ; Attanasio, Clement R. ; Cocchi, Anthony ; Lieber, Derek ; Smith, Stephen ; Ngo, Ton ; Barton, John J. ; Hummel, Susan F. ; Sheperd, Janice C. ; Mergen, Mark: Implementing Jalapeño in Java. In: *ACM SIGPLAN Notices* 34 (1999), Nr. 10, S. 314–324. – ISSN 0362–1340

[AACM07] Ancona, Davide ; Ancona, Massimo ; Cuni, Antonio ; Matsakis, Nicholas D.: RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In: *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–868–8, S. 53–64

[ABC07] Antkiewicz, Michal ; Bartolomei, Thiago T. ; Czarnecki, Krzysztof: Automatic Extraction of Framework-Specific Models from Framework-Based Application Code. In: *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–882–4, S. 214–223

[ACN02] Aldrich, Jonathan ; Chambers, Craig ; Notkin, David: ArchJava: Connecting Software Architecture to Implementation. In: *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–472–X, S. 187–197

[AFG+05] Arnold, Matthew ; Fink, Stephen J. ; Grove, David ; Hind, Michael ; Sweeney, Peter F.: A Survey of Adaptive Optimization in Virtual Machines. 93 (2005), S. 449–466. – ISSN 0018–9219

[AGMO06] Aracic, Ivica ; Gasiunas, Vaidas ; Mezini, Mira ; Ostermann, Klaus: An Overview of CaesarJ. In: *Transactions on Aspect-Oriented Software Development I* Bd. 3880, Springer, 2006 (Lecture Notes in Computer Science), S. 135–173

[aic08] aicas GmbH: *JamaicaVM – Java Technology for Realtime*. http://www.aicas.com/sites/jamaica.html. Version:24. August 2008

[AKVN05] Azevedo, Ana ; Kejariwal, Arun ; Veidenbaum, Alex ; Nicolau, Alexandru: High Performance Annotation-aware JVM for Java Cards. In: *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–091–4, S. 52–61

[Ald05] Aldrich, Jonathan: Open Modules: Modular Reasoning About Advice. In: Black, Andrew P. (Hrsg.): *ECOOP 2005 - Object Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings* Bd. 3586, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–27992–X, S. 144–168

[ALRS05a] Apel, Sven ; Leich, Thomas ; Rosenmüller, Marko ; Saake, Gunter: FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++ / Department of Computer Science, Otto-von-Guericke University. Magdeburg, Germany, 2005. – Forschungsbericht

[ALRS05b] Apel, Sven ; Leich, Thomas ; Rosenmüller, Marko ; Saake, Gunter: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: *Proceedings of Fourth International Conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia*, 2005

[ALS06] Apel, Sven ; Leich, Thomas ; Saake, Gunter: Aspectual Mixin Layers: Aspects and Features in Concert. In: *Proceedings of IEEE and ACM SIGSOFT 28th International Conference on Software Engineering (ICSE'06), Shanghai, China*, ACM, May 2006

[AS07] Adams, Bram ; Schutter, Kris D.: An Aspect for Idiom-based Exception Handling. In: *Proc. of the 5th Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD 2007*. New York, NY, USA : ACM, 2007. – ISBN 1–59593–656–1

[ASMT07] Afonso, Francisco ; Silva, Carlos ; Montenegro, Sergio ; Tavares, Adriano: Applying Aspects to a Real-Time Embedded Operating System. In: *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*. New York, NY, USA : ACM, 2007. – ISBN 1–59593–657–8, S. 1

[AT&08] AT&T Research: *Graphviz - Graph Visualization Software*. http://www.graphviz.org/. Version:15. September 2008

[Bat06] Batory, Don: A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In: *Generative and Transformational Techniques in Software Engineering* Bd. 4143, Springer, 2006 (Lecture Notes in Computer Science), S. 3–35

[BDNW07] Bergel, Alexandre ; Ducasse, Stéphane ; Nierstrasz, Oscar ; Wuyts, Roel: Stateful Traits and their Formalization. In: *Journal of Computer Languages, Systems and Structures* 34 (2007), Nr. 2-3, S. 83–108

[BEA08a] BEA Systems Inc.: *JRockit*. http://www.bea.com/jrockit/. Version:24. August 2008

[BEA08b] BEA Systems Inc.: *Understanding LiquidVM*. http://e-docs.bea.com/wls-ve/docs92-v11/config/lvmintro.html. Version:15. September 2008

[Bel73] Bell, James R.: Threaded Code. In: *Communications of the ACM* 16 (1973), Nr. 6, S. 370–372. – ISSN 0001–0782

[BGS82] Brooks, Rodney A. ; Gabriel, Richard P. ; Steele Jr., Guy L.: S-1 Common Lisp Implementation. In: *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*. New York, NY, USA : ACM, 1982. – ISBN 0–89791–082–6, S. 108–113

[BGS83] Brooks, Rodney A. ; Gabriel, Richard P. ; Steele Jr., Guy L.: LISP-in-LISP: High Performance and Portability. In: Bundy, Alan (Hrsg.): *International Joint Conference on Artificial Intelligence* Bd. 2. Karlsruhe, West Germany : William Kaufman, August 1983, S. 845–849

[BGSF06] Bartolomei, Thiago T. ; Garcia, Alessandro ; Sant'Anna, Claudio ; Figueiredo, Eduardo: Towards a Unified Coupling Framework for Measuring AspectOriented Programs. In: *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–584–3, S. 46–53

[Bor08] Bornstein, Dan: *Dalvik Virtual Machine Internals*. Google I/O 2008. http://de.youtube.com/watch?v=ptjedOZEXPM. Version:20. Juni 2008

[BT03] Berry, Daniel M. ; Tichy, Walter F.: Comments on "Formal Methods Application: An Empirical Tale of Software Development". In: *IEEE Transactions on Software Engineering* 29 (2003), Nr. 6, S. 567–571. – ISSN 0098–5589

[But97] Butenhof, David R.: *Programming with POSIX Threads*. Addison-Wesley, 1997 (Professional Computing Series). – ISBN 0201633922

[CH05] Costanza, Pascal ; Hirschfeld, Robert: Language Constructs for Context-oriented Programming. An Overview of ContextL. In: *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*. New York, NY, USA : ACM, 2005, S. 1–10

[Chr04] Christensen, Larry B.: *Experimental Methodolgy*. 9th. Pearson, 2004. – ISBN 0205393691

[CMU08] CMU: Software Engineering Institute: *How Do You Define Software Architecture?* Web. http://www.sei.cmu.edu/architecture/definitions.html. Version:24. August 2008

[Coa03] Coady, Monica Y.: *Improving Evolvability of Operating Systems with AspectC*, The University of British Columbia (Canada), Doktorarbeit, 2003

[CUL89] Chambers, Craig ; Ungar, David ; Lee, Elgin: An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In: *ACM SIGPLAN Notices* 24 (1989), Nr. 10, S. 49–70. – ISSN 0362–1340

[Cza98] Czarnecki, Krzysztof: *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, Technical University of Ilmenau, Doktorarbeit, October 1998

[DB76] Deutsch, L. Peter ; Bobrow, Daniel G.: An Efficient, Incremental, Automatic Garbage Collector. In: *Communications of the ACM* 19 (1976), Nr. 9, S. 522–526. – ISSN 0001–0782

[DKO+97] Dikel, David ; Kane, David ; Ornburn, Steve ; Loftus, William ; Wilson, Jim: Applying Software Product-Line Architecture. In: *Computer* 30 (1997), August, Nr. 8, S. 49–55. – ISSN 0018–9162

[Ecl08] Eclipse Foundation, Inc.: *AJDT: AspectJ Development Tools*. Project page. http://www.eclipse.org/ajdt/. Version:15. September 2008

[Ede93] Edelstein, D. Vera: Report on the IEEE STD 1219-1993 - Standard for Software Maintenance. In: *SIGSOFT Softw. Eng. Notes* 18 (1993), Nr. 4, S. 94–95. – ISSN 0163–5948

[EGKP02] Ertl, M. Anton ; Gregg, David ; Krall, Andreas ; Paysan, Bernd: Vmgen: a generator of efficient virtual machine interpreters. In: *Softw. Pract. Exper.* 32 (2002), Nr. 3, 265–294. http://portal.acm.org/citation.cfm?id=776235.776238&jmp=citedby&coll=GUIDE&dl=portal,ACM. – ISSN 0038–0644

[Erl00] Erlikh, Len: Leveraging Legacy System Dollars for E-Business. In: *IT Professional* 2 (2000), May/June, Nr. 3, S. 17–23. – ISSN 1520–9202

[FAH+06] Fähndrich, Manuel ; Aiken, Mark ; Hawblitzel, Chris ; Hodson, Orion ; Hunt, Galen ; Larus, James R. ; Levi, Steven: Language Support for Fast and Reliable Message-based Communication in Singularity OS. In: *SIGOPS Oper. Syst. Rev.* 40 (2006), Nr. 4, S. 177–190. – ISSN 0163–5980

[Fow05] Fowler, Martin: *Language Workbenches: The Killer-App for Domain Specific Languages?* http://martinfowler.com/articles/languageWorkbench.html. Version:12. Juni 2005

[FY69] Fenichel, Robert R. ; Yochelson, Jerome C.: A LISP Garbage-Collector for Virtual-Memory Computer Systems. In: *Communications of the ACM* 12 (1969), Nr. 11, S. 611–612. – ISSN 0001–0782

[GL03] Gradecki, Joseph D. ; Lesiecki, Nicholas: *Mastering AspectJ: Aspect-Oriented Programming in Java*. New York, NY, USA : Wiley Publishing, Inc., 2003. – ISBN 0471431044

[Gol72] Goldberg, Robert P.: *Architectural Principles for Virtual Computer Systems*. Cambridge, MA, Harvard University, Doktorarbeit, 1972

[Goo08] Google Inc.: *What is Android?* http://code.google.com/android/what-is-android.html. Version:24. August 2008

[GPF06] Gal, Andreas ; Probst, Christian W. ; Franz, Michael: HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In: *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–332–6, S. 144–153

[GR83] Goldberg, Adele ; Robson, David: *Smalltalk-80: The Language and its Implementation*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1983. – ISBN 0201113716

[GSS+06] Griswold, William G. ; Sullivan, Kevin ; Song, Yuanyuan ; Shonle, Macneil ; Tewari, Nishit ; Cai, Yuanfang ; Rajan, Hridesh: Modular Software Design with Crosscutting Interfaces. In: *IEEE Software* 23 (2006), Nr. 1, S. 51–60. – ISSN 0740–7459

[HAT+08] Haupt, Michael ; Adams, Bram ; Timbermont, Stijn ; Gibbs, Celina ; Coady, Yvonne ; Hirschfeld, Robert: *Disentangling Virtual Machine Architecture*. under review for *IET Journal* special issue on *Domain-Specific Aspect Languages*, 2008

[HCN08] Hirschfeld, Robert ; Costanza, Pascal ; Nierstrasz, Oscar: Context-oriented Programming. In: *Journal of Object Technology (JOT)* 7 (2008), March-April, Nr. 3, S. 125–151

[HCU91] Hölzle, Urs ; Chambers, Craig ; Ungar, David: Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In: *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*. London, UK : Springer-Verlag, 1991. – ISBN 3–540–54262–0, S. 21–38

[Hee08] Heesch, Dimitri van: *Doxygen: Source code documentation generator tool*. http://www.stack.nl/ dimitri/doxygen/. Version:15. September 2008

[HL07] Hunt, Galen C. ; Larus, James R.: Singularity: Rethinking the Software Stack. In: *SIGOPS Oper. Syst. Rev.* 41 (2007), Nr. 2, S. 37–49. – ISSN 0163–5980

[IKM+97] Ingalls, Dan ; Kaehler, Ted ; Maloney, John ; Wallace, Scott ; Kay, Alan: Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In: *ACM SIGPLAN Notices* 32 (1997), Nr. 10, S. 318–326. – ISSN 0362–1340

[JJ05] Jin, Min-Sik ; Jung, Min-Soo: A Study on Fast JCVM by Moving Object from EEPROM to RAM. In: *RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Los Alamitos, CA, USA : IEEE Computer Society, 2005, S. 84–88

[Kaf08] Kaffe.org: *Java Virtual Machine*. http://www.kaffe.org/. Version:24. August 2008

[KGT06] Knöpfel, Andreas ; Gröne, Bernhard ; Tabeling, Peter: *Fundamental Modeling Concepts: Effective Communication of IT Systems*. John Wiley & Sons, 2006. – ISBN 047002710X

[KLM+97] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: *ECOOP'97 — Object-Oriented Programming*, 1997, S. 220–242

[Kni05] Kniesel, Günter: *What is (Not) Delegation*. http://javalab.cs.uni-bonn.de/research/darwin/delegation.html. Version:1. August 2005

[Laf08] Laforge, Guillaume: *Groovy: An agile dynamic language for the Java Platform*. http://groovy.codehaus.org/. Version:15. September 2008

[LV95] Luckham, David C. ; Vera, James: An Event-Based Architecture Definition Language. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 9, S. 717–734. – ISSN 0098–5589

[Mar96] Martin, Robert C.: Granularity. In: *C++ Report* 8 (1996), November-December, Nr. 10, S. 57–62. – ISSN 1040–6042

[Mat08] Mathiske, Bernd: *The Maxine Virtual Machine*. JavaOne 2008 Vortrag. http://developers.sun.com/learning/javaoneonline/j1sessn.jsp?sessn=TS-5169&yr=2008&track=coolstuff. Version:Juni 2008

[McC60] McCarthy, John: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. In: *Communications of the ACM* 3 (1960), Nr. 4, S. 184–195. – ISSN 0001–0782

[Mic08a] Microsoft Corporation: *C# Programmer's Reference: Events Tutorial*. http://msdn.microsoft.com/en-us/library/aa645739(VS.71).aspx. Version:15. September 2008

[Mic08b] Microsoft MSDN: *Fibers*. http://msdn.microsoft.com/en-us/library/ms682661(VS.85).aspx. Version:19. September 2008

[Mir99] Miranda, Eliot: Context Management in VisualWorks 5i / ParcPlace Division, CINCOM, Inc. 1999. – Forschungsbericht

[MO98] McGhan, Harlan ; OConnor, Mike: PicoJava: A Direct Execution Engine For Java Bytecode. Ín: *Computer* 31 (1998), Nr. 10, S. 22–30. – ISSN 0018–9162

[MT00] Medvidovic, Nenad ; Taylor, Richard N.: A Classification and Comparison Framework for Software Architecture Description Languages. In: *IEEE Trans. Softw. Eng.* 26 (2000), Nr. 1, S. 70–93. – ISSN 0098–5589

[Nor07] Normington, Glyn: *JSR-291 Dynamic Component Support for Java SE*. Version:10. August 2007. http://www.jcp.org/en/jsr/detail?id=291

[Obj07] Object Management Group: Unified Modelling Language 2.1.2 Infrastructure Specification / Object Management Group. 2007 (Version 2.1.2). – Specification

[Par92] Park, Robert E.: Software Size Measurement: A Framework for Counting Source Statements / Software Engineering Institute, Carnegie Mellon University. Pittsburgh, Pennsylvania 15213, September 1992 (CMU/SEI-92-TR- 20, ESC-TR-92-20). – Forschungsbericht

[Par07] Parr, Terence: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. First. Pragmatic Bookshelf, 2007 (Pragmatic Programmers). – ISBN 0978739256

[PFT03] Pinto, Mónica ; Fuentes, Lidia ; Troya, Jose M.: DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. In: *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*. New York, NY, USA : Springer-Verlag New York, Inc., 2003, S. 118–137

[PHP08] PHP Group, The: *PHP: Hypertext Preprocessor*. http://www.php.net/. Version:15. September 2008

[Pre97] Prehofer, Christian: Feature-Oriented Programming: A Fresh Look at Objects. In: *Lecture Notes in Computer Science* 1241 (1997), S. 419–434

[QMHB04] Quitslund, Philip J. ; Murphy-Hill, Emerson R. ; Black, Andrew P.: Supporting Java Traits in Eclipse. In: *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA : ACM, 2004, S. 37–41

[RP06] Rigo, Armin ; Pedroni, Samuele: PyPy's Approach to Virtual Machine Construction. In: *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–491–X, S. 944–953

[SAO05] Souza, Sergio Cozzetti B. ; Anquetil, Nicolas ; Oliveira, Káthia M.: A Study of the Documentation Essential to Software Maintenance. In: *SIGDOC '05: Proceedings of the 23rd annual international conference on Design of communication*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–175–9, S. 68–75

[SB02] Smaragdakis, Yannis ; Batory, Don: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. In: *ACM Transactions on Software Engineering and Methodology* 11 (2002), Nr. 2, S. 215–255. – ISSN 1049–331X

[SC02] Sobel, Ann E. K. ; Clarkson, Michael R.: Formal Methods Application: An Empirical Tale of Software Development. In: *IEEE Transactions on Software Engineering* 28 (2002), March, Nr. 3, S. 308–320. – ISSN 0098–5589

[SC03] Sobel, Ann E. K. ; Clarkson, Michael R.: Response to "Comments on 'Formal Methods Application: An Empirical Tale of Software Development'". In: *IEEE Transactions on Software Engineering* 29 (2003), Nr. 6, S. 572–575. – ISSN 0098–5589

[Sch08] Schmidt, Gregor: *ContextR & ContextWiki*. Potsdam, Hasso-Plattner-Institut, Masterarbeit, April 2008

[SDNB03] Schärli, Nathanael ; Ducasse, Stéphane ; Nierstrasz, Oscar ; Black, Andrew P.: Traits: Composable Units of Behavior. In: *ECOOP 2003 – Object-Oriented Programming* Bd. 2743/2003 OGI School of Science & Engineering , Oregon Health and Science University, Springer, November 2003 (Lecture Notes in Computer Science), S. 327–339

[SEP06] Sim, Susan E. ; Easterbrook, Steve ; Perry, Dewayne E.: Case Studies for Software Engineers. In: *28th International Conference on Software Engineering (ICSE'06)* Bd. 0. Los Alamitos, CA, USA : IEEE Computer Society, 2006. – ISBN 1–59593–375–1, S. 1045–1046

[SG96] Shaw, Mary ; Garlan, David: *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1996. – ISBN 0–13–182957–2

[SGG02] Silberschatz, Abraham ; Gagne, Greg ; Galvin, Peter B.: *Operating System Concepts*. 6. Wiley, 2002. – ISBN 0471250600

[SGSP02] Spinczyk, Olaf ; Gal, Andreas ; Schröder-Preikschat, Wolfgang: AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In: *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2002. – ISBN 0–909925–88–7, S. 53–60

[SL04] Spinczyk, Olaf ; Lohmann, Daniel: Using AOP to Develop Architectural-Neutral Operating System Components. In: *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*. New York, NY, USA : ACM, 2004, S. 34

[Smi82] Smith, Brian C.: *Procedural Reflection in Programming Languages*, Massachusetts Institute of Technology, Laboratory for Computer Science, Doktorarbeit, 1982. http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-272.pdf

[SN05] Smith, James E. ; Nair, Ravi: *Virtual Machines: Versatile Platforms for Systems and Processes*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2005 (The Morgan Kaufmann Series in Computer Architecture and Design). – ISBN 1558609105

[Sun99] Sun Microsystems Inc.: The K Virtual Machine (KVM). Version:1999. http://java.sun.com/products/cldc/wp/. 1999. – White Paper

[Sun08a] Sun Microsystems Inc.: *The HotSpot Group*. http://openjdk.java.net/groups/hotspot/. Version:15. September 2008

[Sun08b] Sun Microsystems Inc.: *Java Card Technology*. http://java.sun.com/javacard/. Version:24. August 2008

[Sun08c] Sun Microsystems Inc.: *Maxine: Sun's new meta-circular research VM written in Java*. http://research.sun.com/projects/maxine. Version:12. Juni 2008

[SW00] Strachey, Christopher ; Wadsworth, Christopher P.: Continuations: A Mathematical Semantics for Handling Full Jumps. In: *Higher-Order and Symbolic Computation* 13 (2000), April, Nr. 1, S. 135–152

[Tro08] Trolltech: *Signals and Slots*. http://doc.trolltech.com/signalsandslots.html. Version:15. September 2008

[TWFL98] Turner, C. Reid ; Wolf, Alexander L. ; Fuggetta, Alfonso ; Lavazza, Luigi: Feature Engineering. In: *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*. Washington, DC, USA : IEEE Computer Society, 1998. – ISBN 0–8186–8439–9, S. 162

[USA05] Ungar, David ; Spitz, Adam ; Ausch, Alex: Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment. In: *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–193–7, S. 11–20

[Ves93] Vestal, Steve: A Cursory Overview and Comparison of Four Architecture Description Languages / Honeywell Technology Center. 1993. – Forschungsbericht

ANHANG 1 VMADL-GRAMMATIK

Die folgende Grammatik beschreibt die Sprache VMADL in Form der von ANTLR3 verwendeten Syntax. Diese Syntax orientiert sich im Wesentlichen an der EBNF [Par07].

```
grammar Vmadl;
tokens {
    SERVICE     = 'service';
    STARTUP     = 'startup';
    SHUTDOWN    = 'shutdown';
    EXPOSE      = 'expose';
    ADVICE      = 'advice';
    POINTCUT    = 'pointcut';
    REPLACE     = 'replace';
    REQUIRE     = 'require';
    BOOT        = 'boot';
    FOR         = 'for';
    TYPES       = 'types';
    COMBINE     = 'combine';
    STATIC      = 'static';
    EXTERN      = 'extern';
    CONST       = 'const';
    HEXPREFIX   = '0x';
    RESTRICT    = 'restrict';
}

/**  Parser Rules **/
architecture:        ( NEWLINE | service | combination )* EOF;
interaction:         ( NEWLINE | combination )* EOF;
combination:         COMBINE IDENTIFIER ',' IDENTIFIER LBRACE ( combinationContent )* RBRACE;
service:             SERVICE IDENTIFIER LBRACE ( basicModuleContent )* RBRACE;
basicModuleContent:  require | expose | declarations | types | namedPart | replace | NEWLINE;
combinationContent:  require | advice | startup | shutdown | preprocessor | variable
                       | function | NEWLINE;
literalValue:        IDENTIFIER | DIGIT+ | hexValue;
variable:            (STATIC | EXTERN )? typeRef IDENTIFIER ('[' literalValue ']' )?
                       ( '=' literalValue )? ';';
typeRef:             CONST? IDENTIFIER ('*' )*  RESTRICT? CONST?;
hexValue:            HEXPREFIX (DIGIT | 'a'..'f' | 'A'..'F')+;
function:            (STATIC | EXTERN )? typeRef IDENTIFIER
                       LPARENTHESIS
                         (typeRef IDENTIFIER?  (',' typeRef IDENTIFIER? )* )?
                       RPARENTHESIS (content | ';' );
namedPart:           IDENTIFIER LBRACE ( basicModuleContent )* RBRACE;
replace:             REPLACE qualifiedIdentifier LBRACE ( basicModuleContent )* RBRACE;
advice:              ADVICE adviceHeader content;
adviceHeader:        ( ~(LBRACE | RBRACE | ';') )+ ;
declarations:        preprocessor | declaration ;
preprocessor:        '#' ((~( PPCONTINUE | NEWLINE ))* PPCONTINUE)*
                       (~( PPCONTINUE | NEWLINE ))* NEWLINE;
declaration:         (~('#' | NEWLINE | LBRACE | RBRACE | ';' | SERVICE | STARTUP | SHUTDOWN
                       | EXPOSE | ADVICE | POINTCUT | REQUIRE | BOOT | FOR | REPLACE ))
                       ( ~(LBRACE | RBRACE | ';') )+ (content ';'? | ';' );
contentcontent:      (content | (~(LBRACE | RBRACE))+)*;
content:             LBRACE contentcontent RBRACE;
types:               TYPES content;
startup:             STARTUP (LBRACE (expose | advice | boot )* RBRACE);
shutdown:            SHUTDOWN (LBRACE (expose | advice)* RBRACE);
boot:                BOOT ( ~(LBRACE | RBRACE | ';') )+ ';';
expose:              EXPOSE NEWLINE* LBRACE NEWLINE* ( pointcut )* RBRACE;
require:             REQUIRE IDENTIFIER ';';
pointcut:            POINTCUT (~(';'))* ';' NEWLINE*;
qualifiedIdentifier: IDENTIFIER '.' IDENTIFIER;
```

```
/** Lexer Rules **/
IDENTIFIER:         LETTER ( DIGIT | LETTER | '-' | '_' )*;
DIGIT:              '0'..'9' ;
fragment LETTER:    'a'..'z' | 'A'..'Z' ;
LPARENTHESIS:       '(';
RPARENTHESIS:       ')';
SPECIAL:            ':' | '='| '.' | '*' | '+' | '%' | '#' | '/' | '\\'
                      | '_' | ';' | ',' | '|' | '&' | '-' | '"' | '<' | '>' | '[' | ']'
                      | '!' | '\'' | '?' ;
PPCONTINUE:         '\\' NEWLINE;
NEWLINE:            '\r'? '\n' ;
WS:                 (' '|'\t')+ ;
COMMENT:            '//' (~('\n'|'\r'))* NEWLINE ;
ML_COMMENT:         '/*' (~('*/'))*  '*/' ;
LBRACE:             '{' ;
RBRACE:             '}' ;
```

Anhang 2 Inhalt der beigelegten CD

Die beigelegte CD ist eine Ubuntu-Live-CD. Die passenden Compiler sind installiert und erlauben die beigelegten CSOM-VMs zu kompilieren. Der folgende Verzeichnisbaum gibt einen Überblick über den auf diese Arbeit bezogenen Inhalt:

```
+---Arbeit              – diese Arbeit und die BibTeX-Referenzdatenbank
|   +---dot             – Dot-Visiualisierung der CSOM-Aufrufbeziehungen
|   +---Evaluierung
|   |   +---Benchmarks  – Logfiles, als CSV- und Excel-Dateien aufbereitet
|   |   +---Experiment  – das Material für die Experimentdurchführung
|   |   \---loc-stats   – Rohdaten und Aufbereitung der ermittelten Metriken
|   +---figures         – alle angefertigten Abbildungen
|   \---include-graphs  – mit Doxygen generierte Visualisierung von CSOM
\---source
    +---CSOM
    |   +---branches
    |   |   \---smarr   – Skripte für Benchmarks und Metriken
    |   |       +---aop – CSOM-Implementierung mit AspectC++, Build-Skript
    |   |       |   +---basic, gc, green, image, pthreads, refcount
    |   |       |       tagged-int-one, threading (Fallstudie)
    |   |       |   \---tagged-int-one-bad-aop (Test der Auswirkungen
    |   |       |                   verschiedener AspectC++ Notationen)
    |   |       +---features – CSOM-Überarbeitung für g++, Build-Skript
    |   |       |   \---basic, gc, green, image, pthreads, refcount
    |   |       |       tagged-int-one, tagged-int-zero, threading
    |   |       +---Experiment – Quelltexte fürs Experiment
    |   |       |   +---Experimentgruppe
    |   |       |   \---Kontrollgruppe
    |   |       +---trunk       – Überarbeitungen gegenüber den Hauptzweig
    |   |       \---vmadl (Bench und Build-Skripte)
    |   |           +---classdl
    |   |           \---main
    |   \---trunk– Hauptzweig der CSOM-Entwicklung Stand: Revision 705
    +---CSOM-Viz        – Skript zur Visualisierung von Aufrufbeziehungen
    +---VMADL-C-Aspicere2 – die in Abschnitt 3.3 beschriebene Vorversion
    \---VMADL-Compiler  – Compiler inkl. Werkzeug zum Bestimmen der Metriken
```

ANHANG 3 MATERIAL FÜR DIE DURCHFÜHRUNG DES EXPERIMENTS

Auf den nächsten Seiten sind die Unterlagen angehängt, die den Probanden bei der Durchführung des Experiments zur Verfügung gestellt wurden.

Das Deckblatt des Materials enthält jeweils die Daten des Probanden und die Information in welcher Gruppe und in welchem Team er arbeiten soll. Auf den folgenden Blättern wurde jeweils nur noch eine Kennziffer für den Probanden angegeben, um eine spätere Zuordnung zu ermöglichen, aber gleichzeitig eine blinde Bewertung der Ergebnisse zu ermöglichen. Die beigelegten Übersichtblätter über die UML- und FMC-Notation wurden den Probanden zusammen mit dem ersten Teil des Materials zu Beginn des Experiments ausgehändigt.

Der Fragebogen ab Seite 5 des Materials wurde wie im Entwurf des Designs erwähnt, erst nach der Fertigstellung der ersten beiden Aufgaben ausgehändigt. Die auf der letzten Seite dargestellten Referenzarchitekturen wurden zusammen mit dem Fragebogen ausgehändigt.

Hasso-Plattner-Institut
Universität Potsdam
Fachbereich Software-Architekturen

Experiment CSOM

Material und Fragebogen

Proband: «Nachname», «Vorname»

Gruppe: «Gruppe»

Team: «Team»

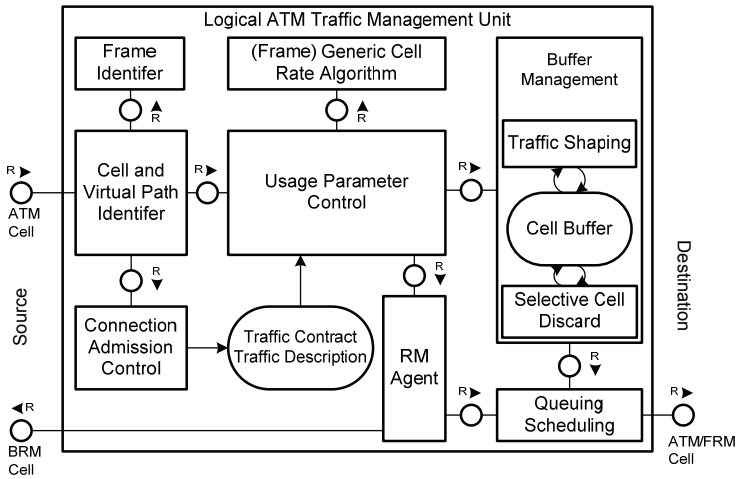
Betreuer: Michael Perscheid

AUFGABENSTELLUNG

1. Analyse des CSOM-Quelltexts

- Ermitteln der Architektur durch Quelltext-Lesen
- Dokumentation der Architektur
  - Selbst gewählte Notation
  - Cheat-Sheets für FMC und UML im Material
  - Fokus auf Compile-Time bzw. Code-Sicht

Beispielarchitektur



Merkmale der Architektur sind

- Wesentliche Komponenten/Teile aus den das System besteht
  - Jeweils für klar abgegrenztes Featurer/Aufgabe zuständig
- Interaktionen zwischen den Komponenten

2. Entwurf einer Implementierungsstrategie für ein Green-Thread-Modul

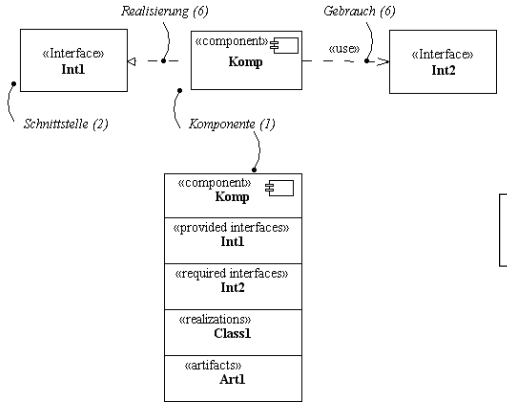
- Anforderungen an Strategie-Beschreibung
  - Strategie soll auf Funktionsebene ansetzten
  - Welche Funktionen und Module müssen angepasst werden?
  - Welches Verfahren wird verwendet um die gewünschte Modularisierung zu erreichen?
  - Exkursion: Green-Threads
- Multi-Threading Emulation
  - Komplett von VM verwaltet, inkl. Scheduling
  - Verwendet, wenn OS keine Threads bereitstellt
    - Oder wenn VM, Interpreter oder verwendete Bibliotheken nicht Thread-safe sind
  - Können effizienter sein als native Threads
    - Kontextwechsel hat weniger Overhead

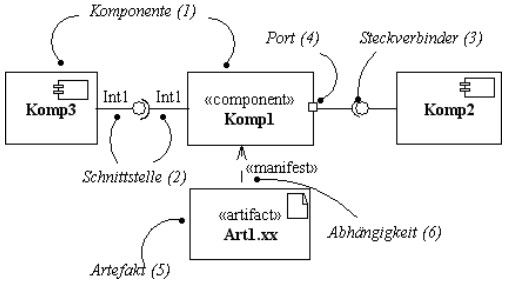
3. Ausfüllen des Evaluierungsfragebogens

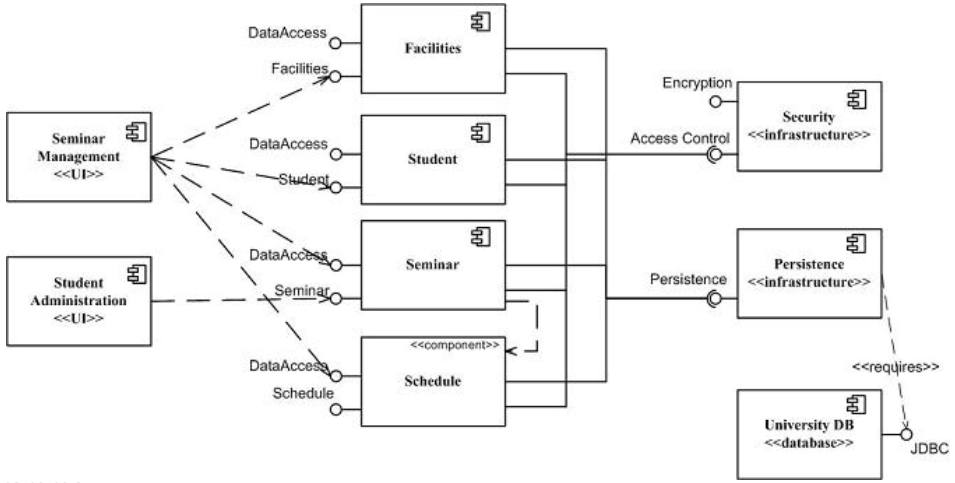
- Besteht aus Fragen zur Experimentdurchführung
  - Probleme, Kritik, mögliche Verfälschungen des Ergebnisses
- Kurze Erhebung von Daten zu Vorkenntnissen
  - Selbsteinschätzung
  - C, Modularisierungsverfahren, AOP
  - CSOM

LÖSUNG AUFGABE 1

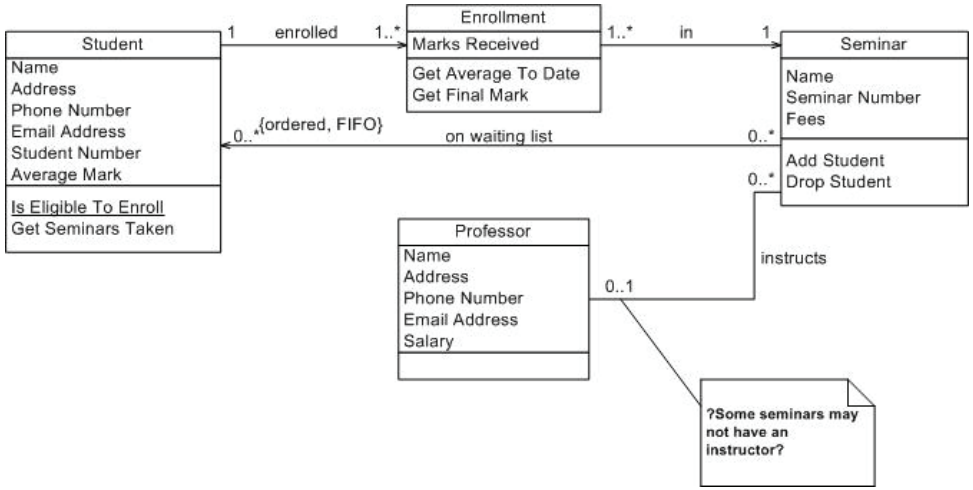
LÖSUNG AUFGABE 2



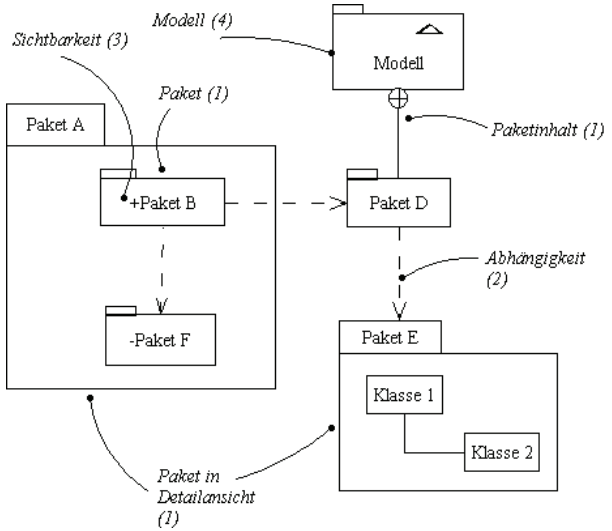




Komponenten Diagramme



Klassen-Diagramm



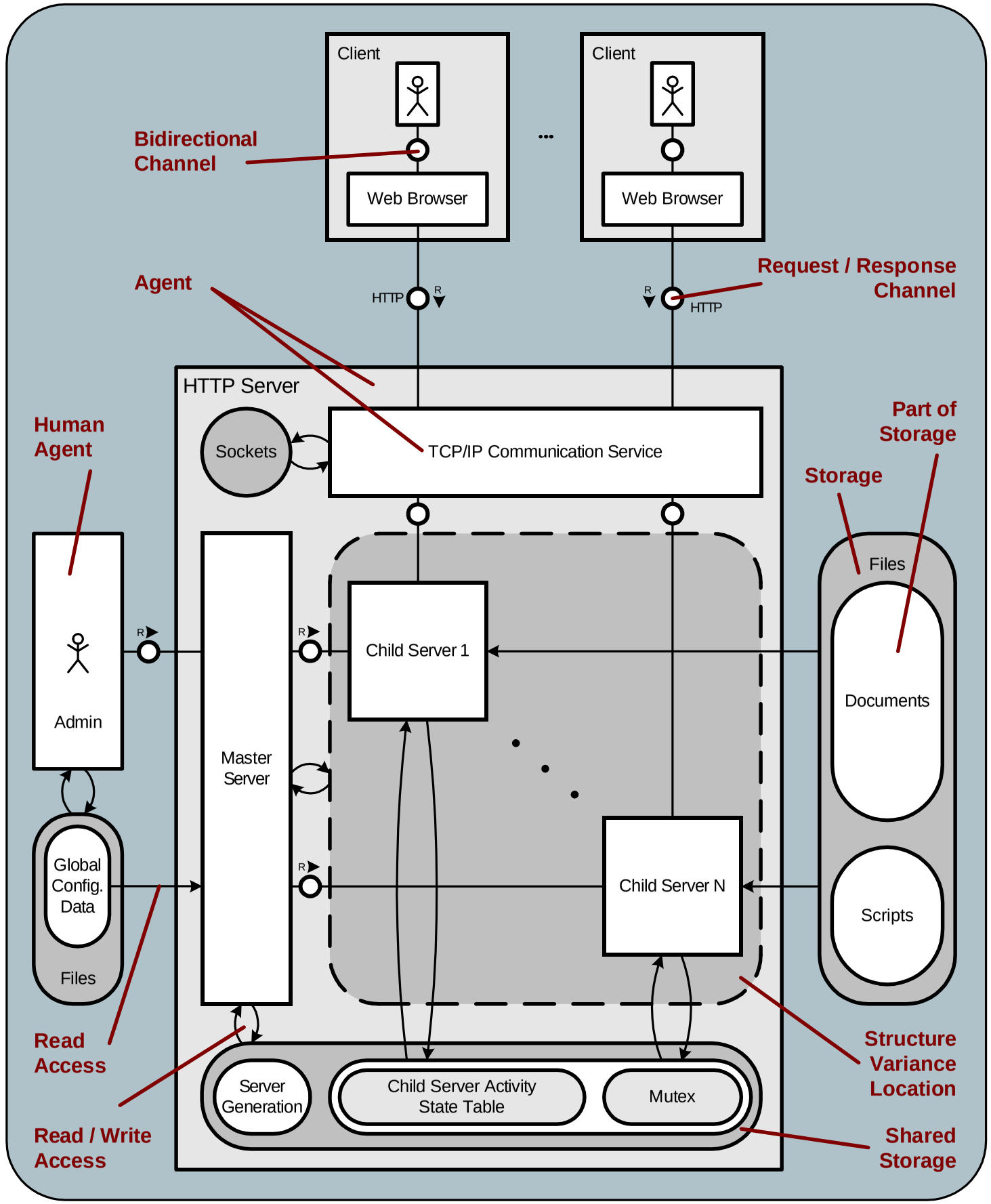
Package-Diagramm

UML Cheat Sheet

FMC

Compositional Structures

Block diagrams - Reference Sheet



FMC Block diagrams show the compositional structures as a composition of collaborating system components.

There are active system components called agents and passive system components called locations. Each agent processes information and thus serves a well-defined purpose. Therefore an agent stores information in storages and communicates via channels or shared storages with other agents. Channels and storages are (virtual) locations where information can be observed.

basic elements		
A	active system component : agent, human agent	Serves a well-defined purpose and therefore has access to adjacent passive system components and only those may be connected to it. A human agent is an active system component exactly like an agent but the only difference that it depicts a human. (Note 1: nouns should be used for identifier "A" Note 2: do not need to be depicted as rectangle or square but has to be angular)
S	passive system component (location) : storage, channel	A storage is used by agents to store data. (Note: do not need to be depicted as ellipse or circle but has to be rounded) A channel is used for communication purposes between at least two active system components. (Note: channels are usually depicted as smaller circles but may also vary like the graphical representation of storage places)
	access type	Directed and undirected edges represent the kind of access an active system component has to a passive system component. The types of access are read access, write access and a combination of both. (Note: usually undirected edges depicting read/write access are used on channels whereas two directed edges also depicting read/write access are used on storages)
common structures		
A S	read access	Agent A has read access to storage S.
A S	write access	Agent A has write access to storage S. In case of writing all information stored in S is overwritten.
A S	read / write access (modifying access)	Agent A has modifying access to storage S. That means that some particular information of S can be changed.
A1 A2	unidirectional communication channel	Information can only be passed from agent A1 to agent A2.
A1 A2	bidirectional communication channel	Information can be exchanged in both directions (from agent A1 to agent A2 and vice versa).
A1 RES REQ A2 A1 R A2	request / response communication channel (detailed and abbreviation)	Agent A1 can request information from agent A2 which in turn responds (e.g. function calls or http request/responses). Because it is very common, the lower figure shows an abbreviation of the request/response channel.
A1 S A2	shared storage	Agent A1 and agent A2 can communicate via the shared storage S much like bidirectional communication channels.
advanced		
A1 A2	structure variance	Structure variance deals with the creation and disappearance of system components. An agent (A1) changes the system structure (creation/deletion of A2) at a location depicted as dotted storage. System structure change is depicted as modifying access. After creation agent A1 can communicate with agent A2 or vice versa.

1. Selbsteinschätzung

1. Programmiererfahrung in Jahren: ____
2. Programmiererfahrung in Jahren in der Wirtschaft: ____

1.1 Modularisierungstechniken

	-2	-1	0	1	2	weiß nicht
Programmieren sie regelmäßig:	nie		manchmal		immer	weiß nicht
3. prozedural, also bspw. mit C oder Pascal?	☐	☐	☐	☐	☐	☐
4. funktional, also bspw. mit Lisp, Scheme, Haskell, Scala, etc?	☐	☐	☐	☐	☐	☐
5. objektorientiert, also bspw. mit C++, Java, Smalltalk, C#, etc?	☐	☐	☐	☐	☐	☐

Welche Techniken zur *Multidimensional Separation of Concerns* kennen und verwenden sie?	unbekannt		bekannt		oft angewandt	weiß nicht
6. Subject-oriented Programming	☐	☐	☐	☐	☐	☐
7. Aspect-oriented Programming	☐	☐	☐	☐	☐	☐
8. Context-oriented Programming	☐	☐	☐	☐	☐	☐
9. Feature-oriented Programming	☐	☐	☐	☐	☐	☐

Welche Sprachmittel verwenden sie zur Modularisierung ihrer Programme?	unwichtig		normal		wichtigste	weiß nicht
10. Funktionen	☐	☐	☐	☐	☐	☐
11. Klassen	☐	☐	☐	☐	☐	☐
12. Interfaces	☐	☐	☐	☐	☐	☐
13. Packages	☐	☐	☐	☐	☐	☐
14. Mixins	☐	☐	☐	☐	☐	☐
15. Traits	☐	☐	☐	☐	☐	☐
16. Sonstige:						

1.2 C-Programmierung

	-2	-1	0	1	2	weiß nicht
	gar nicht		normal		sehr gut	weiß nicht
17. Wie gut verstehen sie C-Code?	☐	☐	☐	☐	☐	☐
Verwenden sie C?	nie		manchmal		immer	weiß nicht
18. Beheben sie ab und an Fehler oder machen kleine Änderungen an C-Programmen?	☐	☐	☐	☐	☐	☐
19. Entwickeln sie Anwendungsprogramme, die in C geschrieben sind?	☐	☐	☐	☐	☐	☐
20. Entwickeln sie Systemprogramme, die in C geschrieben sind?	☐	☐	☐	☐	☐	☐
21. Fühlen sie sich als C-Guru und schreiben sie zum Frühstück den GCC in C neu?	☐	☐	☐	☐	☐	☐

1.2 Kenntnisse in Aspekt-orientierter Programmierung	-2	-1	0	1	2	weiß nicht
22. Programmiererfahrung in Jahren:	___					
Verwenden sie eines dieser AOP-Werkzeuge?	nie		manchmal		immer	weiß nicht
23. AspectJ	☐	☐	☐	☐	☐	☐
24. CeasarJ	☐	☐	☐	☐	☐	☐
25. AspectWerkz	☐	☐	☐	☐	☐	☐
26. JBossAOP	☐	☐	☐	☐	☐	☐
27. AspectC++	☐	☐	☐	☐	☐	☐
28. andere: _____						
Welche AOP-Sprachmittel verwenden sie bzw. sind für sie wichtig?	unwichtig		normal		wichtigste	weiß nicht
29. Advice: around, before, after	☐	☐	☐	☐	☐	☐
30. cflow-Konstruktionen	☐	☐	☐	☐	☐	☐
31. Inter-Typ-Definitionen	☐	☐	☐	☐	☐	☐
32. andere: _____						

2. Ergebnisse der Aufgaben

Architektur	-2	-1	0	1	2	weiß nicht
1. Wie schwierig war es, die Architektur des Systems zu erkennen?	sehr schwierig ☐	☐	normal ☐	☐	sehr einfach ☐	☐
2. Wie detailliert ist die von ihnen dokumentierte Architektur?	sehr grob ☐	☐	☐	☐	sehr detailliert ☐	☐
3. Sollte sie weiter verfeinert werden, bevor sie von anderen Entwicklern als Entwicklungsgrundlage verwendet werden kann?	nein ☐	☐	etwas ☐	☐	sehr stark ☐	☐
4. Wie viel Zeit haben sie in etwa benötigt, um die Architektur zu erkennen?	----------					
5. Wie viel Zeit haben sie in etwa benötigt, um die Architektur zu dokumentieren?	----------					
Implementierungsstrategie	gar nicht		etwas		sehr stark	weiß nicht
6. Beeinflusst das implementierte Feature die von ihnen erkannte Architektur?	☐	☐	☐	☐	☐	☐
7. Haben sich vorhandene Modulbeziehungen geändert?	☐	☐	☐	☐	☐	☐
8. Ließ sich die Strategie so gestallten, dass nur neue Module hinzukommen ohne bestehende zu ändern?	☐	☐	☐	☐	☐	☐
9. Ließ sich die Strategie so gestallten, dass nur Beziehungen mit neuen Modulen hinzugefügt werden mussten, ohne Beziehungen zwischen bestehenden Modulen verändern zu müssen?	☐	☐	☐	☐	☐	☐

10. Welche Gesichtspunkte des Problems ließen sich nicht modularisieren?

11. Welche Gesichtspunkte der ursprünglichen Architektur wurden verändert?

12. Wie viel Zeit haben sie etwa benötigt, um eine Implementierungsstrategie zu entwickeln?

13. Wie viel Zeit haben sie etwa benötigt, um die Strategie zu dokumentieren?

Implementierungsstrategie bezogen auf die Referenzarchitektur	-2	-1	0	1	2	weiß nicht
14. Beeinflusst das implementierte Feature die Referenzarchitektur?	gar nicht ☐	☐	etwas ☐	☐	sehr stark ☐	☐

15. Welche Änderungen haben sie im Vergleich zur Referenzarchitektur vorgenommen?

16. Waren sie in der Lage, alle Abhängigkeiten zwischen den Modulen zu erkennen?	unvollständig ☐	☐	die relevanten ☐	☐	alle ☐	weiß nicht ☐

3. CSOM und Architektur

	-2	-1	0	1	2	weiß nicht
1. Hatten sie bereits vor diesem Experiment Kontakt mit dem CSOM-Quelltext?	nein ☐				ja ☐	☐
2. Wie schwierig war es für sie, sich im CSOM-Quelltext zurecht zu finden?	sehr schwierig ☐	☐	normal ☐	☐	sehr einfach ☐	☐
3. Wie sehr half dabei die Erinnerung an die VM-Vorlesung?	gar nicht ☐	☐	etwas ☐	☐	sehr stark ☐	☐
4. Wie sehr half dabei, dass sie sich bereits früher mit dem Quelltext beschäftigt haben?	gar nicht ☐	☐	etwas ☐	☐	sehr stark ☐	☐
5. Wie sehr unterscheidet sich die von ihnen wahrgenommene Architektur von der Strukturierung des Quellcodes?	gar nicht ☐	☐	etwas ☐	☐	sehr stark ☐	☐

6. Welche Unterschiede sehen sie bei CSOM zwischen Compile-Time und Laufzeit-Architektur, also der Architektur, die sich durch die Strukturierung des Quelltextes ergibt und der Architektur, die die Laufzeit-Komponenten bilden?

4. Sprachmittel (Non-VMADL)

1. Sind die Sprachmittel, die ihnen mit C zur Verfügung stehen, ausreichend für eine geeignete Modularisierung?	gar nicht ☐	☐	etwas ☐	☐	sehr geeignet ☐	☐

2. Welche Sprachmittel können sie sich für eine bessere Modularisierung vorstellen?

3. Welche konkreten Sprachmittel haben sie vermisst, die sie eventuell sonst verwenden?

4. Sprachmittel (VMADL)

	-2	-1	0	1	2	weiß nicht
1. Sind die vorgestellten und in CSOM verwendeten Sprachmittel ausreichend, um Modularisierung zu erreichen?	gar nicht ☐	☐	etwas ☐	☐	sehr geeignet ☐	☐

2. Gibt es Sprachmittel, die sie für die Lösung der Aufgabe vermisst haben? Wenn ja, welche?

3. Beeinflusst die Verwendung von AOP die Verständlichkeit?	Verschlechterung ☐	☐	nein ☐	☐	Verbesserung ☐	☐
4. Beeinflusst die Verwendung von AOP die Komplexität?	höhere Komplexität ☐	☐	nein ☐	☐	geringere Komplexität ☐	☐
5. Beeinflussen Intertyp-Definitionen die Verständlichkeit?	Verschlechterung ☐	☐	nein ☐	☐	Verbesserung ☐	☐

5. Durchführung und Fragebogen

Allgemeines	-2	-1	0	1	2	weiß nicht
1. War die Aufgabenstellung im angesetzten Zeitrahmen erfüllbar?	zu wenig Zeit ☐	☐	passend ☐	☐	zu viel Zeit ☐	☐
2. Hat der Durchführende sie ausreichend betreut und vorbereitet?	gar nicht ☐	☐	normal ☐	☐	sehr gut ☐	☐
3. War die Aufgabenstellung klar verständlich?	gar nicht ☐	☐	normal ☐	☐	sehr gut ☐	☐
4. Hat ihnen Vorwissen zur Erledigung der Aufgabe gefehlt?	gar nicht ☐	☐	etwas ☐	☐	sehr viel ☐	☐
5. War das zur Verfügung gestellte Material ausreichend?	zu wenig ☐	☐	passend ☐	☐	zu viel ☐	☐
6. Hat sie die Notation bei der Dokumentation der Architektur behindert?	gar nicht ☐	☐	etwas ☐	☐	sehr stark ☐	☐

7. Wenn ja, was würden sie anderes machen oder was hätten sie zusätzlich gebraucht?

8. War ihnen die verwendete Entwicklungsumgebung bereits bekannt?	gar nicht ☐	☐	normal ☐	☐	sehr gut ☐	☐
9. Wie haben sie sich mit der Entwicklungsumgebung durch den Quelltext bewegen können?	schlecht ☐	☐	normal ☐	☐	sehr gut ☐	☐
10. Hat sie die Entwicklungsumgebung beim Lesen des Quelltexts behindert?	nein ☐	☐	etwas ☐	☐	sehr stark ☐	☐

Potenzielle Fehler im Experimentdesign	-2	-1	0	1	2	weiß nicht
11. Haben sie sich in einer Wettbewerbssituation mit den anderen Teilnehmern befunden?	nein ☐	☐	etwas ☐	☐	sehr stark ☐	☐

12. War die Aufteilung in zwei Personenteams geeignet für die Erledigung der Aufgabe?	hinderlich	☐	☐	☐	kein Einfluss	☐	☐	☐	förderlich	☐	☐
13. Wie wären sie allein mit Erledigung der Aufgabe in dem Zeitrahmen zurecht gekommen?	gar nicht	☐	☐	☐	genauso	☐	☐	☐	viel besser	☐	☐

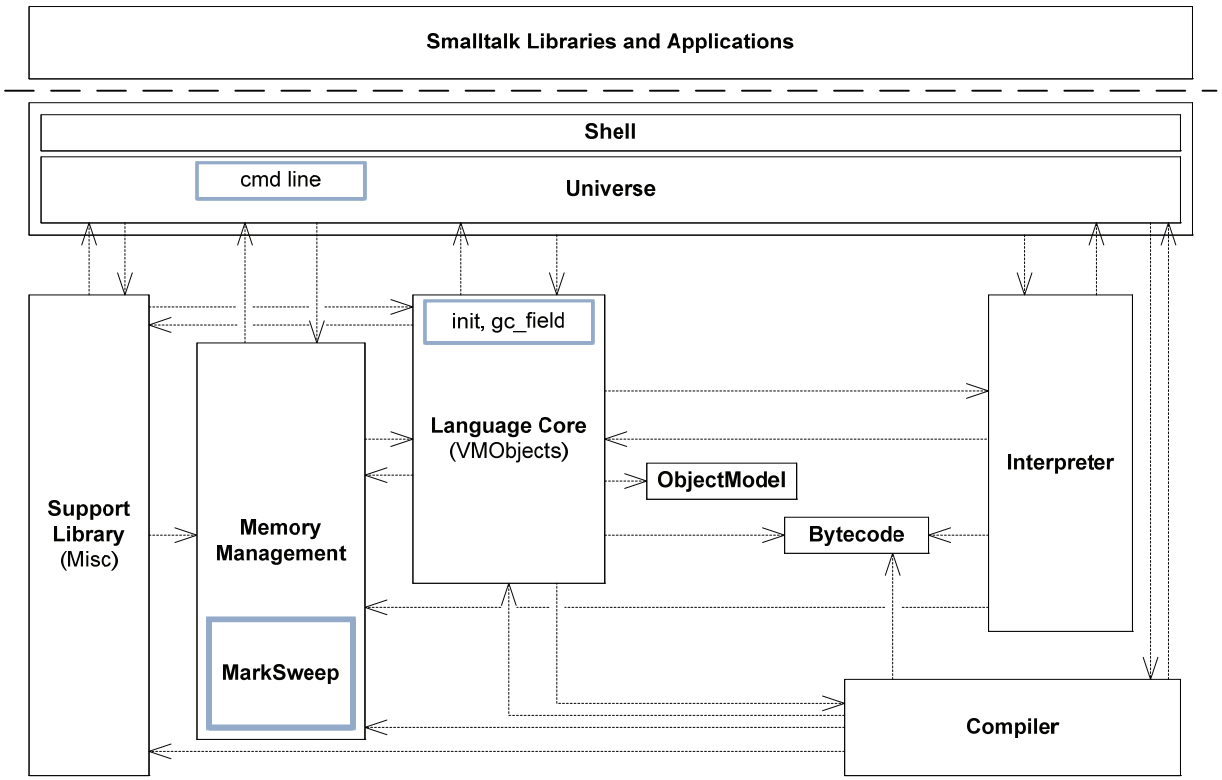
14. Gab es Probleme in ihrem Team gab, die sich negativ auf ihre Ergebnisse auswirken?

15. Aus welchem Grund haben sie am Experiment teilgenommen?

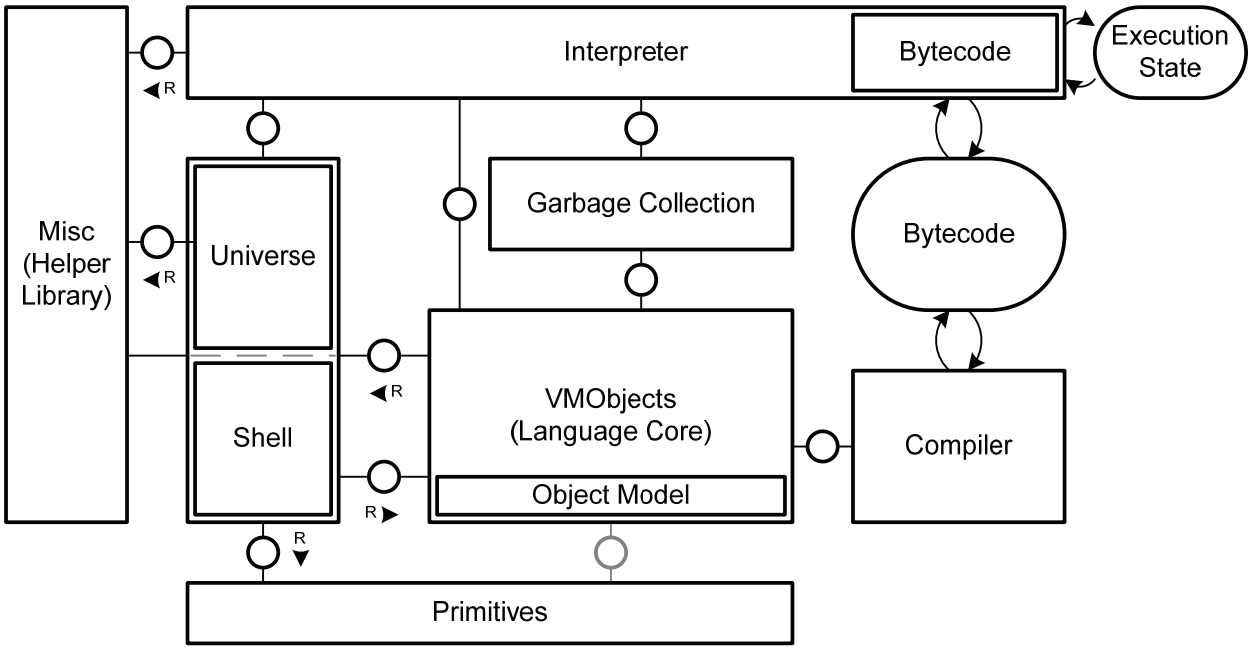
Fragebogen

16. Gibt es aus ihrer Sicht Fehler im Fragebogen?

CSOM Referenzarchitektur



UML-Klassendiagramm



FMC-Klassendiagramm