UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

DISSERTATION

Automatische Kompatibilitätsprüfung Framework-basierter Anwendungen

vorgelegt von

Dipl. Inform. Fabian Christ

als Teilleistung zur Erlangung
des akademischen Grades

Doktor der Naturwissenschaften
Dr. rer. nat.

der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Paderborn, November 2012

Fabian Christ
Automatische Kompatibilitätsprüfung Framework-basierter Anwendungen
Dissertation, Universität Paderborn, 2012

Gutachter:
Prof. Dr. Gregor Engels
Prof. Dr. Wilhelm Schäfer

Tag der Verteidigung: 20.12.2012

Zusammenfassung

Software-Architekturen betrieblicher Informationssysteme bestehen aus Architekturbausteinen wie Frameworks, Komponenten und Bibliotheken. Die Entwicklung dieser Architekturen unter Einbeziehung wiederverwendbarer Architekturbausteine ist eine Voraussetzung einer industriell organisierten Software-Entwicklung. Für eine effiziente Trennung der Zuständigkeiten werden dabei häufig Architekturbausteine von Drittanbietern wiederverwendet.

Frameworks als erweiterbare Architekturbausteine bieten besondere Vorteile. Sie erlauben sowohl die Wiederverwendung der Funktionalität als auch der durch das Framework vorgegebenen Software-Architektur. Beispiele sind Frameworks für Benutzungsoberflächen oder für die Anbindung von Datenbanken. Durch Implementierung anwendungsspezifischer Erweiterungen wird ein Framework für den konkreten Anwendungsfall angepasst. Eine Anwendung, deren Software-Architektur ein Framework einsetzt, benutzt das Framework über dessen Erweiterungspunkte.

Im Laufe der Evolution einer solchen Anwendung entsteht häufig die Situation, dass das Framework durch eine neuere Version aktualisiert werden soll. Die Aktualisierung enthält das Risiko, dass Inkompatibilitäten zwischen bestehender Anwendung und neuer Framework-Version auftreten, die wiederum zu aufwendigen Anpassungen führen. Daher müssen mögliche Inkompatibilitäten vor der Aktualisierung erkannt und bewertet werden. Nach aktuellem Stand der Technik ist dies nicht möglich, so dass es in der industriellen Praxis zu unvorhergesehenen Problemen verbunden mit hohen Kosten kommt.

Wir stellen ein Verfahren zur automatischen Kompatibilitätsanalyse Framework-basierter Anwendungen vor, mit dem das beschriebene Problem gelöst wird. Durch eine Kombination aus Codeanalyse und neuartiger Framework-Beschreibung lassen sich mögliche Inkompatibilitäten vor Durchführung der Aktualisierung automatisch berechnen. Eine prototypische Implementierung des Verfahrens im Werkzeug »Companian« demonstriert die praktische Einsetzbarkeit unseres Verfahrens.

Für die Definition der benötigten Framework-Beschreibungssprache mittels Meta-Modellierung setzen wir die neuartige Modellierungstechnik der parametrisierten Meta-Modelle ein. Die Technik unterstützt unseren Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen. Mit diesem Ansatz können bestehende Sprachen für Teilaspekte neu definierter Sprachen wiederverwendet werden. Unser pragmatischer Ansatz stellt dabei sicher, dass hierbei nur die Sprachen benutzt werden können, die syntaktisch und semantisch den geforderten Sprachen entsprechen.

Abstract

The software architectures of enterprise information systems consist of architectural building blocks like frameworks, components, and libraries. The development of such architectures considering reusable architectural building blocks is a prerequisite for the industrial practice of software development. For an efficient separation of concerns third-party building blocks are reused.

Frameworks as extensible building blocks have several advantages. They admit to reuse the framework's functionality and its architecture. Examples are user interface or database connectivity frameworks. A framework is customized by implementing application specific extensions. An application uses a framework through its extension points.

During the evolution of such an application the situation arises where the framework has to be updated to a newer version. An update contains the risk of incompatibilities between the existing application and the new framework version. This may result in high efforts for required adjustments. Thus, possible incompatibilities have to be recognized and evaluated before an update is performed. This is not possible in accordance with the current state-of-the-art and results in unforeseen problems accompanied by high consequential costs.

We propose a process for an automatic compatibility analysis of framework-based applications that solves the problem. Combining code analysis and a novel framework description our process detects possible incompatibilities before an update is performed. A prototypical implementation in the »Companian« tool demonstrates the practical feasibility of our process.

For the definition of the required framework description language via metamodelling we make use of our novel modelling technique called parametrized metamodels. This technique supports our approach of a requirement-based reuse of languages. With this approach, existing languages can be reused as sub-languages for certain aspects of a newly defined language. We specify language requirements for those aspects where we would like to reuse an existing language. Our practical approach ensures that languages can be reused only if they match a required language syntactically and semantically.

Dank

Die Arbeit als Forscher und Verfasser einer Dissertation ist nicht möglich ohne die Unterstützung durch Familie, Freunde und Kollegen. Ich möchte an dieser Stelle den Menschen danken, die mich durch diese Zeit mit Rat und Tat begleitet haben.

Der größte Dank geht an Irina und Jakob – meiner kleinen Familie, die mir so viel Kraft und Rückhalt gibt, dass man es schwerlich in Worte fassen kann.

Weiterer Dank geht an meine Eltern, die mir all das ermöglicht haben; meinem Papa für wertvolle Korrekturen, meiner Mama für das gute Essen und meiner Schwester für schlaue Kommentare. Ich danke meinen Onkels und Tanten sowie meiner erweiterten Familie um Irinas Eltern und Schwestern samt Männern für ihr immerwährendes Interesse an meiner Arbeit.

Gute Freunde und Kollegen, mit denen man sich jederzeit fachlich als auch freundschaftlich auseinandersetzen konnte, waren für mich extrem wichtig. Ohne besondere Reihenfolge danke ich der „Kleenen“ Annette, der Korrekturleserin Petra, dem Bio-Doktor-Anwärter Christian H. und natürlich den Mitbewohnern in der ZM1 mit Benjamin, Johnny, Henning, Markus, Christian S., Christian G., Stefan S. und all den anderen Mitstreitern der FG-Engels und des s-lab.

Ich danke den Professoren des s-lab, namentlich Prof. Kastens, Prof. Kleine Büning, Prof. Rammig und Prof. Schäfer, für ihre wertvollen Anmerkungen während der jährlichen s-lab Research Days.

Schließlich gilt mein Dank meinem Doktorvater Prof. Dr. Gregor Engels, der mich inhaltlich mit vielen Denkanstößen und Ratschlägen durch die Promotion begleitet hat. Ich danke Gregor für eine Denkschule, in der er mir die Welt der wissenschaftlichen Herangehensweise an Probleme mit für mich neuen Denkweisen eröffnet hat.

Paderborn, 15. November 2012

Inhaltsverzeichnis

1

Einführung in die Evolution von Anwendungen und Frameworks

„We believe the information system space will be dominated by enterprise frameworks well into the next century.“

-- David S. Hamu and Mohamed E. Fayad in [Hamu1998]

Das einleitende Zitat von David S. Hamu und Mohamed E. Fayad aus dem Jahr 1998 beruht auf den damaligen wissenschaftlichen Erkenntnissen im Bereich der komponentenorientierten Softwareentwicklung mit Hilfe von Frameworks. Gut zehn Jahre später lautet das Resümee, dass die Autoren Recht behielten und Frameworks eine dominierende Rolle spielen.

Frameworks, als wiederverwendbare und erweiterbare Einheiten einer Software-Architektur, haben sich im Bereich der Entwicklung von Informationssystemen etabliert. Die Kombination aus Objektorientierung und komponentenorientierter Entwicklung, die durch Frameworks realisiert wird,

ermöglicht ein Höchstmaß an Wiederverwendung von Software. Ein Framework wird eingesetzt, um einen bestimmten Teil einer Softwarearchitektur und deren Funktionalität zu realisieren. Durch den Einsatz eines Frameworks kann so der Eigenentwicklungsanteil reduziert werden. Neben der Funktionalität des Frameworks wird implizit auch die Architektur des Frameworks in die eigene Architektur integriert, so dass nicht nur die Funktionalität, sondern auch die Architektur des Frameworks in der Anwendung wiederverwendet wird. Dies kann soweit führen, dass die Architektur des Frameworks die Gesamtarchitektur der Anwendung bestimmt, was in vielen Fällen auch gewollt ist, da man davon ausgeht, dass die etablierte Architektur des Frameworks eine gute Vorgabe für die Architektur der Anwendung darstellt. So basiert die zu erstellende Anwendung im Sinne von Funktionalität und im Sinne von Architektur auf dem eingesetzten Framework.

Neben seiner wiederverwendbaren Funktionalität und Architektur ist ein Framework durch seinen Zweck als anpassbares Gerüst einer Anwendung charakterisiert. Die Anpassung erfolgt dabei an so genannten Erweiterungspunkten, für die entsprechende Erweiterungen entwickelt werden können. Ein schematisches Bild einer Anwendung, die ein Framework benutzt, zeigt Abbildung 1.1.

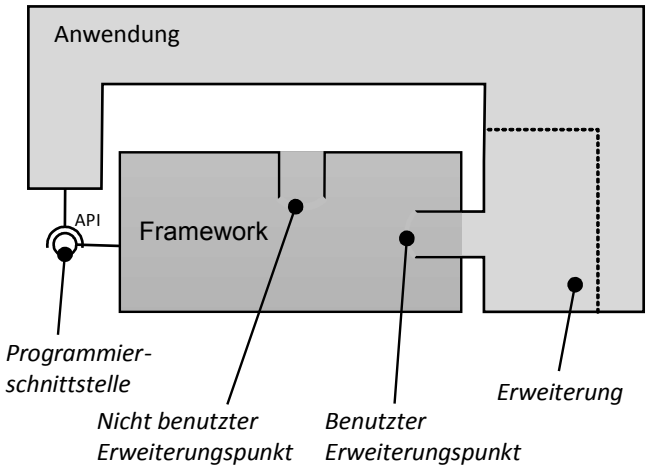


Abbildung 1.1: Anwendung benutzt ein Framework

Ein Framework besitzt eine Programmierschnittstelle (API), über die es in den Ablauf der Anwendung integriert werden kann. Über diese Schnittstelle lässt sich das Framework typischerweise auch konfigurieren. Eine solche Schnittstelle findet man in der Regel bei allen Arten von wiederverwendbaren Software-Komponenten und -Bibliotheken. Ein zentraler Unterschied bei der Wiederverwendung von Frameworks ist ihre Wiederverwendung durch Implementierung anwendungsspezifischer Erweiterungen. Dieser Mechanismus ist, wie wir sehen werden, grundlegend verschieden von den Wiederverwendungsmöglichkeiten anderer Konzepte. Über angebotene Erweiterungspunkte kann das Framework gezielt angepasst und seine Funktionalität erweitert werden. Entscheidender Unterschied ist hier, dass Erweiterungen vom Framework aufgerufen werden, so dass der Kontrollfluss vom Framework und nicht von der

Anwendung gesteuert wird. Daher nennt man dieses Prinzip in der Softwaretechnik »Inversion of Control«.

Häufig wird innerhalb einer Anwendung mehr als ein Framework eingesetzt. Ist dies der Fall, muss sichergestellt werden, dass die zu erstellende Anwendung die Frameworks korrekt benutzt und dass die Frameworks untereinander kompatibel sind. Die Kompatibilität bezieht sich sowohl auf die Funktionalität, als auch auf die Architektur. In diesem Fall betrachten wir die Kompatibilität von Anwendungen zu Frameworks und der eingesetzten Frameworks untereinander. Der Begriff der Kompatibilität nimmt in diesem Zusammenspiel eine zentrale Rolle ein, die wir in dieser Arbeit noch genauer betrachten werden.

Bleiben wir jedoch bei der grundlegenden Situation, dass eine Anwendung ein Framework benutzt und stellen uns eine typische Situation in einem Software-Entwicklungsprojekt vor. Der Projektleiter eines solchen Projektes weiß, dass ein bestimmtes Framework in der zu entwickelnden Anwendung benutzt wird. Nun wird eine neue Version des Frameworks veröffentlicht, in der unter anderem einige Fehler behoben wurden. Um diese verbesserte Framework-Version zu benutzen, muss das Framework innerhalb der Anwendung aktualisiert werden. Der Projektleiter steht vor der Aufgabe zu entscheiden, ob und wann man auf die neue Framework-Version aktualisiert. Aus Erfahrung weiß der Projektleiter, dass bei einer solchen Aktualisierung Inkompatibilitäten entstehen können, so dass unter Umständen umfangreiche Anpassungen an die neue Framework-Version notwendig werden.

Idealerweise könnte der Projektleiter die Aktualisierung simulieren und würde zeitnah und gesichert herausfinden, ob eine Umstellung problemlos durchzuführen wäre. Etwaige Probleme würden ihm angezeigt, so dass er auf Basis dieser Daten entstehende Entwicklungsaufwände abschätzen und eine fundierte Entscheidung fällen kann. Leider stehen Projektleiter vor dem Problem, dass nach aktuellem Stand der Forschung zu Frameworks eine solche Simulation nicht möglich ist. Projektleiter wissen somit nicht, welche Anpassungen an der Anwendung notwendig wären, würde man das Framework aktualisieren. Es könnte zu unvorhergesehenen Inkompatibilitäten kommen, welche mit viel Aufwand korrigiert werden müssen, was wiederum enorme Auswirkungen auf Zeit- und Projektpläne haben kann.

Die Entscheidung für oder gegen eine Aktualisierung ist in der Praxis ein Vorgehen von Versuch und Irrtum, in das die Erfahrungen und Einschätzungen von Entwicklern einfließen. Hierzu werden häufig, je nach Umfang des Softwaresystems, ein oder mehrere Entwickler beauftragt, die Aktualisierung probehalber zu versuchen, um dann zu beobachten, was nicht funktioniert hat. Wir nennen dies eine Migration auf Probe. Dieses manuelle Vorgehen hat eine Reihe von Nachteilen [Christ2011].

- Es ist ein manueller und damit zeit- sowie kostenaufwändiger Prozess.
- Entwickler werden gebunden, die ansonsten für andere (wichtigere) Entwicklungen eingesetzt werden könnten.

- Der Aufwand einer Migration auf Probe ist im Worst-Case genauso hoch wie der einer tatsächlichen Migration. Es entsteht Aufwand, den man eigentlich erst investieren möchte, nachdem man sich endgültig für eine Migration entschieden hat.

Projektleiter möchten die Frage der Kompatibilität beantworten können, ohne dass Entwickler eine Migration auf Probe durchführen müssen, die unnötig Aufwand und somit Geld kostet.

In dieser Dissertationsschrift beschäftigen wir uns mit der skizzierten Problemstellung und stellen Sprachen, Methoden und technische Verfahren zur Verfügung, die es ermöglichen, die Auswirkungen einer Framework-Aktualisierung in Bezug auf mögliche Inkompatibilitäten automatisch zu bestimmen, ohne dass eine Migration auf Probe notwendig ist.

Wir wollen die gegebene Situation konkretisieren und fokussieren auf den Bereich der betrieblichen Informationssysteme und dem dortigen Einsatz und der Aktualisierung von Frameworks. Die Softwareentwicklung betrieblicher Informationssysteme basiert in hohem Maße auf dem Einsatz von Frameworks. Die eingesetzten Frameworks bilden eine Technologieplattform, auf deren Basis verschiedene Softwaresysteme entwickelt werden können. Von der gemeinsamen Basis erhofft man sich ein hohes Maß an Kompatibilität zwischen verschiedenen Systemen zu garantieren und die Entwicklungskosten durch das Softwarewiederverwendungspotenzial niedrig zu halten. Derartige Technologieplattformen werden häufig auch als „Middleware“ bezeichnet.

Beispiele für Middleware-Lösungen sind die Java Enterprise Edition (JEE) der Firma Oracle (ehemals Sun Microsystems) oder Microsofts .NET Plattform. Speziell die seit 1999 verfügbare JEE hat sich mit der betriebssystemunabhängigen Programmiersprache Java zu einem Marktführer entwickelt. Durch offene Standardisierungsprozesse und die aktive Einbeziehung einer aktiven Open-Source-Bewegung entstand eine große Auswahl an Java-basierten Lösungen, die sich in eine JEE-Landschaft integrieren. Die große Open-Source-Gemeinschaft sorgt für einen stetigen Fluss an Innovationen, die Einfluss auf die zukünftigen JEE-Entwicklungen nehmen.

Die hervorstechendste Eigenschaft einer Middleware ist deren Anpassbarkeit, die durch das Framework-Konzept realisiert wird. Frameworks bilden den Stand der Technik, wenn es um Anpassungs- und Erweiterungsmöglichkeiten gepaart mit einem möglichst hohen Maß an Wiederverwendung geht. In einer Middleware werden in der Regel eine Vielzahl von Frameworks für unterschiedliche Aufgabenbereiche in der Architektur kombiniert, um den größtmöglichen Wiederverwendungseffekt zu erzielen.

Die konkreten Ausprägungen von Frameworks sind vielfältig. Unter anderem gibt es Frameworks für die Darstellung von Benutzungsoberflächen, die Verbindung zu Datenbanken oder die Verschlüsselung von Daten. Die Beispiele zeigen bereits, dass jedes Framework für einen bestimmten Zweck entwickelt wird. Für diesen Zweck stellt das Framework vorgefertigte Features und Erweiterungsmöglichkeiten bereit. So stellt ein Framework für Benutzungsoberflächen Elemente wie Fenster, Menüs, Knöpfe, Eingabefelder etc. zur Verfügung

und bietet die Möglichkeit durch Erweiterungen die Interaktion mit dem Benutzer zu steuern.

Die Entwicklung, aber auch das Erlernen eines Frameworks ist nach [Bosch1997][Fayad1999][Thomas2006] eine zeit- und damit kostenaufwändige Tätigkeit. Experten für spezifische Frameworks sind daher auf dem Arbeitsmarkt sehr gefragt, um Frameworks zielgerichtet in eine neue Software zu integrieren. Die Integration mehrerer Frameworks, wie es in einer Middleware der Fall ist, wirft darüber hinaus weitere Fragen sowie Probleme auf [Mattsson1997].

Kern vieler Probleme ist die Frage nach der Kompatibilität von Frameworks. Die Frage entsteht sowohl beim Einsatz unterschiedlicher Frameworks als auch bei der Planung von Framework-Aktualisierungen, wenn ein Framework durch eine neuere Version ersetzt werden soll. Wir skizzieren die Situation einer Framework-Aktualisierung und stellen in Abbildung 1.2 den Zusammenhang zwischen einer Anwendung und einem benutzten Framework dar. Wir erweitern die Informationen zu dem verwendeten Framework um eine Versionsnummer, um deutlich zu machen, dass jedes Framework nur in einer ganz bestimmten Version verwendet wird. Auch die Anwendung ist versioniert und wird von Version zu Version weiterentwickelt. Wir betrachten die Situation, in der das Framework von Version 1.1.5 auf Version 1.2.0 weiterentwickelt wird – es evolviert.

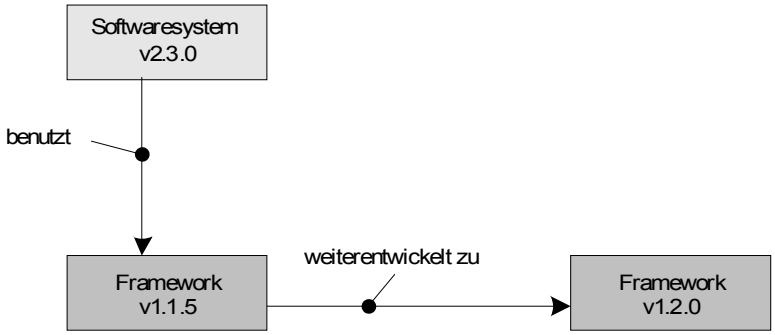


Abbildung 1.2: Ein Softwaresystem evolviert

Ausgehend von der Situation in Abbildung 1.2 stellt sich mit der Evolution des Frameworks das Problem, dass neue Funktionen, die in der Anwendung realisiert werden könnten, auch eine weiterentwickelte Version des Frameworks voraussetzen. Die Verwendung einer neuen Framework-Version hängt jedoch davon ab, ob eine Aktualisierung des Frameworks Auswirkungen auf das bestehende System haben wird. Es ist die Frage nach der Kompatibilität zwischen bestehender Anwendung in Version 2.3.0 zu einer neuen Framework-Version 1.2.0.

Ein Projektleiter würde zunächst sehr genau wissen wollen, welche Auswirkungen eine Aktualisierung nach sich ziehen würde. In der Praxis fehlen häufig verlässliche Informationen über die Abwärtskompatibilität neuer Framework-Version. Hinzu kommt, dass die Ursache für Inkompatibilitäten häufig nicht neue Funktionalitäten des Frameworks sind, sondern durch Refaktorisierungen entstehen [Dig2005]. Falls die neue Framework-Version abwärtskompatibel

wäre, würde es keine Probleme geben. Für den in der Praxis relevanten Fall, dass die neue Framework-Version nicht abwärtskompatibel ist, würde hingegen eine möglichst gute Einschätzung der Situation benötigt, um zu vermeiden, dass eine Aktualisierung so große Entwicklungsaufwände nach sich zieht, dass man die Aktualisierung hätte vermeiden wollen, hätte man die Situation besser einschätzen können.

Betrachten wir die Problemstellung in Abbildung 1.3 noch einmal systematisch. Eine Anwendung in Version 2.3.0 soll weiterentwickelt werden und es stellt sich die Frage, ob im Zuge dieser Arbeit auch die Version des verwendeten Frameworks von 1.1.5 auf 1.2.0 aktualisiert wird. Die Entscheidung für oder gegen eine Migration von einer Framework-Version zur anderen ist abhängig von der Kompatibilität des existierenden Softwaresystems in Version 2.3.0 zur neuen Framework-Version. Mögliche Auswirkungen durch Inkompatibilitäten müssen in die Planung der Weiterentwicklung einfließen.

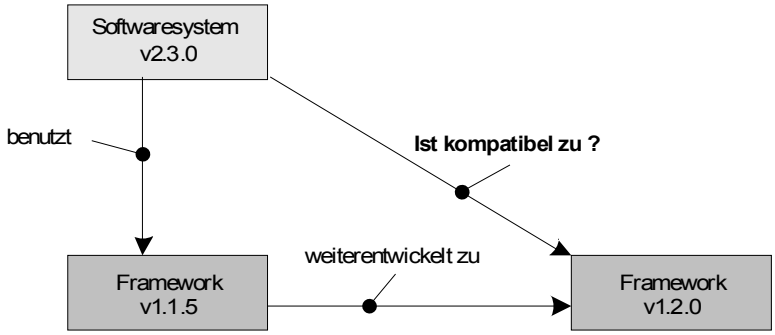


Abbildung 1.3: Problemstruktur

Nach aktuellem Stand der Technik und der Praxis haben Projektleiter und Entwickler wenig Informationen zur Hand, die eine Aussage über die zu erwartenden Auswirkungen zulassen würden. Veränderungen werden typischerweise in Form von »Release Notes« dokumentiert, die auf einer abstrakten Ebene auflisten, was die Entwickler von der einen zur nächsten Version weiterentwickelt haben. Mit abstrakter Ebene ist gemeint, dass man Sätze vorfindet wie: „Feature X implementiert.“, „Fehler ABC behoben.“ oder „Neuer Konfigurationsparameter P eingeführt.“ Problem ist, dass weder standardisiert ist, was man dort wie aufschreibt, noch dass man aus diesen Angaben schließen könnte, welche Auswirkungen für die Kompatibilität der neuen Version zu einer gegebenen Anwendung eine Veränderung hat. Außerdem werden die Veränderungen in dieser Form nur von einer Version auf die nächste erhoben. Es ist kaum möglich Veränderungen über mehrere Versionen hinweg abzuleiten.

Verwandte Arbeiten wie [Mccamant2004] helfen bedingt in dieser konkreten Problemstellung, da die speziellen Gegebenheiten von Frameworks nicht betrachtet werden, sondern Komponenten und Bibliotheken mit Schnittstellen im Allgemeinen. Nichtsdestotrotz helfen diese Vorarbeiten, um dieses spezielle Problem zu lösen.

Das Problem, die Auswirkungen einer Framework-Aktualisierung nicht beurteilen zu können, wird in industriellen Softwareprojekten zu einem Risiko. Der geflügelte Ausspruch: »Never touch a running system«, rührt daher, dass Pro-

jektleiter häufig zu wenig Informationen darüber haben, was geschehen würde, wenn man eine Änderung, wie etwa eine Framework-Aktualisierung, durchführen würde. Um sich nicht der Gefahr auszusetzen, die Anwendung mit unentdeckten Fehlern zu infizieren und instabil werden zu lassen, werden derartige Änderungen häufig vermieden. Sie werden nur dann durchgeführt, wenn es scheinbar keine Alternativen zu dieser Änderung gibt. Wüsste man um die Auswirkungen einer Aktualisierung ohne es ausprobieren zu müssen, könnte man wesentlich zielsicherer und unabhängig von subjektiven Entwicklereinschätzungen abwägen, ob die Aktualisierung durchgeführt wird.

1.1 Aufgabenstellung

Aus den vorangegangenen Betrachtungen und der Problemstruktur in Abbildung 1.3 ergibt sich folgende Ausgangssituation: Gegeben sei eine Anwendung, die ein Framework in einer Version 1 benutzt. Das Framework soll nun durch eine neuere Version 2 aktualisiert werden. Die Aufgabe besteht darin, die folgende Fragestellung automatisiert beantworten zu können:

> *Ist die Anwendung, die bisher Version 1 des Frameworks benutzt hat, auch kompatibel zur Version 2?*

Da die Anwendung das Framework bereits in Version 1 benutzt hat, können wir davon ausgehen, dass die Anwendung kompatibel zur Version 1 des Frameworks ist. Hieraus ergibt sich die nachfolgende *Aufgabenstellung*.

> *Gegeben sei eine Anwendung A, ein Framework F_1 in Version 1 und eine weiterentwickelte Version 2 des Frameworks, bezeichnet als F_2. A sei kompatibel zu F_1.*
>
> *Entwickle eine automatische Kompatibilitätsprüfung zur Bestimmung möglicher Inkompatibilitäten zwischen A und F_2, wie sie bei Aktualisierung des Frameworks von F_1 nach F_2 auftreten würden.*
>
> **Aufgabenstellung**

Entscheidendes Kriterium ist die Kompatibilität zwischen Anwendung und Framework. Wir werden im Verlauf der Arbeit den Begriff der Kompatibilität noch präzisieren. Grundsätzlich bedeutet Kompatibilität in diesem Zusammenhang, dass die Anwendung kompatibel zum Framework im Bezug auf die Benutzung der Erweiterungspunkte ist. Eine Anwendung ist also kompatibel zu einem benutzten Framework, wenn die Erweiterungspunkte des Frameworks korrekt benutzt werden. Mögliche Inkompatibilität entsteht, wenn benutzte Erweiterungspunkte in der neuen Framework-Version verändert wurden. Hierbei ist zu beachten, dass nicht jede Veränderung notwendigerweise auch zu einer Inkompatibilität führt. Die automatische Kompatibilitätsprüfung muss in

der Lage sein, Veränderungen an den benutzten Erweiterungspunkten zu identifizieren und im Hinblick auf mögliche Inkompatibilitäten zu bewerten.

Ausgehend von der Aufgabenstellung können wir nun in Abschnitt 1.2 eine Lösungsstrategie skizzieren, von der wir glauben, dass sie das gestellte Problem löst.

1.2 Lösungsstrategie

Bei Betrachtung der Problemstruktur in Abbildung 1.3 und der industriellen Praxis, mit der dieses Problem angegangen wird, kommt man zu dem Schluss, dass ein Grund für eine Migration auf Probe der Mangel an Informationen ist.

Die erste Informationslücke entsteht, weil niemand weiß, welche Teile des Frameworks überhaupt von der Anwendung benutzt werden. Diese Information wird typischerweise nicht im Detail dokumentiert, sondern es wird lediglich festgehalten, dass die Anwendung das Framework benutzt – nicht in welchem Umfang. Die zweite Lücke ist ein mangelndes Wissen darüber, was sich in einem Framework von einer zur nächsten Version verändert hat.

Es ergibt sich die Konsequenz, dass man die Informationslücken schließen muss. Wir müssen mehr Informationen über die Benutzung des Frameworks durch die Anwendung und über die Unterschiede zwischen zwei beliebigen Framework-Versionen voraussetzen. Da man die Unterschiede zwischen zwei beliebigen Versionen ermitteln möchte, muss jede Framework-Version so dokumentiert sein, dass sich diese Unterschiede automatisch berechnen lassen. Schlüssel zur Bestimmung der Unterschiede zwischen zwei Framework-Versionen ist eine geeignete Beschreibung der jeweiligen Version. Unterschiede lassen sich dann aus der Differenz der Beschreibungen ableiten.

Mit diesen Überlegungen erweitern wir die Problemstruktur aus Abbildung 1.3 um die geforderten zusätzlichen Informationen und erhalten eine Struktur zur Lösungsstrategie in Abbildung 1.4. Dort fordern wir die Bestimmung detaillierterer Informationen über die Framework-Benutzung, aus denen wir ablesen können, welche Veränderungen am Framework überhaupt relevant sind. Denn wurden Teile des Frameworks verändert, die nicht benutzt werden, hat dies keine Auswirkungen auf die Kompatibilität. Darüber hinaus setzen wir das Vorhandensein von auswertbaren Beschreibungen der jeweiligen Framework-Version voraus, so dass wir die benötigten Informationen über die Unterschiede zwischen den Framework-Versionen bestimmen können. Dies sind die Grundlagen für die eigentliche Bestimmung möglicher Inkompatibilitäten zwischen Anwendung und neuer Framework-Version.

Wie in Abbildung 1.4 zu sehen, setzen wir in der Lösungsstrategie das Vorhandensein geeigneter Framework-Beschreibungen voraus. Alternativ wäre auch denkbar, auf diese Voraussetzung zu verzichten und die Unterschiede zwischen den Framework-Versionen durch den direkten Vergleich des Quellcodes beider Versionen zu ermitteln. Vorteil wäre hier, dass keine zusätzliche Beschreibung notwendig wäre. Darüber hinaus besteht bei einer zusätzlichen Beschreibung immer die Gefahr einer Abweichung zwischen Beschreibung und tatsächlicher Implementierung, die es zu verhindern gilt.

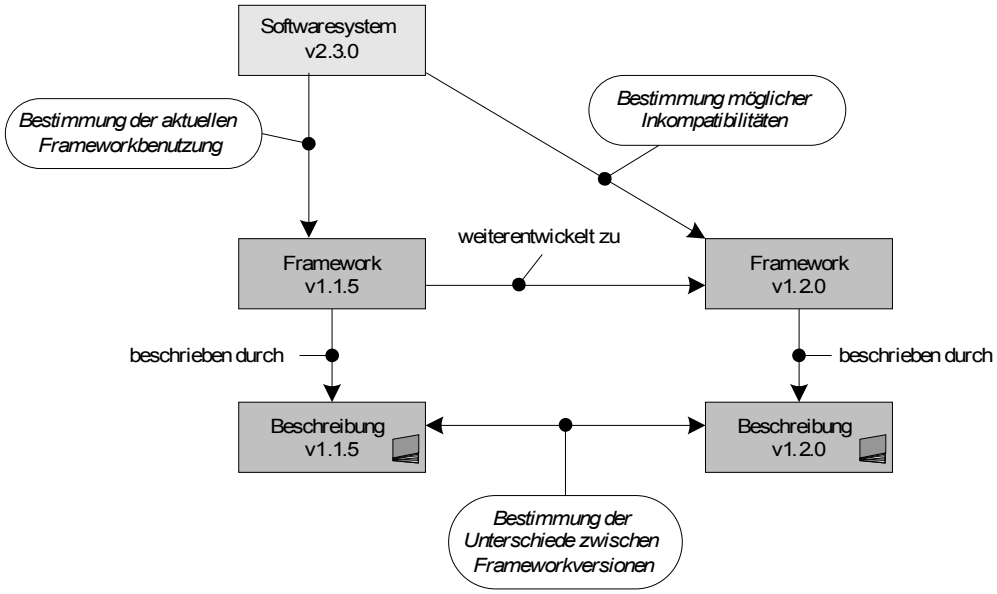


Abbildung 1.4: Lösungsstrategie einer automatischen Kompatibilitätsprüfung

Eine zusätzliche Beschreibung bietet jedoch den schwerer wiegenden Vorteil, dass durch eine Beschreibung auch eine Abstraktion von der eigentlichen Implementierung geschaffen wird. Wir verfolgen die These, dass diese zusätzliche Abstraktion ein wichtiger und unvermeidlicher Schritt im Vergleich zweier Framework-Versionen ist.

Ein Vergleich des Quellcodes beider Versionen würde Unterschiede auf Implementierungsebene bestimmen. In Abschnitt 2.5 gehen wir auf verwandte Arbeiten in diesem Bereich ein, die die Evolution eines Frameworks lediglich auf Ebene der API betrachten. Aus unserer Sicht ist diese Betrachtungsweise nicht ausreichend.

Auf API-Ebene würde man zum Beispiel erkennen, dass neue Klassen oder Methoden hinzugefügt, andere Klassen gelöscht und weitere verändert wurden. Ohne eine zusätzliche Abstraktion könnte man aus diesen Veränderungen nicht schließen, dass hierdurch zum Beispiel einer neuer Erweiterungspunkt implementiert wurde. Beim Vergleich der Kompatibilität zweier Framework-Versionen benötigen wir jedoch Vergleiche auf solch einer Abstraktionsebene, um verständliche Aussagen für einen Projektleiter zu gewinnen. Der Projektleiter ist an einer Aussage wie »Der Erweiterungspunkt „X“ wurde gelöscht.« sicherlich mehr interessiert als an einer Liste gelöschter Klassen.

Es gäbe nun die Möglichkeit zu versuchen, diese zusätzliche Abstraktion automatisch zu bestimmen, wie es verwandte Arbeiten aus dem Bereich des Reverse Engineerings [Chikofsky1990][Tonella2005][Tonella2005a] machen. Es zeigt sich jedoch, dass die automatische Bestimmung dieser zusätzlichen Abstraktion für einen Rechner im allgemeinen ein zu komplexes Problem darstellt, so dass die Genauigkeit der Abstraktion leidet [Canfora2007]. Mit der Einführung einer Beschreibung, die durch einen Menschen angefertigt wird, der in der Lage ist zu abstrahieren und Konzepte zu benennen, lösen wir dieses

Problem. Durch Vergleich der abstrakteren Beschreibungen können wir direkt Rückschlüsse auf Inkompatibilitäten auf dem Abstraktionsniveau der Beschreibungen ziehen und so bessere Ergebnisse erzielen als es bei einem Quellcodevergleich möglich wäre.

Weiteres Argument für die Verwendung einer Beschreibung ist die Tatsache, dass man hierfür die Beschreibung des Frameworks, wie sie für Framework-Benutzer vorgesehen ist, verwenden kann. Diese Form der Beschreibung ist ein Artefakt, das innerhalb der Framework-Entwicklung erstellt werden muss. Es stellt sich lediglich die Frage, wie diese Beschreibung sowohl inhaltlich als auch formal beschaffen sein muss, um für die Bestimmung der Unterschiede zwischen zwei Framework-Versionen verwendet werden zu können.

1.3 Struktur der Arbeit

In diesem Kapitel haben wir das Problem einer Framework-Aktualisierung betrachtet und die Aufgabenstellung abgeleitet. Als Hypothese für das weitere Vorgehen haben wir eine Lösungsstrategie entworfen, deren Validität wir im Verlauf dieser Arbeit prüfen und deren Umsetzung wir konkretisieren werden.

Im folgenden Kapitel 2 schaffen wir die Grundlagen, um die beschriebene Lösungsstrategie zu verfeinern. Hierzu definieren wir, was Frameworks eigentlich sind und betrachten die Kompatibilität einer Anwendung zu einem Framework. Wir betrachten bestehende Frameworks, analysieren Begriffsdefinitionen, zeigen welche Arten von Frameworks es gibt und wie sie sich gegen andere Konzepte wie Softwarekomponenten und -bibliotheken abgrenzen. Mit diesen Grundlagen schaffen wir das benötigte Verständnis von Frameworks und ihrer Kompatibilität und können uns zu verwandten Arbeiten abgrenzen.

In Kapitel 3 stellen wir auf Basis der geschaffenen Grundlagen den verfeinerten Lösungsansatz zur automatischen Kompatibilitätsprüfung vor. Hier vertiefen wir die Lösungsstrategie aus Abbildung 1.4 und zeigen entsprechende Konzepte zur Benutzungs-, Differenz- und Kompatibilitätsanalyse. Wir kommen zu dem Schluss, dass zur Umsetzung dieses Ansatzes eine ganzheitliche Form von Framework-Beschreibung benötigt wird.

In Kapitel 4 befassen wir uns daher mit existierenden Ansätzen für Framework-Beschreibungssprachen und stellen einen Vergleich im Hinblick auf die Anforderungen in dieser Arbeit an. Es zeigt sich, dass wir eine neue ganzheitliche Framework-Beschreibung benötigen, die es im Rahmen dieser Arbeit zu entwickeln gilt. Diese ganzheitliche Framework-Beschreibung erfordert einige Besonderheiten in der Modellierung, da hierfür existierende Sprachen als Teilsprachen wiederverwendet werden sollen. Leider fehlen uns geeignete Techniken im Bereich der Meta-Modellierung von Sprachen, die unseren Anforderungen genügen. Aus diesem Grund führen wir in Kapitel 5 die neue Technik der parametrisierten Meta-Modelle ein, die wir verwenden werden, um die geforderte ganzheitliche Framework-Beschreibungssprache in Kapitel 6 zu definieren.

In Kapitel 6 zeigen wir mit dem »Framework Description Meta-Model« (FDMM) eine neue Sprache für die Beschreibung von Frameworks. Das

FDMM ist ein parametrisiertes Meta-Modell, das über die Parametrisierung existierende Sprachen für bestimmte Teilaspekte wiederverwendet und so Frameworks ganzheitlich beschreiben kann.

Mit dem FDMM können wir in Kapitel 7 die automatische Kompatibilitätsprüfung umsetzen und zeigen, wie man eine Differenzanalyse zwischen zwei FDMM-basierten Framework-Beschreibungen durchführt. Auf Grundlage der Differenzanalyse und der Analyse der bisherigen Framework-Benutzung zeigen wir, wie man Rückschlüsse auf die Kompatibilität im Szenario der Framework-Aktualisierung ziehen kann. Wir stellen hierfür eine prototypische Werkzeugunterstützung vor und führen Fallstudien zur Evaluierung des Ansatzes durch.

In Kapitel 8 schließen wir die Arbeit mit einer Diskussion der erreichten Ergebnisse und gehen in einem Ausblick auf mögliche zukünftige Anwendungsgebiete und Forschungsfelder aufbauend auf dieser Arbeit ein.

2

Frameworks und ihre Kompatibilität

The architect's dream,
the developer's nightmare.

-- Martin Fowler

Nach der vorangegangenen Einführung in die Thematik dieser Arbeit wollen wir in diesem Kapitel den Begriff des »Frameworks« präzisieren und im Speziellen die Kompatibilität einer Anwendung zu einem Framework betrachten. Diese Grundlagen benötigen wir, um im anschließenden Kapitel 3 den Lösungsansatz für eine automatische Kompatibilitätsprüfung entwerfen zu können.

2.1 Frameworks und Wiederverwendung

Triebfeder für die Entwicklung von Frameworks in der Softwaretechnik ist der ewig bestehende Wunsch nach *Wiederverwendung* von Software. Man möchte ein Problem einmal durch Software lösen und dieses Stück Software immer wieder verwenden können, wenn ein vergleichbares Problem zu lösen ist. Dementsprechend bezeichnet man Software als *wiederverwendbar*, wenn

sie ohne oder mit wenigen Anpassungen in einem anderen Kontext wiederverwendet werden kann. Die zu beantwortende Forschungsfrage der Softwaretechniker lautet demnach: Wie entwickelt man wiederverwendbare Software?

Auf diese Frage gab es in der Geschichte der Softwaretechnik verschiedene Antworten. Geht man zurück an die Anfänge der Softwaretechnik, findet man in den ersten Programmiersprachen Lösungen für die Wiederverwendung durch die Benutzung von *Makros*. Ein Makro ist ein Block eines Programms, der einmal notiert wird und dann immer wieder ausgeführt werden kann. Anstatt einen Block immer wieder an die Stellen eines Programms zu kopieren, an dem er ausgeführt werden soll, kann man an diese Stellen einen Makro-Aufruf setzen, der zu dem einmal notierten Block verzweigt. Das Makro ist die einfachste Form der Wiederverwendung.

Die nächste Stufe der Wiederverwendung ist die Verwendung von parametrisierten Makros oder *Funktionen*. Eine Funktion ist ein Makro, dem Parameter übergeben werden können. Die Parameter sind die Eingabewerte der Funktion, mit denen sie ein Ergebnis berechnet. Das berechnete Ergebnis wird anschließend zurückgegeben. Dies entspricht der mathematischen Definition einer Funktion der Form y = f(x). Die Funktion f wird mit dem Parameter x aufgerufen und berechnet das Ergebnis y in Abhängigkeit von x. Eine solche Funktion kann mit unterschiedlichen Werten immer wieder aufgerufen werden, so dass dieser Programmteil wiederverwendbar ist.

Die nächste Lösung für das Problem der Wiederverwendung wurde zu Beginn der Neunzigerjahre mit der objektorientierten Programmierung entwickelt. Hierbei steht das Konzept des *Objektes* im Mittelpunkt der Softwareentwicklung. Ein Objekt ist in der Softwaretechnik das Konzept für alles Konkrete und Abstrakte, das in einer Software abgebildet werden soll. In einem Objekt werden sowohl dessen Verhalten als auch dessen Eigenschaften implementiert.

Jedes Objekt als Instanz einer Klasse stellt schon eine Form der Wiederverwendung dar. Der Programmcode einer Klasse wird für alle Objekte wiederverwendet. Zusätzlich kann durch das Konzept der Vererbung der Programmcode in allen abgeleiteten Klassen als Erbe wiederverwendet werden. Durch diese Konzepte wird ein höherer Grad der Wiederverwendung erreicht als durch die Benutzung von Makros und Funktionsaufrufen.

Der objektorientierte Ansatz ist der heutige Standard in der Softwareentwicklung, da er sich als flexibel und leistungsfähig erwiesen hat. Doch auch die Möglichkeiten zur Wiederverwendung, die dieser Ansatz bietet, haben sich als unzureichend herausgestellt, da bei größeren Softwaresystemen der Wunsch besteht, größere Einheiten als Klassen wiederzuverwenden. Ein erster Schritt war die Verwendung von Software-Bibliotheken als wiederverwendbare Funktionssammlungen. In einer Bibliothek werden thematisch zusammengehörige Funktionen, deren Implementierung auch mehr als eine Klasse umfassen kann, gebündelt und als Gesamtpaket wiederverwendet. Ein Beispiel wäre eine Bibliothek zur Verarbeitung von Textdateien, die vorgefertigte Funktionen zum Laden, Lesen, Editieren und Speichern anbietet.

Betrachtet man die historische Entwicklung der Wiederverwendungsmöglichkeiten von Software aus Architektursicht, so haben sich auch hier die Möglichkeiten entwickelt. Wo zu Beginn der Geschichte vornehmlich monolithische

Anwendungen konstruiert wurden, die oftmals auf so genannten Host-Systemen von IBM liefen, gehen heutige Entwicklungen hin zu service-orientierten Architekturen.

Grundlage für diese Entwicklung bildet die Zerlegung von Software in Komponenten und die Möglichkeit ganze Komponenten wiederzuverwenden. Mit der komponentenbasierten Softwareentwicklung wurden Verfahren geschaffen, um derartige Komponenten zu beschreiben und die Komposition von Komponenten zu ermöglichen.

Um den Grad an Wiederverwendung noch weiter zu erhöhen, wurden für einzelne Probleme spezielle Komponenten entwickelt, die schon einen großen Teil der zu erstellenden Anwendung bereitstellten. Derartige Komponenten konnten nicht nur aufgerufen, sondern durch eigene Erweiterungen gezielt angepasst werden. Diese Art Komponenten bezeichnet man heute als *Frameworks* oder auch *Programm-Gerüste* im Deutschen. Durch die weite Verbreitung und Etablierung des englischen Begriffs, verwenden wir in dieser Arbeit die Bezeichnung »Frameworks«.

2.1.1 Einsatz von Frameworks

Ein Framework ist eine wiederverwendbare, unvollständige Softwareanwendung, die man durch Erweiterung und Anpassung vervollständigen kann. Frameworks haben ihren Ursprung in der objektorientierten Softwareentwicklung und basieren von daher auf den dort zur Verfügung stehenden Konzepten. Diese Art der Frameworks bezeichnet man genauer als »*Application Frameworks« [Johnson1988]*, da sie als Grundgerüst für neue Anwendungen dienen. Ist nur von der Kurzform Frameworks die Rede, so sind in der Regel und insbesondere in dieser Arbeit »Application Frameworks« gemeint.

Die Wiederverwendung von Software steht im Fokus eines Frameworks. Hierbei sind zwei Aspekte der Wiederverwendung zu beachten:

- Frameworks erlauben die Wiederverwendung des Programmcodes z.B. in Form von Klassen.
- Frameworks erlauben die Wiederverwendung des Designs bzw. der Softwarearchitektur.

Je nach Umfang einer Software benötigt man mehrere Frameworks, um sämtliche Anforderungen der eigenen Software abzubilden. Für einige Anforderungen gibt es eventuell keine wiederverwendbaren Frameworks, so dass hierfür eigene Lösungen entwickelt werden müssen. In jedem Fall müssen die verwendeten Frameworks durch eigene Erweiterungen zu vollständigen Anwendungen weiterentwickelt werden. Bei Verwendung mehrerer Frameworks müssen diese miteinander verbunden werden, so dass sie als Einheit verwendet werden können. Dies führt zur Entwicklung so genannter *Technologie-* oder *Software-Stacks* [Christ2008]. Die Entwicklung von Software-Stacks ist somit die Fortsetzung der Framework-Entwicklung. Wir zeigen ein Beispiel für einen solchen Software-Stack in Abbildung 2.1. Dort sehen wir eine Drei-Schichten-Architektur bestehend aus einem GUI Framework an der Spitze, gefolgt von einem

Framework für die Geschäftslogik und einen Framework zur Speicherung der Daten in einer Datenbank.

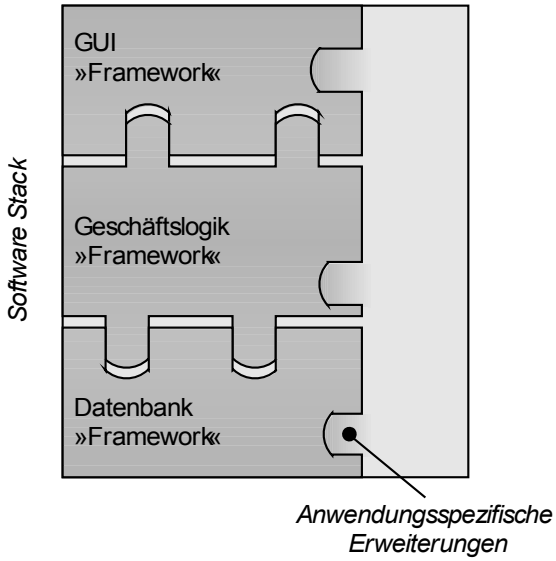


Abbildung 2.1: Schematischer Aufbau eines Software-Stacks

Ein Software-Stack besteht zu großen Teilen aus wiederverwendeten Frameworks und somit stellt ein Software-Stack seinerseits wieder ein komplexes Framework dar. Um die Interaktion zwischen den Frameworks zu ermöglichen, müssen Verbindungen geschaffen werden. Mit anderen Worten: Die Frameworks müssen miteinander *gekoppelt* werden.

Durch die Kopplung der Frameworks wird ein höherer Wiederverwendungsgrad erreicht, da mehrere Frameworks gleichzeitig wiederverwendet werden. Aus diesem Blickwinkel erkennt man, dass Software-Stacks eine Weiterentwicklung der Wiederverwendungsmöglichkeiten darstellen. Je mehr Software wiederverwendet werden kann, umso kostengünstiger und schneller lässt sich neue Software auf dieser Basis entwickeln. Der Einsatz von Software-Stacks ist in diesem Sinne eine sehr ökonomische Vorgehensweise in der Softwareentwicklung.

Eine besondere Form Software-Stack bilden die so genannten *Open-Source-Stacks* (OSS) [Christ2008]. Wie man aus dem Namen »OSS« bereits ableiten kann, sind die eingesetzten Frameworks als Open-Source-Software (OS-Software) verfügbar. Der Einsatz von OS-Software hat sich mit dem Erfolg von Linux neben dem Bereich der Betriebssysteme in vielen weiteren Bereichen etabliert. Beispiele sind Web-Anwendungen oder mobile Anwendungen auf Smartphones. Die Praxis des OS-Softwareentwicklungsprozesses mit einer unabhängigen Gemeinschaft ehrenamtlicher Entwickler hat gezeigt, dass mit diesem Ansatz qualitativ hochwertige Software entwickelt werden kann. Der Gemeinschaftsgeist und die Möglichkeit neue Ideen mit Vielen zu teilen und zu verwirklichen, machen die OS-Bewegung auch zu einem Motor für Innovationen.

In einem OSS werden OS-Frameworks und OS-Bibliotheken integriert. Die Bestandteile agieren so miteinander, dass sie eine Einheit bilden und als solche eine Basis für die Entwicklung neuer Software darstellen. Wir definieren einen OSS wie folgt:

> *Ein Open-Source-Stack (OSS) ist die Basis für die Entwicklung neuer Software. Diese Basis besteht aus einem vorkonfigurierten Verband von Open-Source-Komponenten, die im Zusammenspiel als Einheit agieren und als Einheit benutzt werden.*
>
> **Open-Source-Stack nach [Christ2008]**

Betrachten wir ein Beispiel für einen OSS mit einer Drei-Schichten-Architektur bestehend aus Präsentations-, Anwendungs- und Persistenzschicht. Innerhalb des OSS wird in jeder Schicht der Architektur ein OS-Framework eingesetzt, das jeweils architektonisch und funktional zu den Anforderungen des Stacks passen muss. Für weitere Details betrachten wir den schematischen Aufbau eines solchen Drei-Schicht-OSS in Abbildung 2.2.

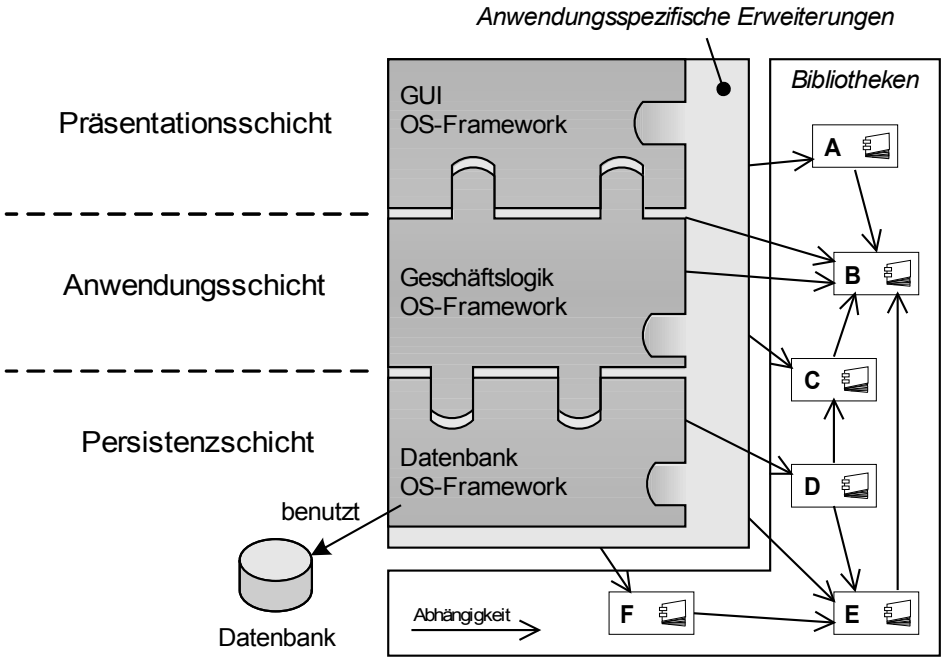


Abbildung 2.2: Schematischer OSS für eine 3-Schichten-Architektur

Die Herausforderung bei der Erstellung eines OSS ist die Beherrschung der schnell komplex werdenden Abhängigkeiten. Die Abhängigkeiten beginnen damit, dass die eingesetzten Frameworks über ihre Erweiterungspunkte miteinander interagieren können müssen. Hinzu kommt der Anteil anwendungsspezifischer Erweiterungen und der damit verbundene Programmcode der eigentlichen Anwendung. Die weitere Komplexität der Abhängigkeiten entsteht nun insbesondere dadurch, dass es nicht nur Abhängigkeiten über die verwen-

deten Erweiterungspunkte gibt, sondern zusätzlich zu und zwischen verwendeten Bibliotheken.

Wie bei vielen Anwendungen bilden Bibliotheken auch bei OSS eine Grundlage von Funktionen, die über Grenzen von Anwendungsschichten hinweg benötigt werden. Beispiele hierfür sind Funktionen für reguläre Ausdrücke auf Zeichenketten, um Log-Meldungen zu erzeugen oder bestimmte Datenformate wie XML zu verarbeiten. Häufig benötigen komplexere Bibliotheken wiederum andere Bibliotheken, um zu funktionieren. Auch in Anwendungen, die keine Frameworks, sondern nur Bibliotheken einsetzen, entsteht schnell ein komplexes Abhängigkeitsgeflecht aus Bibliotheken, die wie Frameworks versioniert werden.

Neben dem Einsatz in Software-Stacks können auch in konkreten Anwendungen mehrere Frameworks gleichzeitig zum Einsatz kommen. Die Fähigkeit zur Kopplung ermöglicht es, dass für verschiedene Aufgabenbereiche unterschiedliche Frameworks zum Einsatz kommen, die miteinander interagieren können. Aus dieser Überlegung und der Aufgabenstellung können wir ableiten, dass das Management einer solchen Anwendung mit verschiedenen Frameworks nur praktikabel ist, wenn man Frameworks anhand einer Version unterscheiden kann. Wie jede Anwendung unterliegen auch Frameworks Weiterentwicklungszyklen, die durch unterschiedliche Versionen gekennzeichnet werden.

Nach diesen eher grundsätzlichen Überlegungen über den Einsatz von Frameworks wollen wir die Auffassung darüber, was ein Framework eigentlich ist, konkretisieren und Einsatzszenarien für Frameworks betrachten.

2.1.2 Frameworks als Grundgerüst

Die intuitive Vorstellung von einem Framework ist meist die, dass ein Framework das Grundgerüst einer Anwendung liefert, auf dem eine neue Anwendung entwickelt werden kann. Das Framework bietet eine Fülle von Funktionen an, die genutzt werden können, um sich im Prinzip nur noch mit der speziellen Anwendungsentwicklung beschäftigen zu müssen. Man setzt voraus, dass die allgemeinen Probleme bereits vom Framework gelöst wurden. So kann das Framework bei richtigem Einsatz die Entwicklungszeit im Vergleich zu einer Entwicklung ohne Framework deutlich verkürzen.

Ein Beispiel soll diese Vorstellung verdeutlichen: Die Aufgabe ist, eine Web-Anwendung zu entwickeln, die über einen Web-Browser bedient werden kann. Die Anwendung muss also Anfragen entgegennehmen, die über das Netzwerk mittels des HTTP-Protokolls transportiert werden. Anschließend muss die Anwendung abhängig von der Anfrage dynamisch eine Antwort generieren und im HTML-Format zurück an den Web-Browser senden. Abbildung 2.3 stellt den Ablauf in Form eines UML-Aktivitätendiagramms dar.

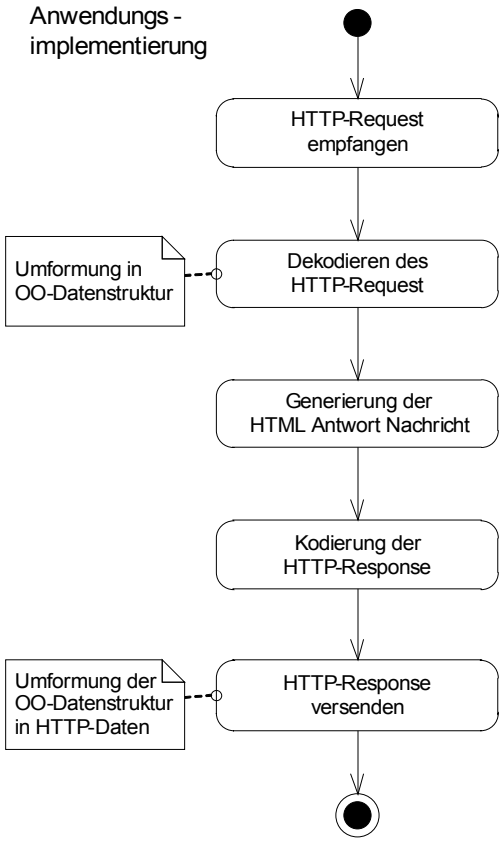


Abbildung 2.3: Aufgabenstellung für die Entwicklung einer Web-Anwendung

Stellt man sich nun vor, dass man mehrere solcher Anwendungen entwickeln muss, kommt man zu dem Schluss, dass bestimmte Teile dieser Anwendung in jeder Web-Anwendung benötigt werden. Überlegt man sich welche Teile unverändert bleiben, wird klar, dass der Empfang sowie das Dekodieren des HTTP-Requests und das Kodieren mit dem Versand der HTTP-Response immer identisch sind. Der einzig variable Teil ist die inhaltliche Generierung der HTML-Nachricht, die in jeder Web-Anwendung unterschiedlich sein wird.

Mit dieser Überlegung kann man ein Framework entwickeln, das den Empfang und den Versand von HTTP-Nachrichten übernimmt und an dem Punkt erweitert werden kann, an dem die inhaltliche HTML-Nachricht erstellt werden soll. Abbildung 2.4 stellt dieses Framework in einer Übersicht dar. Es wird deutlich, dass ein Softwareentwickler nur noch die entsprechende Erweiterung implementieren muss. Um die übrigen Teile der Aufgabe braucht er sich keine Gedanken zu machen, da diese vom Framework bereits gelöst sind. Die Entwicklung kann in kürzerer Zeit zu geringeren Kosten erfolgen.

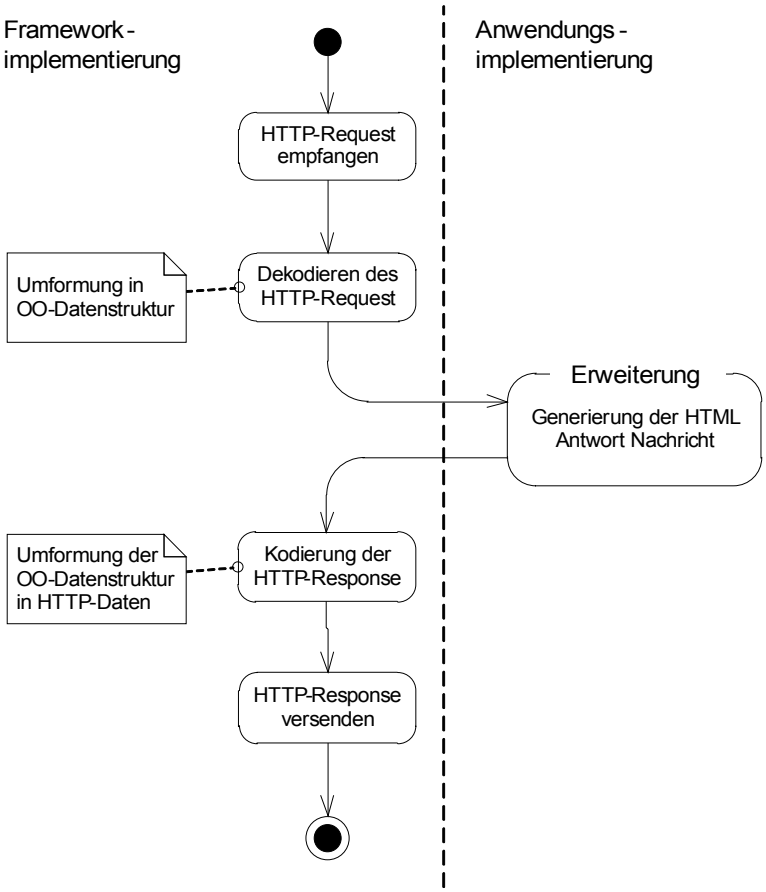


Abbildung 2.4: Framework für die Entwicklung von Web-Anwendungen

Schon dieses einfache Beispiel zeigt einige Framework-Merkmale, die wir festhalten wollen. Zunächst einmal dienen Frameworks der Wiederverwendung von größeren Programmeinheiten. Die Erweiterbarkeit an definierten Erweiterungspunkten macht sie wiederverwendbar für Anwendungsentwickler. Außerdem können Frameworks offensichtlich einen eigenen Kontrollfluss haben, den sie zeitweise an eine Erweiterung übergeben und dann von diesem wieder übernehmen können.

Neben diesem intuitiven Verständnis für Frameworks wollen wir das Beispiel einer Web-Anwendung auf Basis eines Frameworks im folgenden Abschnitt vertiefen, um einige Besonderheiten in den Details eines Frameworks herauszuarbeiten.

2.1.3 Ein Framework für Web-Anwendungen

Die Java Servlet-Technologie [Sun2003] ist ein von der Firma Sun Microsystems spezifizierter Standard zur Implementierung von Web-Anwendung in der Programmiersprache Java. Die Servlet-Spezifikation [Sun2003] definiert ein Servlet als eine Komponente zur dynamischen Generierung von Inhalten, die von einem Container verwaltet wird. Der Container wird häufig als Servlet-Engine bezeichnet. Servlets kommunizieren mit Clients über das Anfrage-Ant-

wort-Schema, das von der Servlet-Engine implementiert wird. Eine solche Servlet-Engine ist häufig Bestandteil eines Anwendungsservers (engl. application server) oder allgemein eines Web-Servers.

In diesem Beispiel wollen wir den *Jetty* [Jetty2008] Web-Server näher betrachten. Jetty ist eine in der Programmiersprache Java geschriebene Servlet-Engine, die das Anfrage-Antwort-Schema für HTTP implementiert. Jetty ist ein modular aufgebautes Framework und verfügt über vielfältige Erweiterungsmöglichkeiten. Abbildung 2.5 gibt uns einen Überblick, über den grundlegenden Aufbau und die Arbeitsweise von Jetty.

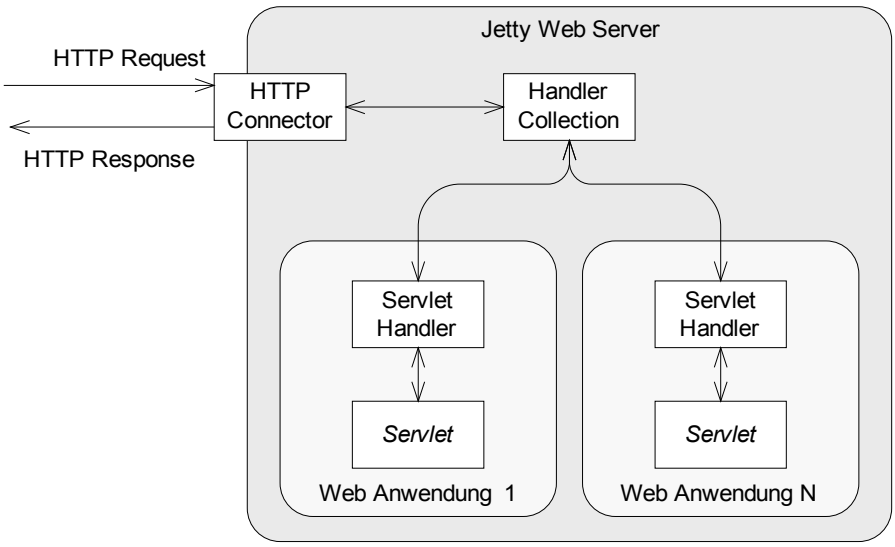


Abbildung 2.5: Jetty Architekturüberblick

Jetty nimmt HTTP-Requests von Clients über einen HTTP-Connector entgegen und leitet diese an vorhandene Web-Anwendungen weiter. Die Handler-Collection ist ein Verteiler, der die Requests an entsprechende Servlet-Handler in den Web-Anwendungen weitergibt, die wiederum das passende Servlet mit dem Request aufrufen. Die Antwort (HTTP-Response) auf den HTTP-Request wird am Ende der Kette zurückgereicht und über den HTTP-Connector schließlich an den Client gesendet.

Eine Web-Anwendung bildet gemäß der Servlet-Spezifikation einen Erweiterungspunkt im Jetty-Framework. Zur Implementierung einer entsprechenden Erweiterung, also einer Web-Anwendung, muss der Framework-Benutzer ein Servlet implementieren. Ein Servlet muss hierfür die vom Framework vorgegebene Servlet-Schnittstelle aus Abbildung 2.6 implementieren.

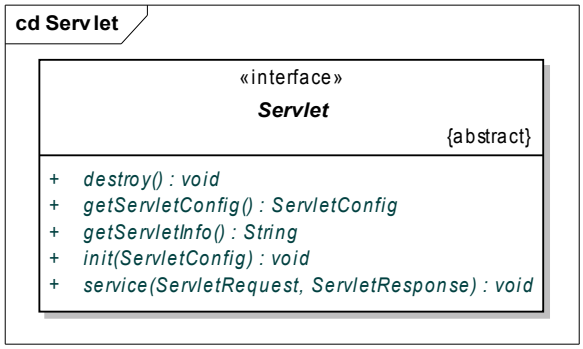


Abbildung 2.6: Die `Servlet`-Schnittstelle

Die Servlet-Schnittstelle definiert unter anderem den Lebenszyklus des Servlets. Dieser Lebenszyklus beginnt mit dem Aufruf von `init`. Anschließend werden bis zum Lebensende des Servlets `ServletRequests` über die `service` Methode übergeben. Wird das Servlet beendet, wird abschließend die `destroy` Methode aufgerufen. Die einzelnen Funktionen der Schnittstelle sind in Tabelle 2.1 beschrieben.

Tabelle 2.1: Funktionen der Servlet-Schnittstelle

Signatur	Beschreibung
`destroy(): void`	Wird zum Lebensende des Servlets aufgerufen bevor das Servlet-Objekt zerstört wird.
`getServletConfig(): ServletConfig`	Gibt die aktuelle `ServletConfig` zurück.
`getServletInfo(): String`	Gibt Informationen zum Servlet wie Name, Autor, Copyright etc. in einem `String` zurück.
`init(ServletConfig): void`	Wird zur Initialisierung des Servlet aufgerufen, wobei eine `ServletConfig` zur Konfiguration übergeben wird.
`service(ServletRequest, ServletResponse): void`	Bei jedem Request, den das Servlet verarbeiten soll, wird diese Methode aufgerufen. Das Servlet verändert das `ServletResponse` Objekt, um die Antwort zu generieren.

Die Servlet-Schnittstelle ist nicht auf das HTTP Protokoll beschränkt. Um ein Servlet speziell für HTTP zu implementieren, gibt es die Klasse `HttpServlet`, die für eine Erweiterung erweitert werden muss. Den Zusammenhang zwischen der `Servlet`-Schnittstelle und dem `HttpServlet` zeigt das Klassendiagramm in Abbildung 2.7.

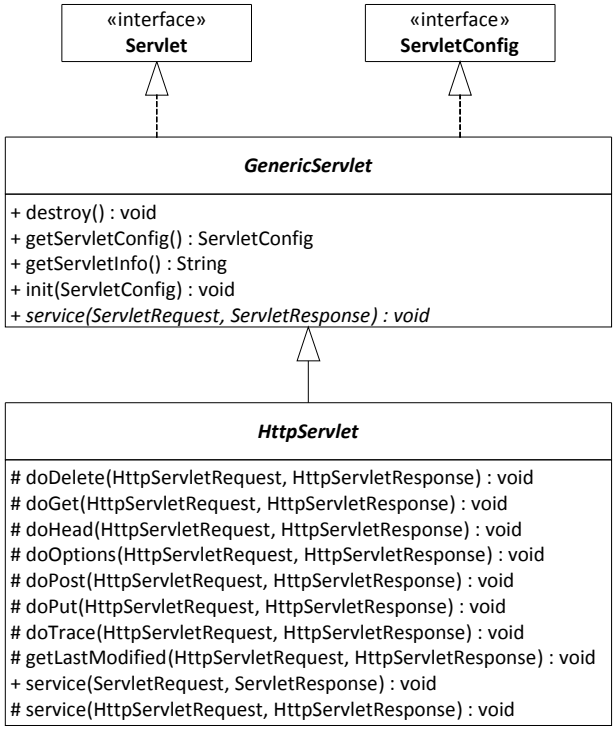


Abbildung 2.7: `HttpServlet` Basisklassen

Die abstrakte Klasse `GenericServlet` bildet eine rudimentäre Implementierung einer `Servlet`-Schnittstelle. Diese Implementierung wird von der spezielleren Klasse `HttpServlet` erweitert. Das folgende Sequenzdiagramm in Abbildung 2.8 spezifiziert den operationalen Ablauf bei Aufruf der `service(ServletRequest, ServletResponse)` Methode eines `HttpServlet`.

Bei Verwendung des `HttpServlet` wird ausgehend von der `service(ServletRequest, ServletResponse)` Methode der Request in einen `HttpServletRequest` umgeformt und an die `service(HttpServletRequest, HttpServletResponse)` Methode übergeben. Diese ruft dann, je nachdem welche HTTP-Methode im `HttpServletRequest` gesetzt ist, eine der `do*` Methoden auf. Aus Gründen der Übersichtlichkeit haben wir darauf verzichtet alle Alternativen für die `do*` Methoden aufzulisten.

Eine Implementierung eines Servlet für HTTP erweitert die `HttpServlet` Klasse und überschreibt die gewünschten `do*` Methoden, um Antworten auf die unterschiedlichen HTTP Anfragen wie GET und POST zu verarbeiten. In der Regel werden die Methoden `doGet` und `doPost` überschrieben. Alternativ kann auch hier eine der beiden `service` Methoden direkt überschrieben werden. Außerdem sollten die Methoden aus `GenericServlet`, wie `getServletInfo`, überschrieben werden und sinnvolle Werte zurückzugeben.

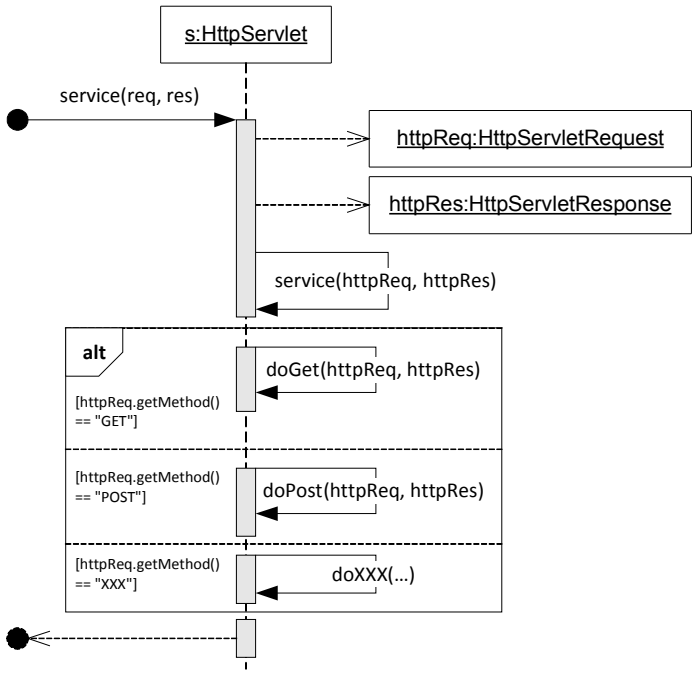


Abbildung 2.8: HttpServlet Service Ablauf

Damit das eigene Servlet vom Jetty Server aufgerufen wird, muss es in die Konfigurationsdatei der Web-Anwendung entsprechend eingetragen werden. Diese Konfigurationsdatei, auch als »deployment descriptor« bezeichnet, trägt den Namen `web.xml` und liegt im Verzeichnis `WEB-INF` der Web-Anwendung. Die Servlet-Spezifikation [Sun2003] schreibt für Web-Anwendungen eine bestimmte Verzeichnisstruktur vor, die in Listing 2.1 wiedergegeben ist.

Listing 2.1: Web-Anwendung Verzeichnisstruktur

```
1  +-- WEB-INF
2      +-- classes
3      +-- lib
4      web.xml
```

Im Verzeichnis `classes` der Verzeichnisstruktur wird die Implementierung des Servlets abgelegt. Das Verzeichnis `lib` kann weitere benötigte Bibliotheken in Form von JAR-Dateien aufnehmen. Die Verzeichnisse `classes` und `lib` bilden zusammen den Klassenpfad (engl. classpath) für die Ausführung des Servlets.

Die Datei `web.xml` ist eine XML-Datei, in der diverse Einstellungen und Parameter für die Web-Anwendung festgelegt werden können. Unter anderem werden die Servlets aufgelistet und es wird definiert, bei welchem URL-Muster welches Servlet aufgerufen werden soll. Listing 2.2 zeigt die wichtigsten Eigenschaften zur Konfiguration eines Servlets als Auszug aus dem XML-Schema des »Servlet 2.4 Deployment Descriptor«.

Listing 2.2: Auszug aus XML-Schema des Servlet 2.4 Deployment Descriptor

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
3        targetNamespace="http://java.sun.com/xml/ns/j2ee"
4        xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
5        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6        version="2.4">
7
8     <xsd:element name="web-app" type="j2ee:web-appType">
9        <xsd:annotation>
10          <xsd:documentation>
11             The web-app element is the root of the deployment
12             descriptor for a web application.
13          </xsd:documentation>
14       </xsd:annotation>
15       <xsd:unique name="web-app-servlet-name-uniqueness">
16          <xsd:annotation>
17             <xsd:documentation>
18                The servlet element contains the name of a servlet.
19                The name must be unique within the web application.
20             </xsd:documentation>
21          </xsd:annotation>
22          <xsd:selector xpath="j2ee:servlet" />
23          <xsd:field xpath="j2ee:servlet-name" />
24       </xsd:unique>
25    </xsd:element>
26
27    <xsd:complexType name="servletType">
28       <xsd:sequence>
29          <xsd:element name="servlet-name"
30                       type="j2ee:servlet-nameType" />
31          <xsd:choice>
32             <xsd:element name="servlet-class"
33                      type="j2ee:fully-qualified-classType">
34             </xsd:element>
35          </xsd:choice>
36       </xsd:sequence>
37    </xsd:complexType>
38
39    <xsd:complexType name="servlet-mappingType">
40       <xsd:annotation>
41          <xsd:documentation>
42             The servlet-mappingType defines a mapping
43             between a servlet and a url pattern.
44          </xsd:documentation>
45       </xsd:annotation>
46
47       <xsd:sequence>
48          <xsd:element name="servlet-name"
49             type="j2ee:servlet-nameType" />
50          <xsd:element name="url-pattern"
51             type="j2ee:url-patternType" />
52       </xsd:sequence>
53       <xsd:attribute name="id" type="xsd:ID" />
54    </xsd:complexType>
55
56    <xsd:complexType name="web-appType">
57       <xsd:choice minOccurs="0" maxOccurs="unbounded">
58          <xsd:element name="servlet"
59             type="j2ee:servletType" />
60          <xsd:element name="servlet-mapping"
61             type="j2ee:servlet-mappingType" />
62       </xsd:choice>
63    </xsd:complexType>
```

```
64 </xsd:schema>
```

Das `web-app` Tag ist das Wurzelelement einer `web.xml`. Für die Konfiguration eines Servlets werden noch die Elemente `servlet` und `servlet-mapping` benötigt. Das `servlet` Element definiert einen in der Web-Anwendung eindeutigen Namen (`servlet-name`) für das Servlet und den voll qualifizierten Klassennamen der Implementierung. Damit dieses Servlet bei bestimmten URL Aufrufen angesprochen wird, muss noch ein `servlet-mapping` definiert werden. Es beschreibt ein entsprechendes URL-Muster (`url-pattern`) das einem Servlet zugeordnet wird.

Zusammenfassend sind drei Schritte für eine Implementierung einer Web-Anwendung als Erweiterung des Jetty-Frameworks erforderlich:

1. Implementierung der Servlet-Schnittstelle aus Abbildung 2.7
2. Anlage der Verzeichnisstruktur gemäß Listing 2.1
3. Anlage bzw. Erweiterung einer `web.xml` als Instanz des Schemas in Listing 2.2

Nach diesem konkreten Beispiel der technischen Details eines Frameworks, betrachten wir nachfolgend existierende Definitionen sowie unterschiedliche Arten und Ausprägungen von Frameworks.

2.2 Definitionen von Frameworks

In diesem Abschnitt betrachten wir Definitionen aus der Literatur, um den Begriff des Frameworks nach den einführenden Beispielen genauer zu erfassen. Im Speziellen betrachten wir Definitionen für »Application Frameworks«, um aus diesen Framework-Merkmale abzuleiten. Die erste und älteste Definition, auf die wir uns beziehen, stammt aus [Johnson1988]:

> *A framework [...] is an abstract design for a particular kind of application, [...]*
>
> **Framework nach [Johnson1988]**

Nach dieser Definition transportiert ein Framework ein abstraktes Design, das man auch als Architektur bezeichnen kann. Wörtlich übersetzt sagt die Definition sogar: „ein Framework *ist* ein abstraktes Design“. Auf einer etwas technischeren Ebene betont Deutsch noch mehr die Tatsache, dass es bei einem solchen Design bzw. einer solchen Architektur um Wiederverwendung geht [Deutsch1989]:

> *A framework is a reusable design of a program or a part of a program expressed as a set of classes.*
>
> **Framework nach [Deutsch1989]**

Diese wiederverwendbare Architektur dient somit einer bestimmten Art von Anwendung, ist also für Anwendungen einer bestimmten Domäne bestimmt.

Wir können also festhalten:

- Ein Framework ist ein wiederverwendbares abstraktes Design.
- Ein Framework ist für Anwendungen einer bestimmten Domäne konzipiert.

Die Fokussierung auf eine bestimmte Domäne bzw. auf eine bestimmte Klasse von Problemen, die mit einem spezifischen Framework gelöst werden können, wird von Johnson in einer weiteren Charakterisierung von Frameworks in den Mittelpunkt gerückt [Johnson1992]:

> *A framework is a reusable design for solutions to problems in some particular problem domain.*
>
> **Framework nach [Johnson1992]**

Auch Schmid betont noch einmal, dass die Wiederverwendung eines Frameworks bedeutet, dass man Anwendungen auf Basis eines Frameworks erstellt. So wird die Anwendung zu einer Spezialisierung der Domäne des Frameworks [Schmid1997]:

> *A framework is a generic application that allows the creation of different applications from an application (sub) domain.*
>
> **Framework nach [Schmid1997]**

Außerdem wird klar, ein Framework *ist* bereits eine generische Anwendung, die spezialisiert werden kann. Der Gedanke der Spezialisierung bei Verwendung eines Frameworks wird auch von Fayad und Schmidt aufgegriffen, die allgemein ein Framework als halb-fertige Anwendung verstehen, die man durch Spezialisierung zu einer spezifischen Anwendung weiterentwickeln kann [Fayad1997]:

> *A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications.*
>
> **Framework nach [Fayad1997]**

Nach der ersten Definition aus dem Jahr 1988 verfeinert Johnson die Definition für Frameworks einige Jahre später und offenbart hiermit weitere Framework-Merkmale [Johnson1997a]:

> *Structure: A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact.*
>
> *Purpose: A framework is the skeleton of an application that can be customized by an application developer.*
>
> **Framework nach [Johnson1997a]**

Zunächst wird die Definition zweigeteilt: Der erste Teil beschreibt die Struktur, die ein Framework ausmacht und der zweite Teil die Zweckbestimmung, die ein Framework erfüllt.

Diese Definition beschreibt ein Framework als wiederverwendbares Design. Das Design kann das gesamte zu erstellende System umfassen oder nur einen Systemteil betreffen. Verwirklicht wird dies durch den Einsatz abstrakter Klassen und der Art wie deren Instanzen interagieren.

Der Zweck eines Frameworks zeigt sich darin, dass ein Framework das Skelett oder Grundgerüst für eine bestimmte Art von Anwendung ist. Dieses Grundgerüst wird jedoch erst durch Erweiterungen und Anpassungen, die ein Softwareentwickler vornimmt, zu einer funktional vollständigen Anwendung.

Ein Framework bildet somit ein, in Teilen abstraktes, Grundgerüst für die Entwicklung von Anwendungen, wobei der Softwareentwickler die abstrakten Teile so erweitert, dass sie seinen Anforderungen an die zu erstellende Anwendung entsprechen.

Aus dieser Definition halten wir fest:

- Ein Framework ist ein wiederverwendbares Design, das mit den Mitteln der objektorientierten Abstraktion realisiert werden kann.
- Ein Framework ist ein Anwendungsgrundgerüst, dass durch einen Softwareentwickler durch Benutzung der Erweiterungspunkte zu einer funktional vollständigen Anwendung erweitert wird.

Die gezielte Anpassbarkeit eines Frameworks über objektorientierte Abstraktion ist nah verwandt mit dem Einsatz von Design-Patterns [Gamma1993], da Design-Patterns ihren Ursprung in der Untersuchung von Frameworks auf wiederverwendbares Design haben. Die Möglichkeiten der objektorientierten Programmierung bieten geeignete Konstrukte, die es erlauben, ein Framework als wiederverwendbares, abstraktes Design zu entwickeln und an definierten Erweiterungspunkten Anpassungen zuzulassen.

2.2.1 Arten von Frameworks

Wir können verschiedene Arten von Frameworks unterscheiden. Die Unterscheidungskriterien sind zum einen die Art, wie das Framework erweitert wird [Fayad1997] und zum anderen der Typ der Erweiterungen.

Bei der Art der Erweiterungen muss man unterscheiden, welche Informationen das Framework preisgibt. Ein *Blackbox-Framework* gibt keine Informatio-

nen über den internen Aufbau des Frameworks nach außen. Blackbox-Frameworks definieren benötigte Schnittstellen für Komponenten, die in das Framework eingehängt werden können. Der Erweiterungsmechanismus beruht auf dem Prinzip der Komposition [Fayad1997].

Im Gegensatz zu einem Blackbox-Framework propagiert ein *Whitebox-Framework* seinen internen Aufbau nach außen, so dass flexiblere objektorientierte Mechanismen zur Implementierung von Erweiterungen genutzt werden können. Whitebox-Frameworks werden zum Beispiel durch Vererbung oder durch dynamische Bindungen von Klassen erweitert. Diese Art der Erweiterung ist mit einem Blackbox-Framework nicht möglich, da entsprechende Interna nicht bekannt sind [Fayad1997].

Neben der Art der Erweiterung können wir zusätzlich den Typ der Erweiterungen betrachten. Objektorientierte Frameworks, die man auch als Anwendungs-Frameworks (engl. Application Frameworks) bezeichnet, verwenden Objekte als Erweiterung. Bei dieser Art implementieren Anwendungen Klassen gemäß den Vorgaben eines Erweiterungspunktes. Instanzen dieser Klassen bilden zur Laufzeit die eigentliche Erweiterung.

Verwendet man komplexere Einheiten als Objekte für die Erweiterung spricht man häufig von Komponenten-Frameworks (engl. Component Frameworks). In einem solchen Framework werden Erweiterungen durch Komponenten realisiert. Wir werden Merkmale von Software-Komponenten als verwandtes Konzept zu Frameworks in Abschnitt 2.3.1 noch genauer betrachten. Komponenten-Frameworks werden häufig als Blackbox-Frameworks konzipiert, da Erweiterungen über Kompositionsmechanismen in das Framework eingehängt werden können.

Eine weitere Klasse von Frameworks entsteht, wenn die Erweiterungen ganze Software-Systeme sind, die in einer Gesamtarchitektur des Unternehmens zusammengeschaltet werden sollen. Diese Form von Frameworks bezeichnet man als Unternehmens-Framework (engl. Enterprise Frameworks). Ein Unternehmens-Framework steuert komplexere Prozesse im Unternehmen und erlaubt die Anbindung von Software-Systemen an definierten Erweiterungspunkten. In diesem Zusammenhang wird häufig auch das Prinzip einer serviceorientierten Architektur genannt, die zum Beispiel durch den Einsatz eines Unternehmens-Frameworks umgesetzt oder unterstützt werden kann.

Es lassen sich je nach Erweiterungstyp sicherlich noch weitere Arten von Frameworks unterscheiden. Allen in dieser Arbeit betrachteten Frameworks ist jedoch gemein, dass sie über definierte Erweiterungspunkte angepasst werden können.

2.2.2 Benutzer von Frameworks

In dieser Arbeit beziehen wir uns immer wieder auf eine Reihe von Benutzerrollen, die bei der Entwicklung und Verwendung von Frameworks zentral sind. Zunächst haben wir den »Projektleiter«, der für die Planung und Durchführung eines Softwareentwicklungsprojektes verantwortlich ist. In Bezug auf Frameworks steht der Projektleiter häufig vor dem Problem, dass er entschei-

den muss, ob die Aktualisierung eines Frameworks durchgeführt werden soll oder nicht.

Der »Software-Architekt« hingegen ist für die Planung der Software im Sinne ihrer Architektur unter Beachtung gegebener Anforderungen verantwortlich. Der Software-Architekt bestimmt in erster Linie was konstruiert werden soll. Mit welchen technischen Details die Architektur umgesetzt wird, bestimmen ein oder mehrere »Software-Entwickler«.

Die Rolle des Software-Entwicklers kann im Bezug auf den Einsatz und die Entwicklung von Frameworks weiter unterteilt werden. Wir unterscheiden den »Framework-Benutzer«, der als Software-Entwickler ein Framework verwendet und Erweiterungen für dieses programmiert. Daneben gibt es noch den »Framework-Entwickler«, der als Software-Entwickler das Framework selbst entwickelt und für die Implementierung und Dokumentation des Frameworks samt seiner Erweiterungspunkte verantwortlich ist.

Die beschriebenen Rollen sind in der folgenden Abbildung 2.9 noch einmal mit Hilfe eines UML-Akteurmodells [UMLSUP23] wiedergegeben. Hier wird nochmals deutlich, dass Framework-Benutzer und -Entwickler jeweils Software-Entwickler sind, die verschiedene Perspektiven bei der Arbeit mit Frameworks repräsentieren.

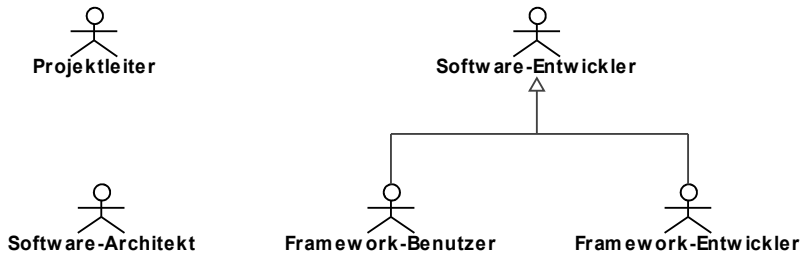


Abbildung 2.9: Rollen

2.3 Das Framework-Benutzungs-Modell

Aus der vorangegangenen Einführung, den Beispielen und Definitionen für Frameworks ergeben sich eine Reihe von Merkmalen bezüglich der Benutzung von Frameworks. Um die zentralen Framework-Merkmale zu veranschaulichen, überführen wir sie in eine Übersichtsdarstellung des Framework-Konzeptes. Hierzu erweitern wir in Abbildung 2.10 das schematische Bild aus Abbildung 1.1. Es zeigt eine Anwendung, die ein Framework benutzt. Eine solche Anwendung implementiert mindestens eine Erweiterung des Frameworks. Da ein Framework beliebig viele Erweiterungspunkte haben kann, unterscheiden wir in *benutzte* und *nicht benutzte* Erweiterungspunkte. Zur Benutzung einer Erweiterung müssen drei Arten von Schnittstellen implementiert werden, die jeder Erweiterungspunkt definiert. Dies sind die *Erweiterungsschnittstelle*, die *Bindungsschnittstelle* und die *Lebenszyklusschnittstelle*.

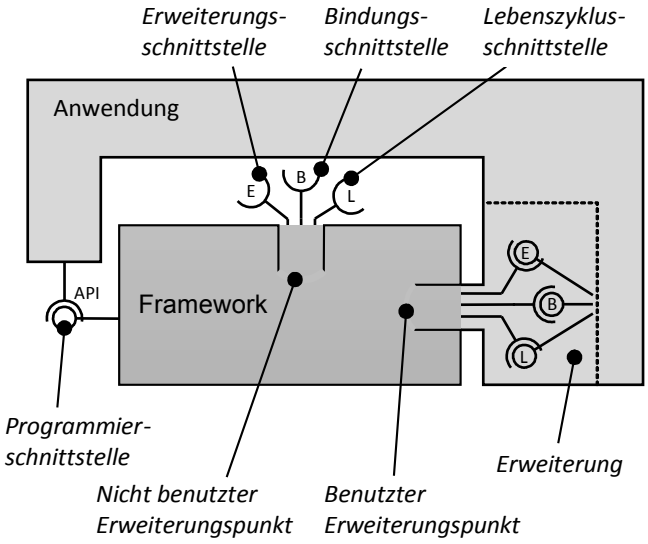


Abbildung 2.10: Schematisches Framework-Benutzungs-Modell

Die Erweiterungsschnittstelle definiert die fachliche Funktion des Erweiterungspunktes. Diese fachliche Schnittstelle muss von einer Erweiterung implementiert werden, um die fachliche Funktionalität des Frameworks an dieser Stelle zu erweitern. Die Bindungs- und Lebenszyklusschnittstellen sind eher technischer Natur. Die Bindungsschnittstelle definiert den Mechanismus, über den die Erweiterung an das Framework gebunden wird. Die Bindung ist notwendig, damit das Framework die Erweiterung finden und gemäß der Lebenszyklusschnittstelle initialisieren kann. Die Lebenszyklusschnittstelle definiert somit, wie eine Erweiterung initialisiert wird und welche Phasen sie durchläuft. In der Phase des Lebenszyklus, in der die Erweiterung fachlich bedient wird, kommt die Erweiterungsschnittstelle zum Tragen.

Zur besseren Vergleichbarkeit des Framework-Konzeptes mit anderen Konzepten wie Software-Komponenten und -Bibliotheken verwenden wir ein Framework-Benutzungs-Modell. Wir definieren das Framework-Benutzungs-Modell als Taxonomie, die auf der obersten Ebene aus acht Kategorien besteht. Diese acht Kategorien bilden ein Vergleichsschema für softwaretechnische Konzepte, auf deren Basis wir das Framework-Konzept mit anderen Konzepten vergleichen wollen. Dieses Schema erhebt keinen Anspruch auf Vollständigkeit, aber es wird uns helfen die unterschiedlichen Konzepte zu unterscheiden. Wir stellen das verwendete Schema in Tabelle 2.2 vor.

Tabelle 2.2: Vergleichsschema für softwaretechnische Konzepte

Kategorie	Erläuterung
Charakter	Welche beschreibenden Charakterzüge kann man dem Konzept zuordnen?
Einsatz	Wie wird das Konzept softwaretechnisch eingesetzt?
Verhalten	Wie wird der Aspekt des Verhaltens der Software vom Konzept berücksichtigt?

Kategorie	Erläuterung
Wiederverwendung	Wie wird der Aspekt der Wiederverwendung von Software vom Konzept unterstützt?
Konfiguration	Wie erfolgt die Konfiguration der Software nach diesem Konzept?
Integration	Wie erfolgt die Integration der Software gemäß dem Konzept?
Anpassung	Wie können Anpassungen der Software gemäß dem Konzept durchgeführt werden?
Identifikation	Wie wird Software gemäß dem Konzept eindeutig identifiziert?

Mit Hilfe dieses Schemas und unseren Betrachtungen zum Framework-Konzept können wir nun die Merkmale von Frameworks in Tabelle 2.3 auflisten und für spätere Referenzierung festhalten. Zusätzlich werden wir diese Merkmale anschließend in ein Framework-Benutzungs-Modell überführen.

Tabelle 2.3: Merkmale von Frameworks

Kategorie	ID	Merkmal Ein Framework ...	Erläuterung
Charakter	FM-1	… ist ein unvollständiges System.	Ein Framework ist ein unvollständiges Softwaresystem. Es bildet das Grundgerüst für neue Anwendungen.
Einsatz	FM-2	… ist zweckgebunden.	Ein Framework wird für einen bestimmten Zweck entwickelt. Der erfüllte Zweck bestimmt sich durch das domänenspezifische und architektonische Problem, das mit dem Framework gelöst wird.
	FM-3	… ist ein Architekturbaustein.	Ein Framework ist häufig ein Architekturbaustein im Gefüge einer übergeordneten Gesamtarchitektur.
Verhalten	FM-4	… besitzt einen eigenen Kontrollfluss.	Ein Framework hat einen eigenen Kontrollfluss, der den Aufrufzeitpunkt von Erweiterungen definiert. Der Kontrollfluss kann temporär an Erweiterungen übergeben werden.
Wiederverwendung	FM-5	… erlaubt Wiederverwendung der Funktionalität.	Ein Framework erlaubt die Wiederverwendung der angebotenen fachlichen Funktionalität.
	FM-6	… erlaubt Wiederverwendung der Architektur.	Ein Framework erlaubt die Wiederverwendung der Architektur, die es für seine Erweiterungen definiert.

Kategorie	ID	Merkmal Ein Framework ...	Erläuterung
Konfiguration & Integration	FM-7	… besitzt eine Programmierschnittstelle.	Ein Framework bietet eine Schnittstelle, um das Framework zu integrieren und zu konfigurieren. Diese Schnittstelle wird meist als allgemeine Programmierschnittstelle (API) bezeichnet.
Anpassung	FM-8	… ist erweiterbar.	Ein Framework kann durch Erweiterungen erweitert werden. Erweiterungen erfolgen an definierten Erweiterungspunkten.
Identifikation	FM-9	… ist versionierbar.	Frameworks werden für eine eindeutige Identifizierung versioniert.

Ein zentraler Aspekt des Framework-Konzept ist die Anpassung durch Erweiterungen. Wir halten daher in Tabelle 2.4 eine Reihe weiterer Merkmale für Erweiterungspunkte festhalten.

Tabelle 2.4: Merkmale von Erweiterungspunkten

ID	Merkmal Ein Erweiterungspunkt ...	Erläuterung
EM-1	… hat eine fachliche Schnittstelle.	Ein Erweiterungspunkt definiert eine Erweiterungsschnittstelle, die von der Erweiterung implementiert werden muss. Die Erweiterungsschnittstelle definiert die fachliche Beziehung zwischen Framework und Erweiterung.
EM-2	… hat eine Bindungsschnittstelle.	Um eine Erweiterung mit dem Framework zu verbinden, definiert ein Erweiterungspunkt eine Bindungsschnittstelle, die von der Erweiterung implementiert werden muss. Ohne Bindungsschnittstelle könnte das Framework die Erweiterung nicht binden und somit verwenden.
EM-3	… hat eine Lebenszyklusschnittstelle.	Eine Erweiterung durchläuft einen Lebenszyklus gemäß einer durch den Erweiterungspunkt definierten Lebenszyklusschnittstelle, die von der Erweiterung implementiert werden muss. Ein klassischer Lebenszyklus wäre etwa die Initialisierung, die Benutzung und die Beendigung der Erweiterung.
EM-4	… definiert Installationsvorgaben.	Ein Erweiterungspunkt definiert ein Format und eine Installationsprozedur für seine Erweiterungen.
EM-5	… definiert einzuhaltende Bedingungen.	Neben den zu implementierenden Schnittstellen können einzuhaltende Bedingungen von einem Erweiterungspunkt definiert werden. Eine Bedingung könnte zum Beispiel sein, dass Erweiterungen einen bestimmten Wertebereich für Rückgabewerte einhalten müssen.

ID	Merkmal Ein Erweiterungspunkt …	Erläuterung
EM-6	… unterstützt Erweiterungsvarianten.	In der Praxis gibt es häufig mehrere technische Varianten, um eine Erweiterung für einen Erweiterungspunkt zu implementieren. Daher kann es zu einem Erweiterungspunkt mehrere technische Erweiterungsvarianten geben.

Zusammenfassend können wir nun die erfassten Merkmale in Tabelle 2.3 und Tabelle 2.4 mit dem eingeführten Schema in ein Framework-Benutzungs-Modell, zu sehen in Abbildung 2.11, einordnen.

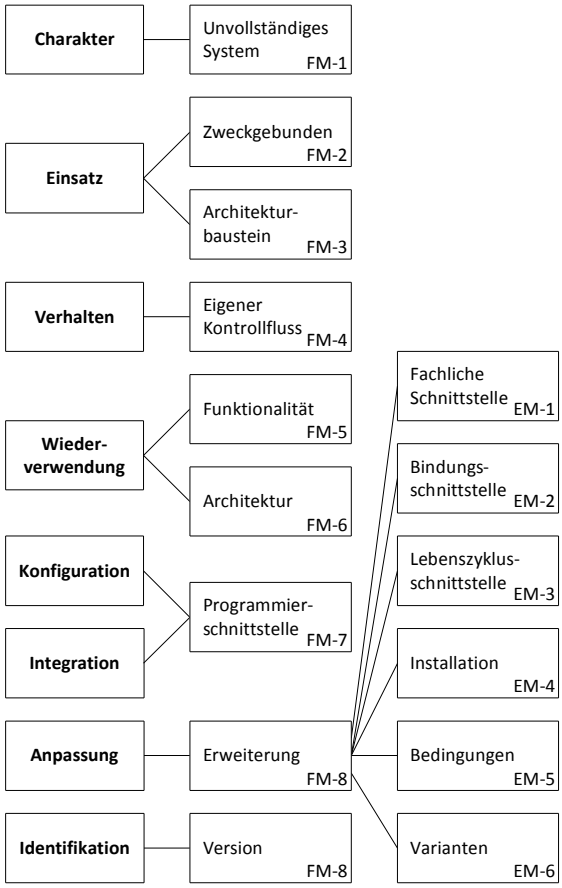


Abbildung 2.11: Framework-Benutzungs-Modell

Aus dem Framework-Benutzungs-Modell in Abbildung 2.11 wird besonders deutlich, dass Erweiterungen eine Form der Anpassung darstellen. Dieses spezielle Konzept finden wir nur bei Frameworks und deren Ausprägung lässt sich durch die untergeordneten Merkmale EM-1 bis EM-6 von Erweiterungen weiter verfeinern.

Wir wollen auf die einzelnen Merkmale nun nicht noch einmal eingehen, da diese bereits ausreichend erläutert wurden. Stattdessen wenden wir uns in den

folgenden Abschnitten verwandten Konzepten zu, die wir auf Basis unseres Schemas vergleichen wollen.

2.3.1 Verwandte Konzepte

In den vorangegangenen Abschnitten haben wir eine genaue Vorstellung gewonnen, was ein Framework ist. Nun wollen wir dieses Konzept gegenüber zwei anderen Konzepten abgrenzen, die im Zusammenhang mit Frameworks häufig genannt werden. Dies ist zum einen der Begriff einer »(Software-) Komponente« und der Begriff der »(Software-) Bibliothek«. In diesem Abschnitt werden die Begriffe gegeneinander abgegrenzt und Kriterien aufgestellt, die es erlauben, ein gegebenes Stück Software als Framework, Komponente oder Bibliothek zu identifizieren.

Software-Komponenten

Die komponentenbasierte Softwareentwicklung (engl.: Component-Based Software Engineering (CBSE)) ist ein Paradigma für die Entwicklung von Software. Der Grundgedanke ist, Software in Komponenten zu zerlegen, die getrennt voneinander entwickelt werden können und am Ende der Entwicklung als eine Software zusammenarbeiten.

Die komponentenbasierte Softwareentwicklung betrachtet eine Komponente als *Softwarekomponente*. Diese Einschränkung ist bedeutend, da durch den häufigen Gebrauch des Begriffs Komponente in unterschiedlichen Zusammenhängen auch andere Sichtweisen möglich sind. Betrachtet man Komponenten z.B. aus dem Blickwinkel einer Softwarearchitektur sind eher *Systemkomponenten* gemeint, die nicht unbedingt einer Softwarekomponente entsprechen. Ein Drucker würde in diesem Zusammenhang beispielsweise ebenfalls als Komponente bezeichnet werden. Da der überladene Gebrauch des Begriffs Komponente zu Verwirrung führt, wird in dieser Arbeit, wenn nicht explizit anders angegeben, der Begriff Komponente synonym zu Softwarekomponente gebraucht.

Im Folgenden werden einige Definitionen für den Begriff der Komponente vorgestellt und analysiert. Zunächst betrachten wir eine relativ allgemein gehaltene Definition einer Komponente, wie sie in [Brown1997] verwendet wird:

> *We characterize a component as an independently deliverable set of reusable services.*
>
> **Komponente nach [Brown1997]**

Hiernach sind Komponenten Einheiten, die unabhängig lieferbar sind, womit zum Ausdruck kommt, dass es Lieferanten und Kunden für Komponenten gibt, so dass ein Kunde Komponenten beziehen und weiterverwenden kann. Voraussetzung für ein solches Geschäft ist die unabhängige Lieferbarkeit.

Eine einzelne Komponente besteht aus einer Menge wiederverwendbarer Dienste, die in der Komponente gekapselt sind. Interessant an dieser Definition ist, dass Komponenten als Dienstmenge charakterisiert werden. Dies zeigt bereits die nahe Verwandtschaft zur Serviceorientierung, da auch Komponenten die Aufgabe erfüllen sollen, als Dienste für andere benutzbar zu sein.

Eine weitere, häufig verwendete Definition einer Komponente findet sich in [Szyperski2002]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*
>
> **Komponente nach [Szyperski2002]**

Die grundlegende Eigenschaft einer Komponente ist die Fähigkeit zur Komposition. Geregelt wird die Komposition durch vertraglich vereinbarte Schnittstellen der Komponente und durch Informationen über die benötigte Umgebung, die die Komponente voraussetzt. Diese Umgebungsabhängigkeit enthält eine ganze Reihe von Informationen darüber, wie die Komponente an sich beschaffen sein muss, um mit anderen komponiert zu werden und wie sie ausgeliefert, installiert und aktiviert wird. Die Umgebungsabhängigkeit wird durch zwei Modelle beschrieben [Szyperski2002]:

- Das *Komponentenmodell* definiert die Regeln, nach denen Komponenten komponiert werden.
- Die *Komponentenplattform* definiert die Umgebung und die Art und Weise, wie Komponenten in dieser Umgebung installiert und aktiviert werden.

Eine Komponente muss den Vorgaben beider Modelle entsprechen, um als unabhängige Komponente verwendet werden zu können. Das Komponentenmodell definiert u.a. wie eine Komponente ausdrückt, dass sie eine andere Komponente benötigt, die eine bestimmte Schnittstelle anbietet. Zusätzlich wird im Komponentenmodell definiert wie diese beiden Komponenten schließlich miteinander verbunden werden. Die Komponentenplattform hingegen definiert welche Schnittstellen einzuhalten sind, um Komponenten zu aktivieren und zu deaktivieren.

Komponentenmodell und -plattform bilden die Spezifikation einer Komponente, wobei eine Komponente dementsprechend eine Implementierung einer solchen Spezifikation darstellt. Abbildung 2.12 stellt die Beziehung zwischen Komponentenmodell, Komponentenplattform und Komponente graphisch dar, um den Sachverhalt zu verdeutlichen.

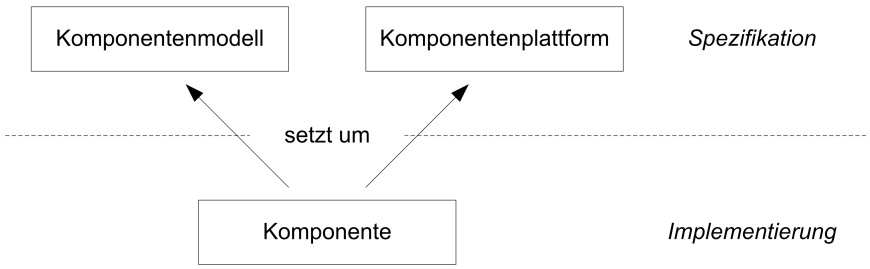


Abbildung 2.12: Beziehung zwischen Komponentenmodell, -plattform und Komponente nach [Szyperski2002]

Eine Implementierung beider Modelle ergibt die Laufzeitumgebung, in der entsprechende Komponenten ausgeführt werden. Erst eine solche Umgebung macht eine Komponente technisch benutzbar.

Trotz möglicher Abhängigkeiten zu anderen Komponenten, wird prinzipiell jede Komponente als unabhängige Einheit ausgeliefert und installiert. So können andere Anbieter diese Komponente wiederverwenden, um mit Hilfe der Komposition neue Komponenten zu erzeugen. Auch so entstehende Komponenten erhalten die Eigenschaft, als unabhängige Einheiten geliefert werden zu können. Hinter dieser Eigenschaft steckt die Vision eines Komponentenmarktes mit Anbietern und Kunden. Anbieter liefern einzelne Komponenten, die andere verwenden können, um daraus neue Komponenten zu komponieren. Ähnlich wie beim Automarkt gäbe es Zulieferer von Komponenten und Softwarehersteller, die daraus Softwaresysteme herstellen.

Eine weitere wichtige Eigenschaft, die Szyperski beschreibt, um Komponenten von Klassen und deren Objekten abzugrenzen, wird nicht in der Definition genannt. Nämlich die, dass eine Komponente im Gegensatz zu einem Objekt keinen (von außen) beobachtbaren Zustand besitzt. Dies gilt auch für den Fall, dass eine Komponente im Innern objektorientiert aufgebaut ist. Begründet liegt diese Eigenschaft darin, dass es zu einer Komponente keine Instanz gibt. Ein Objekt ist die Instanz einer Klasse, besitzt damit einen Zustand und zwei Objekte werden durch ihre unterschiedlichen Referenzen unterscheidbar. Da eine Komponente keinen beobachtbaren Zustand besitzt, ist eine Kopie einer Komponente nicht vom Original zu unterscheiden. Obwohl etwaige Objekte innerhalb einer Komponente selbstverständlich einen Zustand haben, ist dieser mit Blick auf die Komponente nicht sichtbar. Es existiert keine Verbindung zwischen einer Komponente und den Objekten innerhalb der Komponente, so dass man etwa über den qualifizierten Namen einer Komponente auf ein Objekt zugreifen könnte. Daher besitzt eine Komponente keinen beobachtbaren Zustand.

Szyperski stellt aber auch klar, dass eine Komponente nicht objektorientiert aufgebaut sein muss. Vielmehr sagt das Konzept der Komponente nichts darüber aus, nach welchem Programmiersprachenparadigma sie entwickelt und implementiert werden muss. Komponenten können konzeptionell auch dann komponiert werden, wenn die unterliegenden Implementierungstechniken verschieden sind. Diese Sichtweise ist konform zu der Auffassung, eine Komponente als so genannte *Blackbox* zu sehen, die keinen Blick ins Innere der

Komponente erlaubt. Die Implementierung und innere Details sind von außen und somit für andere Komponenten nicht sichtbar. Die alleinige Spezifikation einer Komponente über deren Schnittstellen ist ausreichend, um die Komponente verwenden zu können.

Das Gegenteil dieser Auffassung sind *Whitebox-Komponenten*, die Details ihrer Implementierung offenlegen. Eine Whitebox-Komponente kann so in ihren Details studiert werden, um ein besseres Verständnis über die Funktionsweise der Komponente an sich zu erhalten. Laut [Szyperski2002] unterscheiden manche Autoren an dieser Stelle noch Whitebox- und *Glassbox-Komponenten*: Eine Whitebox-Komponente erlaubt nach dieser Unterscheidung auch Veränderungen an der Implementierung, wohingegen eine Glassbox-Komponente nur den Blick ins Innere freigibt, aber keine Manipulationen zulässt.

Eine weitere Definition einer Komponente wird in [Heinemann2001] gegeben, die den besonderen Charakter hat, dass dieser Definition eine lange Diskussion verschiedener Experten vorangegangen ist, deren Ergebnis eine gemeinsam akzeptierte Definition war. Diese Definition besteht aus drei Teilen, um eine (Software-)Komponente korrekt einzuordnen. Betrachten wir zunächst die ersten beiden:

> *(a) A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*
>
> **Komponente nach [Heinemann2001]**

> *(b) A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.*
>
> **Komponentenmodell nach [Heinemann2001]**

Teil (a) definiert eine Komponente als Softwareelement, das zu einem Komponentenmodell konform ist. Diese Konformität ermöglicht es, die Komponente unabhängig von anderen Komponenten zur Verfügung zu stellen und ohne weitere Anpassungen, aber standardisierten Regeln folgend, mit anderen Komponenten zu verbinden.

Damit eine Komponente also mit anderen Komponenten interagieren kann, müssen zwei Dinge gegeben sein: ein Komponentenmodell und damit auch ein Kompositionsstandard.

Betrachten wir zu diesem Aspekt Teil (b) der Definition, der die Details eines Komponentenmodells darstellt. Danach definiert das Komponentenmodell präzise die Interaktions- und Kompositionsstandards. Das Komponentenmodell beantwortet also die Frage, wie Komponenten interagieren und wie sie zu neuen komponiert werden können.

Die Implementierung eines Komponentenmodells wiederum ist die (Laufzeit-)Umgebung, in der Komponenten ausgeführt werden können, die dem Komponentenmodell entsprechen.

Diese Betrachtungsweise ist im Prinzip analog zu Szyperskis Definition einer Komponente mit dem Unterschied, dass hier nicht weiter in Komponentenmodell und -plattform unterschieden wird, sondern das Komponentenmodell allein alle Aspekte spezifiziert. Abbildung 2.13 veranschaulicht diesen Zusammenhang.

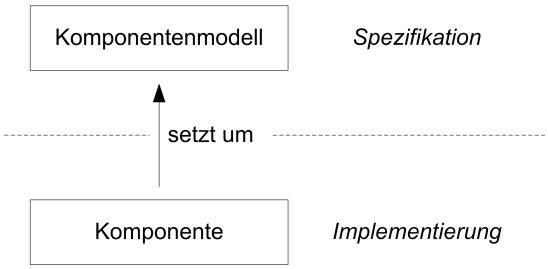


Abbildung 2.13: Beziehung zwischen Komponentenmodell und Komponente nach [Heinemann2001]

Bevor wir betrachten, was eine Komposition von Komponenten ausmacht, vervollständigen wir die Definition einer Komponente aus [Heinemann2001] mit dem dritten Teil:

> *(c) A software component infrastructure is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.*
>
> **Komponenteninfrastruktur nach [Heinemann2001]**

Teil (c) definiert eine Komponenteninfrastruktur als eine Menge von interagierenden Komponenten, die durch ihr Design sicherstellen, dass ein Softwaresystem, das diese Infrastruktur verwendet, klar spezifizierte Anforderungen erfüllt. Der Begriff „performance specification“, wie er in der Definition verwandt wird, ist dem *Performance Specification Guide* [DoD1995] des Verteidigungsministeriums der USA entnommen. Danach wird eine „performance specification“ wie folgt definiert:

> *A performance specification states requirements in terms of the required results and provides criteria for verifying compliance, but it does not state methods for achieving results. It defines the functional requirements for the product, the environment in which it must operate, and the interface and interchangeability requirements.*
>
> **Performance Specification nach [DoD1995]**

Der Begriff „performance specification“ steht also in engem Zusammenhang mit der aus der Softwaretechnik bekannten Anforderungsspezifikation, wobei eine „performance specification“ z.B. keine Unterscheidung in funktionale und nicht-funktionale Anforderungen macht. Zusammenfassend betont Teil (c) der Definition, dass eine Komponenteninfrastruktur klar definierten Anforderungen genügen muss, um sicherzustellen, dass ein Softwaresystem, das auf Basis dieser Infrastruktur konstruiert wird, diese Anforderungen ebenfalls erfüllt.

Um zu verstehen wie innerhalb einer Komponenteninfrastruktur Komponenten komponiert werden können, betrachten wir nun den Kompositionsaspekt etwas genauer. In [Heinemann2001] werden wir auf den ISO/IEC Standard zum *Reference Model of Open Distributed Processing* verwiesen, der eine Komposition in [ISO/IEC10746-2] wie folgt definiert:

> *(a) (Of objects) – A combination of two or more objects yielding a new object, at a different level of abstraction. The characteristic of the new object are determined by the objects being combined and by the way they are combined. The behaviour of a composite object is the corresponding composition of the behaviour of the component objects.*
>
> *(b) (Of behaviours) – A combination of two or more behaviours yielding a new behaviour. The characteristics of the resulting behaviour are determined by the behaviours being combined and the way they are combined.*

Komposition nach [ISO/IEC10746-2]

Um das Verständnis für den Begriff der Komposition zu vervollständigen, folgt noch die Definition für ein Objekt und eine Entität nach [ISO/IEC10746-2]:

> *Object: A model of an entity. An object is characterized by its behaviour and, dually, by its state. An object is distinct from any other object. An object is encapsulated, i.e. any change in its state can only occur as a result of an internal action or as a result of an interaction with its environment.*
>
> *Entity: Any concrete or abstract thing of interest. [...]*

Objekt und Entität nach [ISO/IEC10746-2]

In [Heinemann2001] wird ebenfalls in Blackbox- und Whitebox-Komponenten unterschieden. Blackbox-Komponenten verfolgen das Prinzip des Verbergens von (unnötigen) Informationen mit der Vorgabe, dass eine Komponente so wenig Informationen wie möglich und so viel wie nötig nach außen sichtbar macht.

Whitebox-Komponenten erlauben den Blick ins Innere und nach [Heinemann2001] auch die Möglichkeit der Adaption und Manipulation der Imple-

mentierung. Nachteil der Whitebox-Sicht ist, dass Abhängigkeiten zu einer Komponente entstehen können, die von deren Implementierung abhängt und nicht nur von deren vertraglicher Spezifikation.

Abschließend betrachten wir die *Unified Modeling Language (UML)* und die verwendete Definition einer Komponente. Die *Object Management Group (OMG)* definiert in [OMG2005] eine Komponente wie folgt:

> *A component is a self contained unit that encapsulates the state and behaviour of a number of classifiers. A component specifies a formal contract of services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces.*
> *A compononent is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces.*
>
> **Komponente nach [OMG2005]**

Die Aussagen der OMG über eine Komponente stimmen prinzipiell mit der von Szyperski überein. Die Spezifikation einer Komponente wird in einem Vertrag abgebildet, der die angebotenen und benötigten Schnittstellen der Komponente beschreibt. Mit dieser Spezifikation können Komponenten mit anderen komponiert werden und Komponenten werden austauschbar, wenn eine zweite Komponente den Vertrag der anderen Komponente ebenfalls erfüllt.

Die OMG stellt in [OMG2005] heraus, dass eine Komponente als autonome Einheit innerhalb eines Systems angesehen wird. Dieser Aspekt bezieht sich auch darauf, dass Komponenten als unabhängige Einheiten ausgeliefert und installiert werden.

Auch die OMG unterscheidet zwei Sichten auf eine Komponente, die analog zur Definition von Whitebox- und Blackbox-Komponenten ist. Die OMG bezeichnet die *externe Sicht* (external view = blackbox view) einer Komponente als die Sicht, die nur die vertraglichen Eigenschaften einer Komponente zeigt. Diese Eigenschaften können optional auch Elemente wie Protokolldefinitionen, Zustandsautomaten oder Bedingungen an die Ausführungsreihenfolge von Operationen enthalten.

Die interne Sicht (internal view = whitebox view) richtet sich auf die inneren Eigenschaften und Implementierungsdetails. Es zeigt wie das externe, vertraglich geregelte Verhalten intern realisiert wurde.

Als Modellierungssprache ist die Auffassung der UML darüber, was eine Komponente ausmacht, weiter gefasst, als es die Definitionen aus [Szyperski2002] und [Heinemann2001] sind. Die UML ermöglicht es eine Spezifikation für Komponenten anzugeben, verlangt dies aber nicht. Diese Spezifikation wird in [Szyperski2002] und [Heinemann2001] jedoch in Form eines Komponentenmodells gefordert.

Zur Veranschaulichung zeigt Abbildung 2.14 wie mit Hilfe der UML die Beziehung zwischen Komponentenmodell und Komponente dargestellt werden kann.

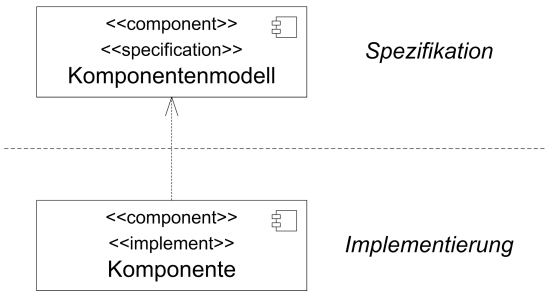


Abbildung 2.14: Beziehung zwischen Komponentenmodell und Komponente in der UML

Was aus der Definition einer Komponente nach der OMG nicht hervorgeht ist, dass die UML eine Komponente als ein Untertyp von Class beschreibt, so dass durch dieses Erbe vielfältige Modellierungsmöglichkeiten entstehen. Eine Implikation aus dieser Beziehung ist, dass eine Komponente in Form einer InstanceSpecification direkt instantiiert werden kann. Diese Möglichkeit wird insbesondere in [Szyperski2002] verboten, da dort Komponenten keinen extern beobachtbaren Zustand haben dürfen – also nicht direkt instantiierbar sind, was die UML jedoch ermöglicht.

Aus diesem Grund erlaubt es die UML anzugeben, ob eine Komponente direkt oder indirekt instantiierbar ist. Standardmäßig ist auch in der UML eine Komponente nicht direkt instantiierbar, so dass dies erst explizit angegeben werden muss. Eine indirekt instantiierte Komponente bedeutet, dass die Komponente selbst nicht die Instanz bildet, sondern dass die implementierenden Artefakte der Komponente instantiiert werden. Auf diese Weise wird die Komponente indirekt instantiiert. Diese Auffassung ist wiederum konform zur gezeigten in [Szyperski2002].

Die hier vorgestellten Definitionen ergeben ein relativ klares Bild, was eine Komponente ist und was nicht. Durch den allgemeinen Charakter des reinen Begriffs 'Komponente' wurde dieser jedoch abseits aller Definitionen vielfach gebraucht, um ganz allgemein Teile eines Software-Systems zu benennen. Manche Autoren verwenden den Begriff beispielsweise bei der Beschreibung von Softwarearchitekturen so allgemein, dass nur im Kontext klar wird, was eine Komponente gerade ist. Danach kann eine Komponente ein Objekt, eine Bibliothek, eine Datenbank oder auch ein kommerzielles Produkt sein [Bass2007]. Diese Verwendung des Begriffs führt im besten Falle zu Verwirrung, im schlechteren Fall zu einem schlechten Software-Design. Aus diesem Grund gibt es die Überlegung, dass man mindestens zwei Arten von Komponenten unterscheiden sollte [Groene2005]:

- System-Komponenten und
- Software-Komponenten

Die Unterscheidung drückt zwei unterschiedliche Sichten auf ein System aus: die Systemsicht und die Softwaresicht. Diese Unterscheidung ist analog zur Unterscheidung zwischen einer Sache und der Beschreibung einer Sache. So ist das System die eigentliche Sache und die Software ist die Beschreibung des

Systems. Wenn die Software ausgeführt wird, verhält sich das System wie es durch die Software beschrieben wurde. So wird die ausgeführte Software zum beschriebenen System.

Software-Komponenten entsprechen dabei den Komponenten, wie sie im vorangegangen Abschnitt definiert und charakterisiert wurden. System-Komponenten auf der anderen Seite sind in erster Linie informationsverarbeitende Teile eines (abstrakten) Systems, die mit anderen Teilen verbunden sind.

Als Ergebnis der vorangegangen Analyse zeigt Tabelle 2.5 eine Aufstellung der sich ergebenen Komponentenmerkmale. Um eine Komponente als solche zu klassifizieren, müssen nach Definition diese Merkmale erfüllt sein.

Tabelle 2.5: Merkmale von Software-Komponenten

Kategorie	ID	Merkmal Eine Software-Komponente ...	Erläuterung
Charakter	KM-1	... ist eine autonome, installierbare Einheit.	Eine Software-Komponente wird als autonome Einheit vertrieben und installiert.
Einsatz	KM-2	... erfüllt eine definierte Aufgabe.	Jede Software-Komponente erfüllt eine für diese Komponente spezifische Aufgabe, für die sie entworfen wurde.
Verhalten	KM-3	... verhält sich gemäß des Prinzips Eingabe, Verarbeitung, Ausgabe.	Eine Software-Komponente verarbeitet Eingabedaten und produziert im Sinne eines Dienstes Ergebnisse als Ausgabe.
Wiederverwendung	KM-4	… erlaubt Wiederverwendung der Funktionalität durch Dritte.	Eine Software-Komponente dient dazu ihre Funktionalität durch Dritte wiederverwendbar zu machen.
Konfiguration Integration Anpassung	KM-5	... ist definiert über ein Komponentenmodell.	Das Komponentenmodell definiert den Aufbau einer Software-Komponente zur Konfiguration, zur Integration mit anderen Software-Komponenten und zur Anpassung.
Identifikation	KM-6	... ist versionierbar.	Software-Komponenten werden für eine eindeutige Identifizierung versioniert.

Auch diese Liste von Merkmalen wollen wir wieder in ein Modell überführen, das wir in Abbildung 2.15 als Komponenten-Benutzungs-Modell bezeichnen. Aus dem Komponenten-Benutzungs-Modell sticht hervor, dass die drei Aspekte Konfiguration, Integration und Anpassung gemeinsam über das Komponentenmodell abgebildet werden. Für das Komponentenmodell verwenden wir ein bereits existierendes Klassifikationsschema aus [Crnkovic2011].

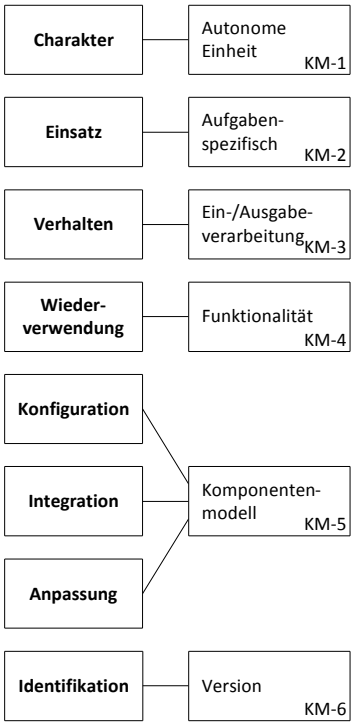


Abbildung 2.15: Komponenten-Benutzungs-Modell

In Abbildung 2.16 zeigen wir einen Auszug aus diesem Schema und gehen auf die Merkmale vertiefend ein, die den Kategorien Konfiguration, Integration und Anpassung entsprechen.

Das Komponentenmodell hat die zentrale Aufgabe, einen Standard zu definieren, gemäß dem alle Komponenten im System konstruiert sein müssen. Erst diese Standardisierung macht es möglich, dass Komponenten über festgelegte Mechanismen konfiguriert, integriert und angepasst werden können. Hierzu macht das Komponentenmodell unter anderem Vorgaben an die Schnittstellenspezifikation, so dass eine gemeinsame Basis für Schnittstellen geschaffen wird.

Komponenten verfügen über angebotene Schnittstellen, über die sie ihre Funktionalität nach außen zugänglich machen. Zusätzlich können Komponenten bestimmte Schnittstellen fordern, die von anderen Komponenten im System angeboten werden müssen. Auf diese Weise realisieren Komponentenmodelle den Aspekt der Integration von Komponenten. Um Komponenten im Sinne von erweiterter Funktionalität anzupassen, können neue Komponenten aus einzelnen Komponenten zusammengesteckt werden. Durch Komposition gemäß der Schnittstellenspezifikation entstehen durch Bindung aus einzelnen Komponenten komplexere neue Komponenten. Ein weiterer Aspekt dieser Anpassung ist die Festlegung möglicher Interaktionen zwischen Komponenten im Komponentenmodell.

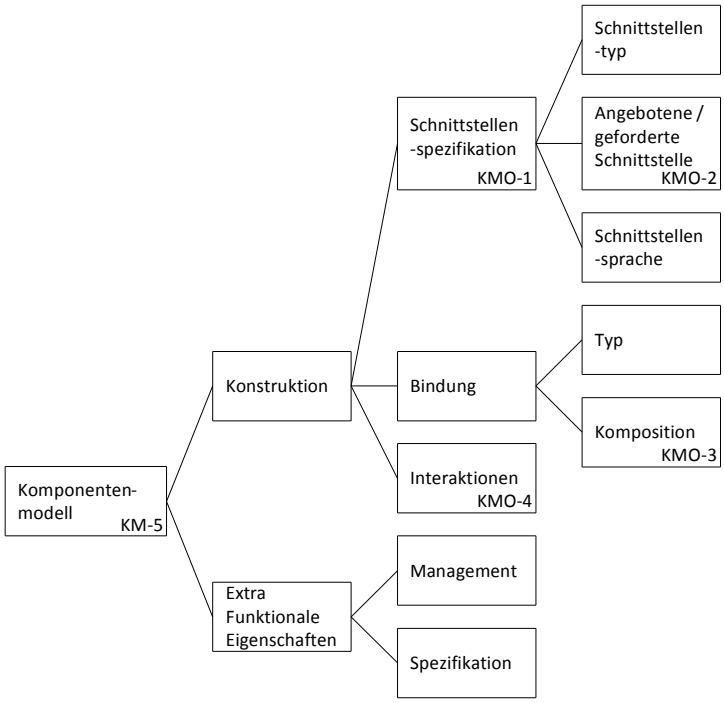


Abbildung 2.16: Komponentenmodell Klassifikationsschema nach [Crnkovic2011]

Komposition ist der zentrale Aspekt der Wiederverwendung von Komponenten. Auch hier können Probleme der Kompatibilität entstehen. Durch die Beschaffenheit einer Komponente sind diese jedoch anders gelagert, als bei den hier betrachteten Frameworks. Ein wichtiger Forschungszweig ist aufbauend auf dem Prinzip der Komposition die dynamische Adaption in komponentenbasierten Systemen, wie sie beispielsweise in [Campo2005], [Attiogbe2006] oder [Escoffier2007] betrachtet wird.

In Tabelle 2.6 fassen wir die vier Komponentenmerkmale Schnittstellenspezifikation, angebotene und geforderte Schnittstellen, Komposition sowie Interaktion für eine spätere Referenzierung noch einmal zusammen.

Tabelle 2.6: Ausgewählte Merkmale von Komponentenmodellen

Kategorie	ID	Merkmal Ein Komponentenmodell...	Erläuterung
Konfiguration	KMO-1	... gibt die Schnittstellenspezifikation vor.	Die Schnittstellen einer Komponente müssen entsprechend der Schnittstellenspezifikation des Komponentenmodells beschaffen sein, um zum Beispiel Komponenten über einen Standardmechanismus konfigurieren zu können.

Kategorie	ID	Merkmal Ein Komponentenmodell...	Erläuterung
Integration	KMO-2	... definiert angebotene und geforderte Schnittstellen.	Ein Teil der Schnittstellenspezifikation schreibt vor, wie angebotene und geforderte Schnittstellen einer Komponente beschaffen sein müssen.
Anpassung	KMO-3	... definiert die Komposition von Komponenten.	Auf Basis der Schnittstellenspezifikation mit angebotenen und geforderten Schnittstellen definiert das Komponentenmodell wie einzelne Komponenten zu neuen Komponenten komponiert werden können.
	KMO-4	... definiert die Interaktion zwischen Komponenten.	Das Komponentenmodell definiert wie Komponenten basierend auf der Schnittstellenspezifikation miteinander interagieren können.

Nachdem wir in diesem Abschnitt die Merkmale von Software-Komponenten erarbeitet haben, widmen wir uns nun den Software-Bibliotheken als ein weiteres softwaretechnisches Konzept.

Software-Bibliotheken

Eine Software-Bibliothek, die wir in Kurzform als Bibliothek bezeichnen, ist auch als Programmbibliothek bekannt und stellt eine Sammlung von Funktionen zu einem gemeinsamen Aufgabenbereich zur Verfügung. In einer Bibliothek werden diese Programmfunktionen zwecks einfacher Wiederverwendung zusammengefasst und können in dieser Form anderen Programmen zur Verfügung gestellt werden. Die Benutzung einer Bibliothek erfolgt über die zugehörige Programmierschnittstelle (engl. Application Programming Interface), genannt API. Die Funktionen werden hierbei im Sinne einer Ergebnisberechnung verwendet, so dass eine Anwendung eine Funktion aufrufen kann und dann mit dem Ergebnis dieser Funktion weiterarbeitet. Eine Bibliothek übernimmt niemals den Kontrollfluss, sondern nimmt eine passive Rolle ein.

Ähnlich wie bei Frameworks gibt es Bibliotheken für alle möglichen Aufgaben und Problemstellungen. Dies sind Bibliotheken für mathematische Berechnungen, für Betriebssystemoperationen, für Datenstrukturen oder für XML-Verarbeitung, um nur ein paar wenige zu nennen.

Aufgrund ihrer Größe im Sinne des Funktionsumfangs werden Bibliotheken häufig als die kleinen Einheiten einer Anwendung betrachtet. Fast jede Anwendung verwendet Bibliotheken, auch wenn es nur die so genannte Standardbibliothek einer Programmiersprache ist, in der zum Beispiel die Datentypen und grundlegende Datenstrukturen definiert werden. Je umfangreicher die Anwendung, umso mehr Bibliotheken werden auch typischerweise für unterschiedliche Aufgaben der Anwendung verwendet. Ebenso kann die Entwicklung einer Anwendung zur Entwicklung eigener Bibliotheken führen, die für eine geplante Wiederverwendung bereits von der Anwendung getrennt werden. Zur Verwaltung der Abhängigkeiten zwischen einer Anwendung und verwendeten

Bibliotheken werden Bibliotheken wie Frameworks und Komponenten versioniert.

Darüber hinaus können Bibliotheken Abhängigkeiten zu anderen Bibliotheken haben, die wiederum Abhängigkeiten zu weiteren haben usw. Es entsteht ein Graph aus Abhängigkeiten, der nicht immer zyklenfrei sein muss. Dadurch, dass Bibliotheken Funktionssammlungen sind, können zwei Bibliotheken sich durchaus gegenseitig verwenden.

Bibliotheken sind nicht erweiterbar, da sie als Funktionssammlung in der Regel abgeschlossene Einheiten bilden. Unterschiedliche Versionen können selbstverständlich eine andere Menge an Funktionen enthalten, aber die Funktionssammlung einer bestimmten Version ist nicht dazu gedacht, von anderen Entwicklern erweitert oder angepasst zu werden. Eine Bibliothek ist in diesem Sinne abgeschlossen.

Zusammenfassend stellen wir in Tabelle 2.7 die beschriebenen Merkmale von Software-Bibliotheken gemäß unserer acht Kategorien zusammen.

Tabelle 2.7: Merkmale von Software-Bibliotheken

Kategorie	ID	Merkmal	Erläuterung
Charakter	BibM-1	Aufgabenbereich	Die Funktionen einer Bibliothek gehören zu einem gemeinsamen Aufgaben- oder Problembereich.
Einsatz	BibM-2	Funktionssammlung	Eine Bibliothek ist eine Sammlung von Funktionen.
Verhalten	BibM-3	Ergebnisberechnung	Bibliotheken haben keinen Kontrollfluss, sondern jede Funktion berechnet per Aufruf ein Ergebnis, dass in der aufrufenden Anwendung direkt verwendet wird. Die aufrufende Anwendung gibt den Kontrollfluss dabei nicht ab.
Wiederverwendung	BibM-4	Wiederverwendbar	Die Bibliothek kann als Funktionssammlung in Anwendungen, die eine Teilmenge dieser Funktionen benötigen, wiederverwendet werden.
Konfiguration & Integration	BibM-5	Programmierschnittstelle	Eine Bibliothek wird über ihre Programmierschnittstelle (API) benutzt.
Anpassung	BibM-6	Abgeschlossenheit	Bibliotheken sind in ihrem Funktionsumfang abgeschlossen und nicht erweiterbar.
	BibM-7	Abhängigkeiten	Eine Bibliothek kann zur Realisierung ihrer Funktionen auf andere Bibliotheken zurückgreifen, die wiederum weitere Bibliotheken verwenden können usw. Bibliotheken bilden einen Graphen aus Abhängigkeitsbeziehungen, der nicht frei von Zyklen sein muss.
Identifikation	BibM-8	Version	Bibliotheken werden für eine eindeutige Identifizierung versioniert.

Diese Merkmale von Software-Bibliotheken wollen wir wiederum in ein Benutzungsmodell überführen. Wir zeigen dieses Bibliotheken-Benutzungs-Modell in Abbildung 2.17.

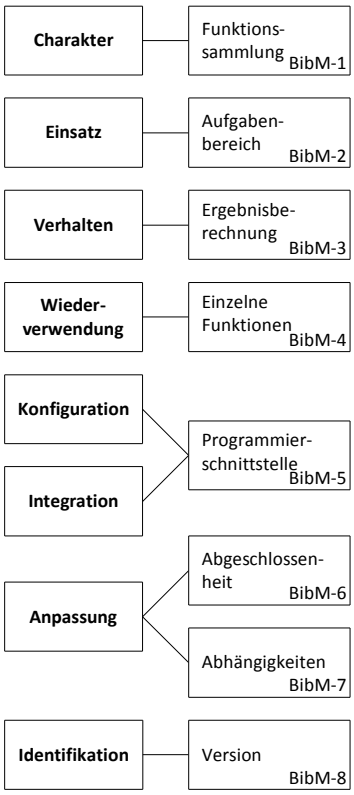


Abbildung 2.17: Bibliotheken-Benutzungs-Modell

Nachdem wir die Benutzungsmodelle für Software-Frameworks, -Komponenten und -Bibliotheken auf Basis ihrer Merkmale aufgestellt haben, können wir diese drei Konzepte der Softwaretechnik einander gegenüberstellen.

2.3.2 Gegenüberstellung

Um Frameworks anhand ihrer Merkmale von verwandten Konzepten abgrenzen zu können, stellen wir die erarbeiteten Merkmale von Frameworks, Komponenten und Bibliotheken in Abbildung 2.18 einander gegenüber.

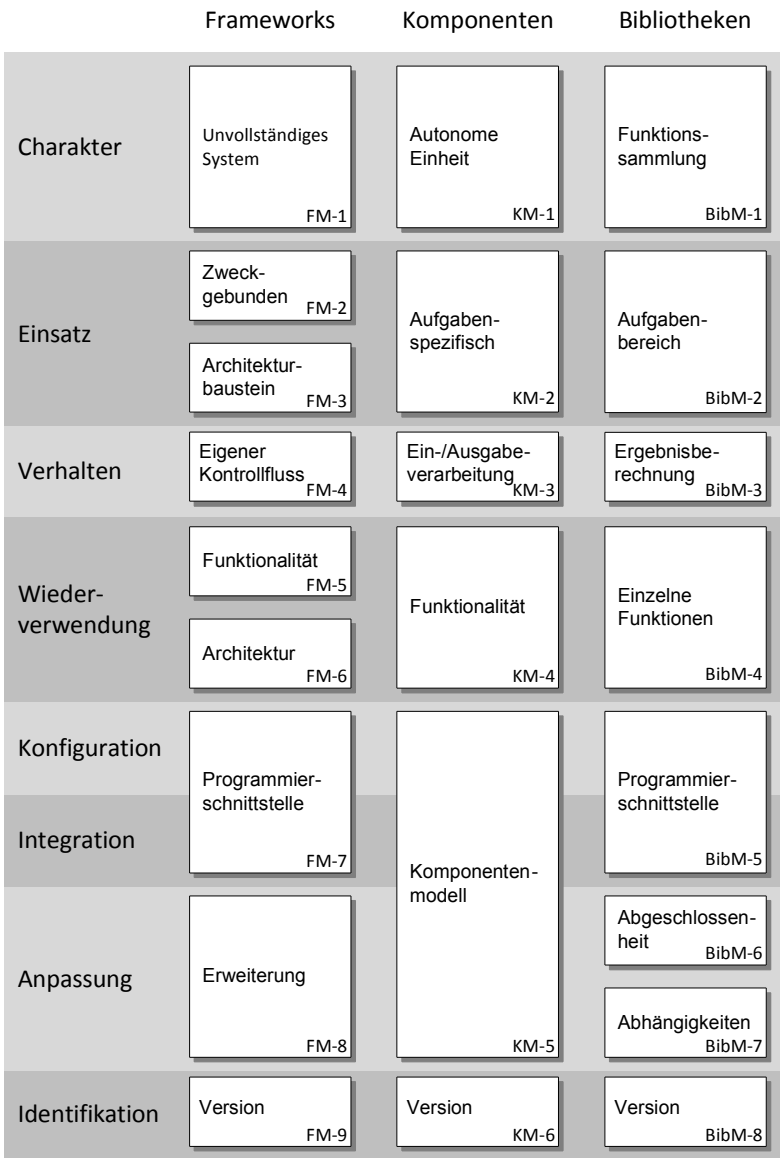


Abbildung 2.18: Vergleich der Benutzungsmodelle

Die drei betrachteten Konzepte unterscheiden sich zunächst grundlegend in ihrem Charakter. Ein Framework als unvollständiges System ist keine autonome Einheit wie eine Software-Komponente, die man ohne Anpassung benutzen kann. Diese beiden Konzepte unterscheiden sich in dieser Hinsicht auch von Bibliotheken, da diese eine ganze Sammlung von Funktionen anbieten und dabei weder erweiterbar sind noch als autonome Einheiten agieren können. Sie bilden vielmehr die Hilfsfunktionen in komplexeren Systemen ab.

Beim Einsatz sind alle drei Konzepte ähnlich gelagert, denn logischerweise sind alle für bestimmte Aufgaben konzipiert worden. Im Punkt Wiederverwendung sticht das Framework hervor, weil es neben der Funktionalität auch eine Wiederverwendung der Architektur ermöglicht.

Im Punkt Verhalten unterscheiden sich alle drei Konzepte recht klar voneinander. Ein Framework übergibt den Kontrollfluss an seine Erweiterungen, die

dann den Kontrollfluss nach Beendigung ihrer Routinen wieder zurück ans Framework geben. Bei Software-Komponenten verwendet man häufig ein Ein-/Ausgabekontrollfluss, der es erlaubt, dass Komponenten wie Filter arbeiten können. So lassen sich Komponenten durch einfache Hintereinanderschaltung komponieren. Eine Bibliothek hingegen bietet Funktionen an, die ein konkretes Ergebnis berechnen. Hier wird der Kontrollfluss von der aufrufenden Instanz nicht direkt abgegeben, sondern die Instanz ruft eine Funktion der Bibliothek auf und verarbeitet selbst das Ergebnis.

Im Bereich Konfiguration und Integration verfügen alle drei Konzepte über entsprechende Programmierschnittstellen. Diese erfüllt in allen drei Konzepten den selben Zweck. Diese Programmierschnittstelle ist jedoch von Mechanismen zu unterscheiden, die es erlauben Anpassungen vorzunehmen. Auf technischer Ebene erkennt man den Unterschied häufig nicht direkt, da Schnittstellen nicht so explizit gruppiert werden.

Anpassungen realisieren alle drei Konzepte unterschiedlich. Das Framework hat hierfür das zentrale Konzept der Erweiterungspunkte. Software-Komponenten definieren ein Komponentenmodell, das wiederum die Möglichkeiten der Komposition festlegt. Bibliotheken erlauben in der Regel gar keine Anpassungen, sondern können nur so verwendet werden, wie sie definiert wurden. Die Unterschiede verschwimmen, sobald diese Konzepte nicht in Reinform vorkommen, sondern kombiniert werden. So können Software-Komponenten auch wie Frameworks aufgebaut sein und dabei Funktionen wie eine Bibliothek anbieten. Es gibt auch den Fall, dass die Erweiterungen für ein Framework als Software-Komponenten definiert werden, so dass beide Konzepte in Kombination angewandt werden. In der Regel wird zum Beispiel eine Komponentenplattform als Framework definiert und implementiert. Diese Vermischung in der Praxis erschwert es häufig eine gegebene Software eindeutig einzuordnen. Häufig wird man Anteile aller drei Konzepte vorfinden.

Als Identifikationsmerkmal verwenden alle drei Konzepte ein Versionierungsverfahren, das es erlaubt, unterschiedliche Versionen eines Framework, einer Software-Komponente oder einer Bibliothek zu unterscheiden und eine evolutionäre Abfolge ablesen zu können. Die konkrete Bildung von Versionsnummern kann dabei unterschiedlichen Richtlinien unterliegen und wird nicht weiter eingegrenzt.

Zusammenfassend halten wir fest, dass sich Frameworks in einigen Merkmalen deutlich von den anderen softwaretechnischen Konzepten unterscheiden. Das Vergleichsschema aus acht Kategorien hilft diese Konzepte miteinander zu vergleichen, um eine gegebene Software entsprechend einordnen zu können. Wie bereits erwähnt finden wir in der Praxis nicht nur Reinformen der erläuterten Konzepte, sondern Software-Ingenieure bedienen sich je nach Bedarf bei allen drei Konzepten. Dieses Vorgehen ist sinnvoll, da in komplexen Software-Systemen, wie wir sie mit den Software-Stacks vorgestellt haben, alle Konzepte im Paralleleinsatz notwendig sind, um das System zu strukturieren. Die Autoren Schmidt und Buschmann haben in [Schmidt2003] auf diese Synergieeffekte bereits hingewiesen. Diese Vermischung der Konzepte macht es häufig schwierig eine eindeutige Zuordnung zu treffen.

2.4 Kompatibilität

Zur Beurteilung der Kompatibilität einer Anwendung zu einem Framework wollen wir in diesem Abschnitt das Kompatibilitätsproblem einführen und für diese Arbeit präzisieren. Eines der ältesten Probleme in der Softwaretechnik ist es, die Interoperabilität von Software und deren Komponenten zu gewährleisten. Die Interoperabilität ist dabei ein Teil des weiter gefassten Konzepts der Kompatibilität von Software. Neben der Interoperabilität gibt es weitere Kompatibilitätsaspekte wie die Austauschbarkeit von Programmen, die Fähigkeiten Programme auf unterschiedlicher Hardware ausführen zu können, Kompatibilität in Datenformaten etc. [Gosden1968].

In dieser Arbeit betrachten wir im Rahmen der Problemstellung die Interoperabilität von Frameworks, wenn ein Framework durch eine neuere Version ersetzt wird. Allgemein wollen wir hierbei von der Kompatibilität von Frameworks sprechen und um diese zu beurteilen, betrachten wir zunächst einige grundsätzlichen Eigenschaften der Frage nach Kompatibilität.

Durch die durchaus große Anzahl an verwendeten Frameworks, die für die Entwicklung eines betrieblichen Informationssystems zum Einsatz kommen, entsteht die Fragestellung der Kompatibilität der interagierenden Frameworks schon in frühen Entwicklungsphasen. Dabei will man die Frage in der Regel nicht nur für zwei, sondern für alle interagierenden Frameworks beantworten. Unter Beachtung der Aufgabenstellung in Abschnitt 1.2 fokussieren wir folgende Fragestellung:

> *Wenn eine Anwendung ein Framework in einer bestimmten Version verwendet und nun eine neue Version des Frameworks verwendet werden soll, ist die Anwendung auch kompatibel zur neuen Version?*

Wir betrachten hierzu die Abwärtskompatibilität einer neuen Framework-Version in Bezug auf die alte Version. Die neue Version ist abwärtskompatibel, wenn das extern beobachtbare Verhalten der alten Version auch in der neuen Version gegeben ist (vergl. [Gosden1968]).

Zur Beantwortung der Kompatibilitätsfrage gehen wir von der Annahme aus, dass Kompatibilität gegeben sei. Nun versuchen wir auf Basis der verfügbaren Informationen über die vorliegenden Framework-Versionen Kriterien zu entdecken, die der Kompatibilität entgegenstehen. Damit ist jede Aussage über die Kompatibilität nur so gut, wie die vorliegenden Informationen. Wenn auf Basis der vorliegenden Informationen kein Kriterium ermittelt werden kann, dass der Kompatibilität entgegensteht, gehen wir davon aus, dass die betrachteten Systeme kompatibel sind. Bei der Suche nach Kriterien gehen wir optimistisch vor, so dass jedes Kriterium, das einen Hinweis auf eine mögliche Verletzung der Kompatibilität gibt, zu der Antwort führt, das keine Kompatibilität gegeben ist. Dies ist sinnvoll, da nur in dem Fall die Kompatibilität bestätigt werden darf, in dem es auf Basis der vorliegenden Informationen keine Anzeichen gibt, dass die Kompatibilität verletzt sein könnte. Die Bestätigung einer vorliegenden Kompatibilität ist somit pessimistisch geprägt.

Die ideale Antwort auf die Kompatibilitätsfrage wäre ein uneingeschränktes »Ja«. Diese Antwort lässt keine weiteren Fragen offen und erfordert keine Anpassungsarbeiten in der Entwicklung. Ist die Antwort jedoch ein »Nein«, stellt sich im Anschluss direkt die Frage nach dem »Warum«. Die Warum-Frage hat in der Praxis den Hintergrund, dass man Rückschlüsse auf etwaige Anpassungsarbeiten ziehen möchte, die man durchführen müsste, um das Kompatibilitätsproblem zu lösen. Wenn keine Kompatibilität vorliegt, will man zumindest eine Aussage über den Grad der Kompatibilität erfahren. Der Grad kann wiederum Rückschlüsse auf den Anpassungsaufwand zulassen. Zur Bestimmung des Grades helfen die ermittelten Kriterien weiter, die der Kompatibilität entgegenstehen.

Leider jedoch ist die Antwort auf die Kompatibilitätsfrage in der Praxis ein abwägendes »vielleicht«, ein optimistisches »sollte« oder es kann gar keine Aussage getroffen werden. Der Grund hierfür ist meist ein Informationsmangel, so dass die vorliegenden Informationen nicht ausreichen, um den Sachverhalt zu klären. In diesem Fall vertraut ein verantwortlicher Projektleiter oftmals auf das Bauchgefühl seiner Entwickler und die Kompatibilitätsfrage wird durch eine Migration auf Probe beantwortet, wodurch Aufwände entstehen, die man eigentlich vermeiden will. Der Projektleiter muss daher über die notwendigen Informationen verfügen, so dass es nur zwei mögliche Antworten auf die Kompatibilitätsfrage gibt:

- *Ja*
- *Nein, aber mit einem Aufwand von X kann die Kompatibilität erreicht werden.*

Da die Abschätzung von Entwicklungsaufwänden ein schwieriges Problem darstellt, in das viele unternehmens- und projektspezifische Faktoren einfließen, kann der Aufwand nicht direkt angegeben werden. Für einen Projektleiter wäre es jedoch eine Hilfe, eine Liste mit Kriterien zu erhalten, die bearbeitet werden müssten, um die Kompatibilität zu erreichen. Die Abschätzung des Aufwandes könnte sich auf diese Liste stützen. Eine solche Aufwandschätzung ist jedoch nicht Teil dieser Arbeit. Stattdessen wollen wir mit den ermittelten Kriterien eine Vorarbeit für eine mögliche Aufwandschätzung leisten. Daher drücken wir die Antwort »Nein« auf die Kompatibilitätsfrage folgendermaßen aus:

- *Nein, da folgende Kriterien der Kompatibilität entgegenstehen.*

Nach diesen allgemeinen Überlegungen zum Begriff der Kompatibilität, wollen wir nun formal spezifizieren, was wir unter dem Begriff der Kompatibilität verstehen. Mit Hilfe dieser Definition können wir dann die Kompatibilität einer Anwendung zu einem Framework konkretisieren und auf die Problemstellung dieser Arbeit anwenden.

Wenn man die Frage nach Kompatibilität stellt, wird für zwei gegebene Entitäten e_1 und e_2 allgemein folgendes formuliert:

$$\text{Ist } e_1 \text{ kompatibel zu } e_2 ?$$

Bei genauerer Betrachtung ist diese Fragestellung unvollständig, denn es stellt sich die Frage unter welchem Gesichtspunkt die Kompatibilität zu betrachten ist, da Kompatibilität immer nur in Bezug zu einer Eigenschaft beurteilt werden kann. Der Fragestellung fehlt somit die Eigenschaft p, unter der die Kompatibilität betrachtet werden soll. Die Frage der Kompatibilität muss demnach vervollständigt werden zu:

Ist e_1 bezüglich der Eigenschaft p kompatibel zu e_2?

Wie gesehen ist die Antwort auf eine solche Frage idealerweise ein »Ja« oder ein einschränkendes »Nein« der Form: e_1 ist nicht kompatibel zu e_2, denn es gibt Kriterien, die dem entgegenstehen. Welche Kriterien dies im Einzelnen sind, ist von Fall zu Fall unterschiedlich, aber generell gilt die Aussage, dass auf Basis der vorliegenden Informationen die Kriterien nicht entkräftet werden können, so dass weitere Prüfungen notwendig sind. Diese die Kompatibilitätsaussage einschränkenden Kriterien werden als *Kompatibilitätseinschränkungen* bezeichnet.

Es kann vorkommen, dass sich eine Kompatibilitätseinschränkung nach weiteren Prüfungen als nicht einschränkend herausstellt. Dieses Phänomen bezeichnet man auch als falsch-positive Kompatibilitätseinschränkungen, die sich nach Prüfung durch einen Menschen, der gegebenenfalls mehr Informationen zur Verfügung hat und diese entsprechend Interpretieren kann, als haltlose Einschränkungen herausstellen. Können alle Einschränkungen durch weitere Prüfungen entkräftet werden, so ist wieder Kompatibilität gegeben.

Eine Kompatibilitätsfunktion $c()$ bildet die Frage nach der Kompatibilität zweier Entitäten e_1 und e_2 bezüglich einer Eigenschaft p auf eine Menge Δ bestehend aus Kompatibilitätseinschränkungen ab, die wir als λ_i mit $i \in \mathbb{N}$ bezeichnen. Jede Einschränkung definiert ein Kriterium, das der Kompatibilität entgegensteht. Ist die Menge der Einschränkungen leer, ist Kompatibilität gegeben. Wir halten fest:

> *Sei Δ die Menge der Kompatibilitätseinschränkungen, E die Menge der Entitäten und P die Menge der Eigenschaften, dann ist die Kompatibilitätsfunktion gegeben durch:*
>
> $$c : E \times E \times P \rightarrow \Delta$$

Die Kompatibilitätsfunktion macht keine Annahmen oder Aussagen darüber, ob die Kompatibilitätseinschränkungen durch spätere Prüfungen entkräftet werden könnten. Die Aussage der Kompatibilitätsfunktion wird auf Basis der vorliegenden Informationen gebildet und ist somit nur für diese Informationen gültig.

Auf Basis der Kompatibilitätsfunktion können wir nun unsere Auffassung über die Kompatibilität zweier Entitäten präzisieren.

Zwei Entitäten $e_1 \in E$ *und* $e_2 \in E$ *sind bezüglich einer Eigenschaft* $p \in P$ *zueinander kompatibel, wenn gilt:*

$$c(e_1, e_2, p) = \emptyset$$

Wir wollen dies an einem Beispiel verdeutlichen. Wir betrachten die Kompatibilität zwischen der Entität einer Anwendung (e_1) und einem Framework (e_2). Die zu betrachtende Eigenschaft (p) sei die Erweiterungseigenschaft des Frameworks. Hiermit ist gemeint, dass wir die Kompatibilität dahingehend prüfen, ob die Erweiterungen innerhalb der Anwendung kompatibel zum gegebenen Framework sind. Die zu erfolgende Prüfung wird definiert durch die festzulegende Kompatibilitätsfunktion c . Teil der Prüfung könnte zum Beispiel sein, die korrekte Implementierung geforderter Methodensignaturen innerhalb der Anwendung sicherzustellen. Wird eine abweichende Methodensignatur in der Anwendung entdeckt, so stellt dies eine Kompatibilitätseinschränkung λ dar. Am Ende der Prüfung hat man eine Menge an Methodensignaturen identifiziert, die nicht kompatibel zum Framework sind.

Die Kompatibilität zwischen zwei Entitäten wird immer in Bezug auf eine Eigenschaft bewertet. Eigenschaften von Softwaresystemen können auf zwei Ebenen betrachtet werden. Je nachdem welche Ebene man wählt, werden bestimmte Informationen zur Beurteilung der Kompatibilität benötigt und die Beantwortung der Kompatibilitätsfrage erweist sich insbesondere für einen Rechner als unterschiedlich schwierig. Wir unterscheiden folgende zwei Ebenen:

- Semantische Ebene
- Syntaktische Ebene

Neben diesen zwei Ebenen können wir weiter die eingesetzte Prüftechnik differenzieren. Für Softwaresysteme gibt es grundsätzlich zwei Formen der Prüftechnik: Zum einen die Technik der statischen Analyse einer Software und zum anderen die Technik eine Software während ihrer Ausführung zu untersuchen. Letztere Technik bezeichnet man als dynamische Analyse einer Software.

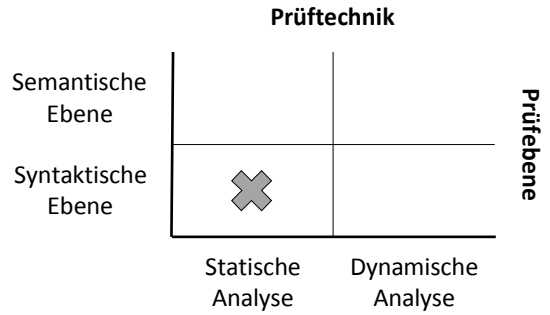


Abbildung 2.19: Prüfebenen und -techniken für Softwaresysteme

Aus der Kombination der Prüfebene mit der Prüftechnik ergeben sich in Abbildung 2.19 vier mögliche Kategorien zur Kompatibilitätsprüfung von

Softwaresystemen. In dieser Arbeit beschränken wir uns auf eine Kompatibilitätsprüfung auf syntaktischer Ebene mit der Technik einer statischen Analyse. Das Kreuz in Abbildung 2.19 markiert die eingesetzte Kombination.

Die in dieser Arbeit angewandte Analyse wird gemäß unserer Lösungsstrategie nicht auf dem Quellcode des Frameworks ausgeführt, sondern auf der Framework-Beschreibung. Wir beurteilen syntaktische Unterschiede zwischen zwei Framework-Beschreibungen und ziehen daraus Schlüsse auf mögliche Inkompatibilitäten. Der höhere Abstraktionsgrad der Beschreibungen gegenüber einem Vergleich auf Quellcodeebene erlaubt es uns, aus syntaktischen Vergleichen Rückschlüsse auf die Semantik zu ziehen. Auf Ebene der Beschreibungen können wir zum Beispiel durch syntaktischen Vergleich feststellen, dass ein Erweiterungspunkt gelöscht wurde. Diese syntaktische Veränderung in der Beschreibung entspricht einer semantischen Veränderungen des Frameworks. Die Bewertung der Kompatibilität auf Basis der Beschreibungen ist somit aussagekräftiger als eine Analyse auf Codeebene.

Neben der Unterscheidung in Prüfebene und -technik unterscheidet man die Eigenschaften eines Softwaresystems in funktionale und nicht-funktionale Eigenschaften. Eine funktionale Eigenschaft legt, wie der Name bereits sagt, eine ausführbare Funktionalität eines Softwaresystems fest. Die Eigenschaften Dateien speichern zu können ist zum Beispiel eine funktionale Eigenschaft. Eine nicht-funktionale Eigenschaft drückt hingegen eine Eigenschaft des Softwaresystems selbst aus. Ein Beispiel wäre, dass die Antwortzeit auf eine Anfrage an ein Softwaresystem immer weniger als zehn Millisekunden beträgt. Wir wollen funktionale und nicht-funktionale Eigenschaften getrennt betrachten. Für beide Arten betrachten wir die Kompatibilität jedoch jeweils auf syntaktischer Ebene mit einer statischen Analyse.

Auf syntaktischer Ebene betrachtet man Sprachkonstrukte, die gemäß einer gegebenen Grammatik eingehalten werden müssen. Ein Übersetzer (Compiler) überprüft unter anderem die syntaktische Korrektheit eines Programms durch statische Analyse während der Übersetzung. In Bezug auf die Benutzung von Frameworks können mit ähnlichen Analysen zuverlässig syntaktische Inkompatibilitäten aufgedeckt werden.

Eine syntaktische Inkompatibilität entsteht zum Beispiel, wenn sich von einer Framework-Version zur anderen die Methodensignatur einer Methode geändert hat, so dass eine Anwendung diese Syntaxänderung berücksichtigen muss. Solch ein syntaktischer Unterschied kann zu einem Bruch der Kompatibilität führen, da die Methode nicht mehr aufgerufen werden kann.

Auf syntaktischer Ebene mit statischer Analyse können wir jedoch nicht entscheiden, ob die Veränderung der Methodensignatur zur Laufzeit überhaupt Auswirkungen gehabt hätte, denn eventuell wäre diese Methode niemals ausgeführt worden. Für eine solche Analyse müssten wir die Analyse auf der semantischen Ebene mit de Prüftechnik der dynamischen Analyse erweitern.

Eine Kompatibilitätsanalyse auf semantischer Ebene ist naturgemäß aussagekräftiger als eine reine Analyse auf syntaktischer Ebene. Eine Analyse auf semantischer Ebene mit dynamischer Prüftechnik wäre zum Beispiel in der Lage zu entscheiden, ob eine Methode bei Ausführung des Frameworks überhaupt aufgerufen wird. Die Bestimmung der tatsächlichen Methodenaufrufe

und damit welcher Programmcode wirklich ausgeführt wird, ist im Allgemeinen ein zu komplexes Problem, als dass es effizient von einem Rechner gelöst werden könnte. Die Explosion der Anzahl möglicher Programmzustände macht eine Analyse von Softwaresystemen realistischer Größe, wie wir sie hier betrachten wollen, unmöglich.

Im Bereich nicht-funktionaler Eigenschaften können qualitative Einschränkungen angeführt werden, die eine Erweiterung eines Frameworks erfüllen muss. Derartige Einschränkungen können sich zum Beispiel auf bestimmte Aspekte der Performanz beziehen oder auch auf die Qualität bereitgestellter Daten. Ein besonderer Aspekt betrieblicher Informationssysteme sind nicht-funktionale Einschränkungen wie das Vertrauen in eine Komponente, Zusicherung der Verfügbarkeit oder auch monetäre Aspekte wie Benutzungsentgelte. Je nach Art des Frameworks und möglicher Erweiterungen können derartige nicht-funktionale Einschränkungen zum Tragen kommen. Mögliche Inkompatibilität nicht-funktionaler Art werden in dieser Arbeit syntaktisch beurteilt. Syntaktisch kann mit einer statischen Analyse geprüft werden, ob in einer neueren Framework-Version ein nicht-funktionaler Aspekt verändert wurde. Die Semantik dieser Veränderung kann jedoch nicht beurteilt werden. Es wird lediglich festgestellt, dass etwas geändert wurde. Der Framework-Benutzer muss die Auswirkungen einer Veränderung interpretieren, wenn sich von einer Framework-Version zur anderen eine nicht-funktionale Einschränkung ändert.

Die Überlegungen zu möglichen Veränderungen am Framework beziehen sich immer auf die Fragestellung der Kompatibilität. Wir wollen die Kompatibilität einer Anwendung zu einem Framework näher betrachten. Zentrale Eigenschaft der Kompatibilität einer Anwendung zu einem Framework ist die korrekte Verwendung der Erweiterungspunkte des Frameworks gemäß der Framework-Beschreibung. Wir legen die Kompatibilität einer Anwendung zu einem Framework bezüglich der Erweiterungseigenschaft wie folgt fest.

> *Eine Anwendung A ist bezüglich der Erweiterungseigenschaft kompatibel zu einem Framework F, wenn die Erweiterungen in A gemäß der Framework-Beschreibung von F implementiert sind.*

Ausgehend von dieser Festlegung können wir nun ableiten, welche Eigenschaft bezüglich der Kompatibilitätsfunktion zwischen einer kompatiblen Anwendung und einem Framework erfüllt sein muss.

> *Sei c eine Kompatibilitätsfunktion, A eine Anwendung, F_1 ein Framework in Version 1 und EW die Erweiterungseigenschaft.*
>
> *Wenn gilt $c(A, F_1, EW) = \emptyset$, dann ist A bezüglich der Erweiterungseigenschaft kompatibel zu F_1.*

Mit Hilfe diesen Festlegungen haben wir die Voraussetzungen geschaffen, um unseren Ansatz einer automatischen Kompatibilitätsprüfung in Kapitel 3 zu

formulieren. Zunächst grenzen wir in Abschnitt 2.5 jedoch das gestellte Problem von verwandten Arbeiten ab.

2.5 Abgrenzung zu verwandten Arbeiten

Um eine klare Abgrenzung zu verwandten Arbeiten herzustellen, beziehen wir uns auf Ansätze, die sich explizit mit der Evolution von Frameworks und dem entstehenden Problem der Kompatibilität existierender Erweiterungen befassen.

Eine eindeutige Zuordnung mancher Arbeiten in diese Kategorie ist jedoch nicht immer möglich. In der Literatur werden Konzepte, wie API, Komponenten und Frameworks, häufig gemischt oder Autoren definieren nicht, welches Konzept sie genau betrachten. Dies mag ein Grund dafür sein, dass wir in der Literatur nur wenige Ansätze finden, die sich explizit mit der Problematik evolvierender Frameworks beschäftigen.

Cortés beschreibt die Evolution von Frameworks als eine Mischung aus Refaktorisierungen und Erweiterungen [Cortes2006a]. In der Tat sind viele Veränderungen an Frameworks auf Refaktorisierungsmaßnahmen zurückzuführen [Dig2005]. Diese Auffassung der Evolution eines Frameworks ist sehr quellcodezentriert und betrachtet Veränderungen auf Ebene der Erweiterungsschnittstellen (siehe Abbildung 2.10) eines Frameworks, die in der Literatur allgemein als API bezeichnet wird. Die Autoren beschreiben in [Cortes2006] [Cortes2006a] die Evolution von Frameworks als eine Menge von Refaktorisierungsschritten, die man regelbasiert erfassen kann. Hat man Kenntnis über die Refaktorisierungsschritte, kann man die die benötigten Veränderungen auch an Erweiterungen vornehmen.

In [Savga2007] stellen die Autoren ein Verfahren vor, um automatisch Adapter für Erweiterungen von sich weiterentwickelnden Frameworks zu generieren. Die Idee ist, dass für eine neue Framework-Version ein Adapter generiert werden kann, der eine bestehende Erweiterung wieder nutzbar macht. Die Autoren beziehen sich dabei auf Kompatibilität auf binärer Ebene. Leider gehen die Autoren in ihrer Arbeit nicht darauf ein, wie man die Veränderungen zwischen zwei Framework-Versionen bestimmt. Stattdessen setzen sie voraus, dass diese Information bereits vorliegt.

Zur Bestimmung der Unterschiede protokollieren manche Ansätze die Veränderungen am Framework während Entwickler es verändern. In [Henkel2005] schlagen die Autoren diesen Weg ein und stellen ein Aufzeichnungswerkzeug vor, mit dem man die Änderungen am Framework protokolliert. Darüber hinaus lassen sich die Aufzeichnung wiedergeben, um benötigte Änderungen an Erweiterungen vorzunehmen.

In [Meng2012] versuchen die Autoren die Evolution eines Frameworks durch Analyse der Kommentare in der Revisionshistorie nachzuvollziehen, die während der Entwicklung des Frameworks entsteht. Hierzu setzen sie Techniken der natürlichen Sprachverarbeitung ein. In die gleiche Kategorie fällt der »SemDiff« Ansatz [Dagenais2008][Dagenais2011], der ebenfalls versucht aus der Framework-Historie die gewünschten Informationen abzuleiten.

Die Autoren in [Meng2012] vergleichen ihren Ansatz mit dem »Aura« Ansatz in [Wu2010]. In Aura kombinieren die Autoren die Analyse von Codeabhängigkeiten mit der textuellen Analyse von Methoden-Signaturen [Wu2010]. Ziel ist es, bei gelöschten Methoden diejenige Methoden automatisch zu erkennen, die stattdessen benutzt werden sollten.

In [Schaefer2008] versuchen die Autoren die Veränderungen durch Analyse von bereits angepassten Anwendungen zu bestimmen. Durch Analyse kompatibler Anwendungen können Rückschlüsse auf Veränderungen gezogen werden.

Die zitierten Ansätze stehen stellvertretend für eine Reihe von Arbeiten, die vorgeben sich mit der Evolution von Frameworks zu beschäftigen. Bei genauerer Betrachtung analysieren jedoch alle Ansätze die Veränderungen nur auf Methoden-Ebene. Die Ansätze sprechen von der API eines Frameworks ohne dies genauer zu spezifizieren. Veränderungen auf abstrakterer Ebene unter Berücksichtigung der Merkmale von Frameworks, wie wir sie definiert haben, werden nicht betrachtet. Aufgrund ihrer Ausrichtung auf APIs unterscheiden sich diese Ansätze grundlegend von der hier verfolgten Lösungsstrategie. Insbesondere fehlt diesen Ansätzen die geforderte Abstraktionsebene.

Des weiteren betrachtet kein Ansatz die Problematik, dass neben der Bestimmung der Veränderungen am Framework auch die Teile des Frameworks bestimmt werden müssen, die von einer Anwendung benutzt werden. Kein Ansatz betrachtet die Situation, dass sinnvollerweise nicht alle Veränderungen am Framework zu betrachten sind. Neben der Abstraktion unterscheidet sich unser Ansatz auch in dieser Hinsicht von den publizierten Ansätzen.

2.6 Zusammenfassung

In diesem Kapitel haben wir mit Hilfe von Beispielen und Definitionen die grundlegenden Merkmale von Frameworks und der Kompatibilität von Anwendungen zu Frameworks herausgearbeitet. Das zentrale Konzept von Frameworks sind ihre Erweiterungspunkte, die es insbesondere im Sinne der Kompatibilität einer Anwendung zu betrachten gilt. Wir haben in Abgrenzung zu verwandten Arbeiten gesehen, dass die in der Literatur verfolgten Ansätze das gestellte Problem nur auf Ebene der API eines Frameworks betrachten. Die in diesem Kapitel geschaffenen Grundlagen ermöglichen es uns nun, unseren Ansatz zur automatischen Kompatibilitätsprüfung in Kapitel 3 zu formulieren.

3

Ansatz zur automatischen Kompatibilitätsprüfung

Selling computers to some of these dot-coms
is like giving a gun to a five-year-old.

-- Sun Microsystem manager

Nachdem wir in den vorangegangenen Kapiteln die Grundlagen zu Frameworks und ihrer Kompatibilität zu Anwendungen erörtert haben, konkretisieren wir nun die in Abschnitt 1.2 entworfene Lösungsstrategie. Nach unserer Festlegung hat eine Anwendung, die zu einem Framework kompatibel ist, die Eigenschaft, dass alle implementierten Erweiterungen gemäß der Framework-Beschreibung des Frameworks implementiert wurden. Für unsere Problemstellung ist gegeben, dass eine Anwendung kompatibel zu einem Framework einer bestimmten Version ist. Wir können nun unsere Aufgabenstellung aus Abschnitt 1.1 in eine *formale Aufgabenstellung* überführen.

Gegeben sei eine Anwendung A, ein Framework F_1 in Version 1 und die Erweiterungseigenschaft EW. Es gelte für eine Kompatibilitätsfunktion $c(A, F_1, EW) = \emptyset$.

Gesucht ist eine Kompatibilitätsfunktion c, die für eine Version 2 des Frameworks F_1, genannt F_2, die Kompatibilitätseinschränkungen $\Delta = c(A, F_2, EW)$ bestimmt.

Formale Aufgabenstellung

Das Ziel ist die Bestimmung von Kompatibilitätseinschränkungen bei Aktualisierung des Frameworks. Die Kompatibilitätseinschränkungen erhalten wir als Ergebnis der Berechnungen der Kompatibilitätsfunktion c. Zur Bestimmung einer Berechnungsvorschrift für c haben wir in Abschnitt 1.2 bereits eine mögliche Strategie skizziert. Die Lösungsstrategie basiert auf der Bestimmung der Unterschiede zwischen den Beschreibungen beider Frameworks. Daraus können wir unter Berücksichtigung der bisherigen Benutzung des Frameworks Rückschlüsse auf Kompatibilitätseinschränkungen ziehen. Wir wollen untersuchen, ob dies eine valide Lösungsstrategie ist.

Die Aufgabenstellung setzt voraus, dass alle Erweiterungen in A gemäß der Framework-Beschreibung von F_1 korrekt implementiert wurden, da $c(A, F_1, EW) = \emptyset$ gilt. Zur Lösung des Problems müssen wir nun Kompatibilitätseinschränkungen bestimmen, die der Kompatibilität von A zu F_2 bezüglich der Erweiterungseigenschaft EW entgegenstehen. Diese Kompatibilitätseinschränkungen geben Auskunft darüber, ob Erweiterungen in A nicht gemäß der Framework-Beschreibung F_2 implementiert wurden.

Wir wissen, dass alle Erweiterungen in A gemäß der Framework-Beschreibung von F_1 korrekt implementiert sind. Die Framework-Beschreibung F_1 definiert somit im Umkehrschluss, wie die Erweiterungen in A implementiert sind. Bestimmen wir nun die Veränderungen der Framework-Beschreibungen von F_1 zu F_2, bestimmen wir implizit auch Veränderungen an den Erweiterungen in A, die notwendig wären, wenn A kompatibel zu F_2 sein soll. Wir müssen jedoch einschränken, dass nicht jede Veränderung an den Framework-Beschreibungen einen eindeutigen Schluss auf eine Inkompatibilität zulässt. Dennoch können wir anhand dieser Unterschiede bewerten, ob Erweiterungen nicht gemäß der Framework-Beschreibung von F_2 implementiert sind. Hierzu benötigen wir eine Verfeinerung des Begriffs der Kompatibilitätseinschränkungen. Wir unterscheiden zwei Klassen von Kompatibilitätseinschränkungen:

- *gesichert kompatibilitätsverletzende* Einschränkungen
- *potentiell kompatibilitätsverletzende* Einschränkungen

Diese Unterscheidung richtet sich danach, ob aufgrund der vorliegenden Informationen aus der Differenz der Framework-Beschreibungen eine Einschränkung als eindeutig kompatibilitätsverletzend eingestuft werden kann. Kann nicht eindeutig entschieden werden, ob eine Einschränkung verletzend

ist, etwa weil hierzu eine Beurteilung auf semantischer Ebene notwendig wäre, handelt es sich um eine potentiell kompatibilitätsverletzende Einschränkung.

Die Menge der Kompatibilitätseinschränkungen besteht somit aus der Vereinigung der Menge der potentiell kompatibilitätsverletzenden Einschränkungen mit der Menge der gesichert kompatibilitätsverletzenden Einschränkungen. Dies führt zu folgenden Festlegungen:

Eine Einschränkung ist *potentiell kompatibilitätsverletzend*, wenn sowohl Konstellationen denkbar sind, in denen die Kompatibilitätseinschränkung nicht verletzend als auch verletzend sein kann. Die Menge der potentiell kompatibilitätsverletzenden Einschränkungen wird als Δ_P bezeichnet.

Eine Einschränkung ist *gesichert kompatibilitätsverletzend*, wenn die Einschränkung eindeutig, d.h. von einem Rechner entscheidbar, der Kompatibilität entgegensteht. Die Menge der gesichert kompatibilitätsverletzenden Einschränkungen wird mit Δ_G bezeichnet.

Die Gesamtmenge der potentiell und gesichert kompatibilitätsverletzenden Einschränkungen ist die disjunkte Vereinigung beider Mengen, so dass wir festhalten:

> *Sei* $\Delta = \Delta_P \cup \Delta_G$ *mit* $\Delta_P \cap \Delta_G = \emptyset$ *die Menge der kompatibilitätsverletzenden Einschränkungen.*

Wir denken, dass die Beurteilung der Kompatibilität durch Bestimmung der Unterschiede zwischen zwei Framework-Beschreibungen unter den gegebenen Bedingungen eine valide Lösungsstrategie ist. Wir wollen diese Strategie konkretisieren und einen Lösungsansatz entwerfen, der aus drei Analyseschritten besteht:

- *Benutzungsanalyse*
- *Differenzanalyse*
- *Kompatibilitätsanalyse*

In Abbildung 3.1 stellen wir den Lösungsansatz und seinen unterliegenden Prozess in einem UML-Aktivitätendiagramm dar. Die ersten beiden Aktivitäten der Benutzungs- und Differenzanalyse werden parallel durchgeführt und sind vorbereitende Schritte der Kompatibilitätsanalyse.

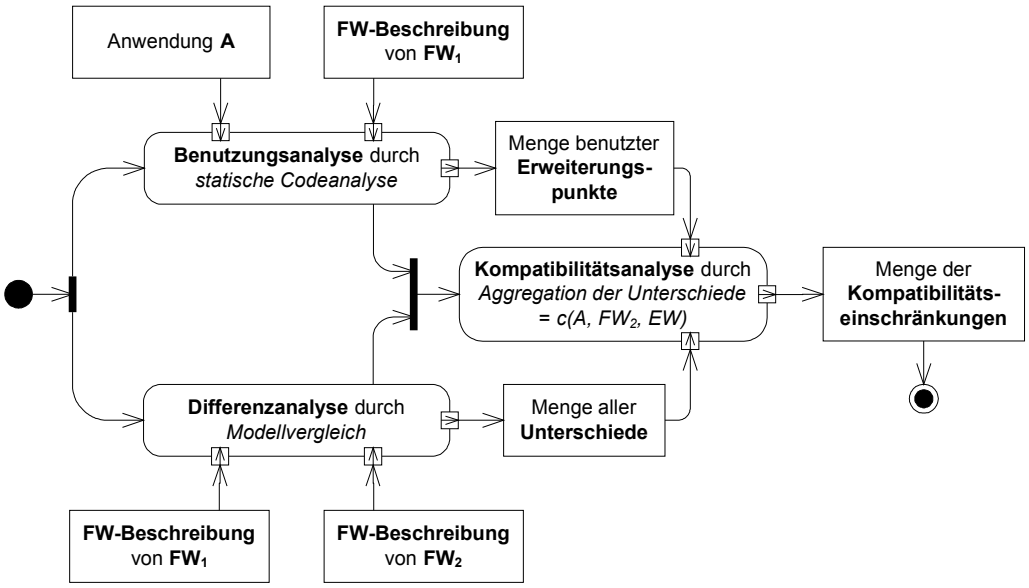


Abbildung 3.1: Lösungsansatz

Wir wollen im Folgenden die einzelnen Analyseschritte des Lösungsansatzes näher betrachten. Als zu betrachtende Eigenschaft haben wir die Erweiterungseigenschaft gewählt. Dies bedeutet, dass wir die Erweiterungen der Anwendung A betrachten müssen. Zur Bestimmung der aktuell implementierten Erweiterungen in A führen wir eine *Benutzungsanalyse* durch. Die Benutzungsanalyse erhält als Eingabe die Anwendung A und die Framework-Beschreibung von FW_1. Mit Hilfe der Technik einer statischen Codeanalyse bestimmt die Benutzungsanalyse die Menge der in A benutzten Erweiterungspunkte.

Neben der Menge implementierter Erweiterungen in A benötigen wir darüber hinaus Informationen über die Veränderungen zwischen den Framework-Beschreibungen von FW_1 und FW_2. Diese Veränderungen können wir mit Hilfe einer *Differenzanalyse* bestimmen. Die Differenzanalyse bestimmt die Menge aller Unterschiede zwischen den Framework-Beschreibungen auf Basis des Vergleichs der unterliegenden Modelle der Framework-Beschreibungen. Wie bereits erwähnt basiert die Differenzanalyse auf einem Vergleich auf syntaktischer Ebene. Wir betrachten keine semantischen Veränderungen und können somit nicht unterscheiden, ob eine syntaktische Veränderung auch zu einer semantischen Veränderung führt. Wir gehen jedoch davon aus, dass eine Veränderung an der Framework-Beschreibung als Auslöser immer eine Veränderung am Framework besitzt. Somit reflektiert jede syntaktische Veränderung der Beschreibung eine Veränderung des Frameworks. Kosmetische Veränderungen der Framework-Beschreibung, die keine semantischen Veränderungen mit sich bringen, schließen wir aus. Diese Veränderungen müssten im Prozess gesondert behandelt werden, idealerweise dadurch, dass diese gezielt als kosmetische Veränderungen durch den Benutzer, der die Änderungen vornimmt, markiert werden.

Die Ergebnisse der beiden vorbereitenden Schritte der Benutzungs- und Differenzanalyse fließen in die abschließende *Kompatibilitätsanalyse*, die auf Basis der Eingangsdaten die Menge $\Delta = c(A, F_2, EW)$ berechnet. In der Kompatibilitätsanalyse werden die Unterschiede unter Berücksichtigung der implementierten Erweiterungen bewertet und die Menge aller potentiellen und gesicherten Kompatibilitätseinschränkungen erstellt.

Der vorgestellte Ansatz ist eine Lösung der formalen Aufgabenstellung auf Seite 66, denn beide geforderten Kriterien werden erfüllt:

- Der Ansatz enthält mit der Kompatibilitätsanalyse eine Berechnungsvorschrift für die Kompatibilitätsfunktion $c()$.
- Der Ansatz berechnet als Ergebnis die gesuchte Menge der Kompatibilitätseinschränkungen $\Delta = c(A, F_2, EW)$.

Nach diesem Überblick über den Lösungsansatz wollen wir in den folgenden Abschnitten die Aktivitäten der Benutzungs-, Differenz- und Kompatibilitätsanalyse aus Abbildung 3.1 genauer betrachten.

3.1 Konzept zur Benutzungsanalyse

Die Benutzungsanalyse bestimmt alle benutzten Erweiterungspunkte in einer Anwendung A. Das Konzept für diese Analyse zeigen wir in Abbildung 3.2. Für die Benutzungsanalyse unterziehen wir der Anwendung zunächst einer *statischen Analyse*, die bestimmt, welche statischen Codeabhängigkeiten zwischen der Anwendung und dem Framework vorliegen. Statische Codeabhängigkeiten entstehen, wenn in der Anwendung Schnittstellen aus dem Framework implementiert werden, Methodenaufrufe zum Framework hin vorliegen oder Klassen aus dem Framework referenziert werden. Die statische Analyse ist die Basis, um die verwendeten Erweiterungspunkte zu identifizieren.

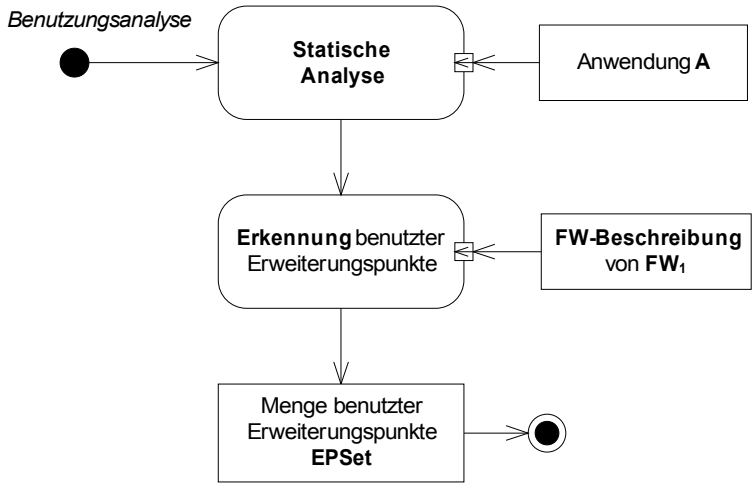


Abbildung 3.2: Konzept zur Benutzungsanalyse

Zur *Erkennung benutzter Erweiterungspunkte* iterieren wir durch alle verfügbaren Erweiterungspunkte, die das Framework FW_1 anbietet. Die Menge der verfügbaren Erweiterungspunkte und ihre Beschreibung entnehmen wir der Framework-Beschreibung von FW_1. Für jeden Erweiterungspunkt EP von FW_1 prüfen wir nun, ob wir eine statische Codeabhängigkeit identifiziert haben, die belegt, dass die Anwendung den EP implementiert.

Die statische Analyse ist in der Lage für alle Klassen in A die implementierten Methoden zu erkennen. Aus der Framework-Beschreibung FW_1 können wir außerdem ermitteln, welche Methoden spezielle Methoden von Erweiterungspunkten (Hook-Methoden) sind. Durch Kombination dieser beiden Informationen können wir feststellen, ob eine Klasse des Frameworks eine Implementierung eines Erweiterungspunktes EP ist. Entdecken wir derartige Zusammenhänge, haben wir einen Beweis gefunden, dass A eine Erweiterung für EP implementiert. In diesem Fall markieren wir den EP als benutzt, so dass wir am Ende eine Menge aller benutzten Erweiterungspunkte erhalten.

3.2 Konzept zur Differenzanalyse

Die Differenzanalyse stützt sich auf den Vergleich der unterliegenden Modelle zweier Framework-Beschreibungen. Wir gehen davon aus, dass der Framework-Beschreibung ein Meta-Modell zugrunde liegt, so dass Instanzen, also konkrete Framework-Beschreibungen, mit Hilfe eines Modellvergleichs verglichen werden können. Das Vorgehen ist ein zweischrittiger Prozess, den wir in Abbildung 3.3 aufzeigen.

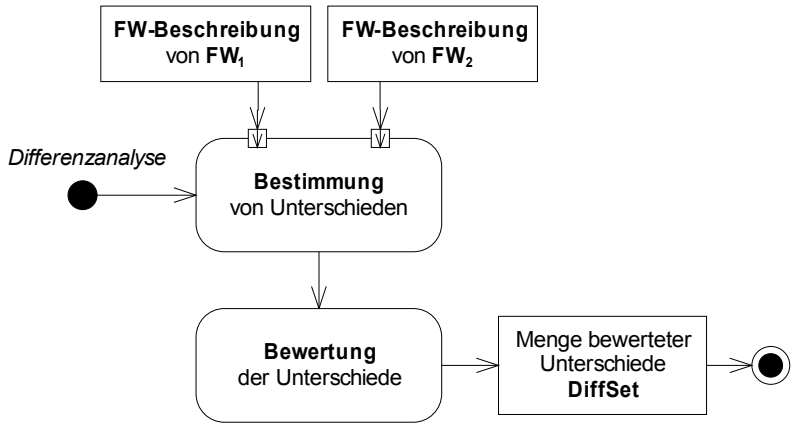


Abbildung 3.3: Konzept zur Differenzanalyse

Die *Bestimmung der Unterschiede* setzt voraus, dass man erkennt, welche Objekte sich in den verglichenen Modellen entsprechen. Diese Korrespondenzen zwischen den Modellen ist die Grundlage, um zu erkennen, welche Objekte hinzugefügt, welche verändert und welche gelöscht wurden. Bei Vergleich zweier Modelle für eine Framework-Beschreibung kennen wir die Semantik der Objekte. So sind wir in der Lage die Veränderungen auf Modelle-

bene im Sinne der Framework-Beschreibung zu interpretieren und so Aussagen über Veränderungen am Framework zu treffen.

Auf diese Art und Weise ist die Differenzanalyse zum Beispiel in der Lage zu erkennen, dass sich der Typ des Rückgabewertes einer Methode geändert hat. Diese Veränderung können wir weiter aggregieren, indem wir zum Beispiel feststellen, dass diese Methode zur Bindungsschnittstelle eines bestimmten Erweiterungspunktes gehört. Somit können wir die Aussage treffen, dass der Erweiterungspunkt verändert wurde und diese Veränderung im Detail belegen.

Jeder erkannte Unterschied wird anschließend einer Bewertung unterzogen. Die Bewertung bestimmt die möglichen Auswirkungen des erkannten Unterschieds auf die Kompatibilität, wenn wir davon ausgehen, dass eine Anwendung die nun veränderte Funktionalität des Frameworks verwendet hat.

Zur *Bewertung der Unterschiede* erstellen wir einen *Kompatibilitätskatalog*, der alle denkbaren Unterschiede zwischen zwei Framework-Beschreibungen systematisch auflistet. Jeder Unterschied wird im Katalog hinsichtlich potentieller und gesicherter Einschränkungen bewertet. So kann man über die Menge der erkannten Änderungen die Auswirkung der Veränderung im Katalog nachschlagen.

Im Kompatibilitätskatalog wird zum Beispiel die Veränderung des Typs eines Rückgabewertes einer Methode, die zu einem Erweiterungspunkt gehört, als gesichert kompatibilitätsverletzende Einschränkung deklariert. Begründung ist, dass bei Aufruf der Methode zur Laufzeit in jedem Fall zu einem Fehler im Programm auftreten würde. Ein Beispiel für eine potentiell kompatibilitätsverletzende Einschränkung wäre die Veränderung einer einzuhaltenden Bedingung an einem Erweiterungspunkt. Da die Analyse nicht bestimmen kann, ob die Veränderung etwa eine Verstärkung oder Abschwächung der Bedingung ist, können wir die Auswirkungen auf die Kompatibilität nicht gesichert einschätzen. Außerdem ist denkbar, dass die Implementierung die neue Bedingung bereits erfüllt, obwohl diese zuvor nicht explizit gefordert war.

Das Ergebnis der Differenzanalyse ist die Menge der bewerteten Unterschiede zwischen den Framework-Beschreibung von FW_1 und FW_2.

3.3 Konzept zur Kompatibilitätsanalyse

Die Kompatibilitätsanalyse erhält als Eingabe die Menge aller benutzten Erweiterungspunkte (EPSet) des Frameworks und die Menge aller Unterschiede (DiffSet) zwischen den Framework-Versionen. Die Kompatibilitätsanalyse ist ein Prozess aus drei Schritten, den wir in Abbildung 3.4 aufgezeichnet haben.

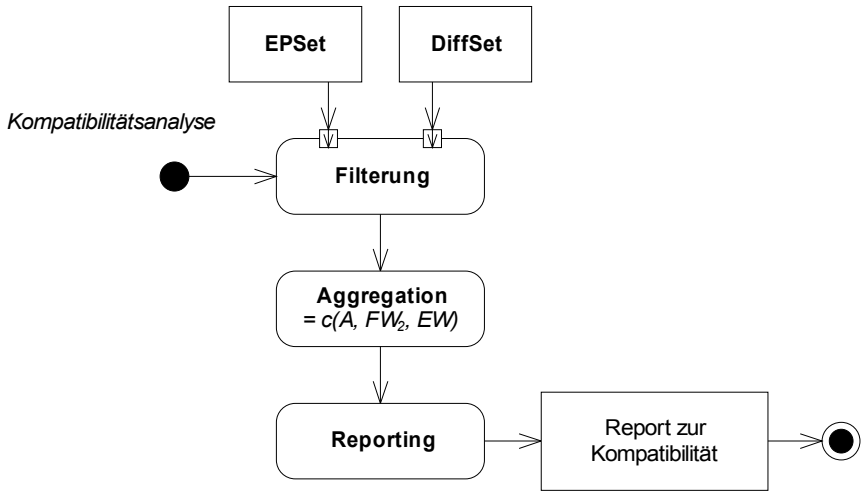


Abbildung 3.4: Konzept zur Kompatibilitätsanalyse

Während der *Filterung* wird die Menge aller Unterschiede gefiltert, so dass nur noch die Unterschiede als Filtrat übrig bleiben, die sich auf die aktuelle Benutzung des Frameworks beziehen. Hintergrund ist die Annahme, dass Veränderungen in Teilen des Frameworks, die nicht benutzt werden, keinen Einfluss auf die Kompatibilität haben können und somit unbeachtet bleiben.

Die Menge aller relevanten Unterschiede wird nun im Hinblick auf mögliche kompatibilitätsverletzende Veränderungen aggregiert. Die *Aggregation* fast Veränderungen zusammen und bewertet diese im Kontext anderer Veränderungen neu. In diesem Schritt steckt die eigentliche Berechnungsvorschrift für die Kompatibilitätsfunktion $c(A, F_2, EW)$. Das Ergebnis der Aggregation ist die Menge Δ der potentiellen und gesicherten Kompatibilitätseinschränkungen.

Die Menge Δ fließt in das abschließende *Reporting* ein. In einem Report werden die erkannten Kompatibilitätseinschränkungen in natürlicher Sprache aufbereitet. Mit Hilfe des Kompatibilitätskatalogs können die Veränderungen und deren Bewertung in Textbausteine überführt werden, die im Report aggregiert werden. So können Veränderungen zum Beispiel in folgende Textbausteine überführt werden:

> „Verletzung der Kompatibilität durch Veränderung des Typs von Typ_1 auf Typ_2 des Rückgabewertes der Methode M der Bindungsschnittstelle BS des Erweiterungspunktes E."

> „Potentielle Verletzung der Kompatibilität durch Veränderung der Bedingung B von $Wert_1$ auf $Wert_2$ am Erweiterungspunkt E."

Der Report zur Kompatibilität kann zum Beispiel einem Projektleiter als Grundlage für weitere Analysen dienen. Selbstverständlich können auch die Rohdaten aus dem Prozess nach außen propagiert werden, so dass etwaige weitere Werkzeuge auf Basis dieser Daten operieren können.

3.4 Zusammenfassung

In diesem Kapitel haben wir den Ansatz zur automatischen Kompatibilitätsprüfung bei Aktualisierung eines Framework erörtert. Hierzu haben wir auf Basis der Definitionen zur Kompatibilität die Aufgabenstellung formalisiert und einen Lösungsprozess entwickelt. Die einzelnen Schritte der Kompatibilitätsprüfung mit Benutzungsanalyse, Differenzanalyse und Kompatibilitätsanalyse wurden konzeptionell beschrieben.

Dreh- und Angelpunkt des Ansatzes ist eine geeignete Framework-Beschreibungssprache, die es erlaubt den aufgezeigten Ansatz umzusetzen. Daher werden wir im Folgenden auf die Details einer geeigneten Framework-Beschreibungssprache eingehen, existierende Ansätze bewerten und eine Lösung für dieses Teilproblem im Kontext der Kompatibilitätsprüfung erarbeiten.

4

Framework-Beschreibungssprachen

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this considering programs to be works of literature.

-- Donald E. Knuth

In den vorangegangenen Kapiteln haben wir Frameworks und ihre Kompatibilität eingehend betrachtet. Wir haben den Einsatz von Frameworks in der Praxis beleuchtet und anhand eines Framework-Benutzungs-Modells dargelegt, wie sich die Benutzung eines Frameworks für einen Entwickler darstellt. Des weiteren haben wir in der Beschreibung unseres Lösungsansatzes dargelegt, welche zentrale Rolle die Framework-Beschreibung im Kontext dieser Arbeit spielt.

In diesem Kapitel untersuchen wir, welche Ansätze zur Framework-Beschreibung existieren und entscheiden, welcher Ansatz in unserer Lösung eingesetzt wird. Hierzu definieren wir im nachfolgenden Abschnitt zunächst Anforderungen an Framework-Beschreibungssprachen, die sich aus dem Kon-

text dieser Arbeit ergeben. Auf Basis dieser Anforderungen beurteilen wir dann bestehende Ansätze.

4.1 Anforderungen

Wir untergliedern die zu stellenden Anforderungen in drei Bereiche. Diese Bereiche ergeben sich aus der Überlegung, wozu man eine Framework-Beschreibungssprache benötigt und wonach man sie beurteilen kann. Wir verwenden folgende drei Bereiche:

- Anforderungen, die sich aus dem Verwendungszweck ergeben und sich auf die *Anwendbarkeit* der Sprache beziehen.
- Anforderungen an die *Ausdrucksfähigkeit* der Sprache, um alle relevanten Merkmale eines Frameworks beschreiben zu können.
- Anforderungen an die *Erweiterbarkeit* der Sprache, um flexibel zukünftige Entwicklungen berücksichtigen zu können.

Diese drei Bereiche wollen wir im Folgenden genauer betrachten und Anforderungen aus jedem Bereich formulieren.

4.1.1 Anwendbarkeit

Wir unterscheiden für eine Framework-Beschreibungssprache drei Anwendergruppen, die wir im Hinblick auf die Anwendbarkeit einer Framework-Beschreibungssprache unterscheiden wollen. Diese drei Anwendergruppen bauen auf dem Rollenkonzept für die Benutzung von Frameworks aus Abschnitt 2.2.2 auf.

- *Framework-Benutzer*, die eine Framework-Beschreibung verwenden, um mit ihrer Hilfe ein Framework zu benutzen.
- *Framework-Entwickler*, die ein Framework beschreiben.
- *Werkzeug-Entwickler*, die die Informationen einer Framework-Beschreibung automatisch auswerten wollen, um unterstützende Werkzeuge zu entwickeln.

Die erste Anwendergruppe ist aus Sicht eines Softwareentwicklers entscheidend, der ein Framework einsetzen möchte. Johnson setzt diese Gruppe in den Fokus, wenn er definiert, was eine Framework-Beschreibung ausmacht. Nach [Johnson1992] erfüllt eine Framework-Beschreibung drei Zwecke.

> *Documentation for a framework has three purposes [...]. Documentation must describe the purpose of the framework, how to use the framework, [and] the detailed design of the framework.*
>
> **Framework-Beschreibung nach [Johnson1992]**

Für einen Framework-Benutzer ist es wichtig, möglichst schnell den Zweck und die Einsatzmöglichkeiten des Frameworks zu verstehen. Diese abstrakte Sicht auf das Framework ermöglicht es zu entscheiden, ob ein bestimmtes Framework die geeignete Wahl ist [Froehlich2000]. Anschließend ist wichtig zu verstehen wie man das Framework benutzt. Hierfür wird man bei manchen Problemstellungen nicht umhin kommen, die detaillierte Funktionsweise des Frameworks zu verstehen.

Die zweite Anwendergruppe der Framework-Entwickler steht vor dem Problem, eine adäquate Framework-Beschreibung erstellen zu müssen. Verschiedene Ansätze müssen dahingehend beurteilt werden, ob die Erstellung einer Framework-Beschreibung ein Hindernis ist und zum Beispiel mit übermäßigem Aufwand verbunden ist. Da die Erstellung von Dokumentationen eine unliebsame Aufgabe in Softwareprojekten ist, der häufig zu wenig Aufmerksamkeit geschenkt wird, muss es für Framework-Entwickler möglichst einfach sein, eine Framework-Beschreibung zu erstellen.

Die letzte Gruppe der Werkzeug-Entwickler ist aus dem Blickwinkel dieser Arbeit von besonderem Interesse, da die Informationen einer Framework-Beschreibung für den Ansatz der Beurteilung der Kompatibilität automatisch auswertbar sein müssen. Es hilft nur bedingt, wenn das gesamte Framework in einem mehr oder weniger strukturierten Fließtext dokumentiert ist, da diese Repräsentation es nicht erlaubt, auf einzelne Informationen gezielt zuzugreifen.

Aus diesen Überlegungen ergeben sich die in Tabelle 4.1 zusammengestellten Anforderungen an die Anwendbarkeit von Framework-Beschreibungssprachen. Hierbei unterteilen wir die Anforderungen anhand der drei Anwendergruppen als entsprechende Zielgruppen.

Tabelle 4.1: Anforderungen an die Anwendbarkeit von Framework-Beschreibungssprachen

ID	Zielgruppe & Anforderung	Erläuterung
ANW-B	*Framework-Benutzer* Eine Framework-Beschreibungssprache muss Konzepte unterstützen, die einen Benutzer bei Verwendung einer Framework-Beschreibung, d.h. beim Explorieren und Erlernen eines Frameworks unterstützen.	Diese Anforderung zielt auf die Strukturierung einer Framework-Beschreibung ab. Diesen Aspekt sollte bereits die Beschreibungssprache berücksichtigen, so dass sich Framework-Entwickler bei der Erstellung über diesen Aspekt keine Gedanken machen brauchen. Für den Framework-Benutzer ist es wichtig eine Beschreibung an die Hand zu bekommen, die ihn zum einen durch das Framework führt und zum anderen als Nachschlagewerk dienen kann [Meusel1997].

ID	Zielgruppe & Anforderung	Erläuterung
ANW-E	*Framework-Entwickler* Die Erstellung der Framework-Beschreibung muss sich so in den Entwicklungsprozess des Frameworks integrieren, dass der Aufwand der Entwickler für die Erstellung möglichst gering ist.	Möglichst viele Arbeitsschritte zur Erstellung der Framework-Beschreibung sollten automatisch durchgeführt werden, so dass der Framework-Entwickler nur noch fehlende Informationen ergänzen muss. Idealerweise ist die Dokumentation des Frameworks direkt mit dessen Entwicklung verwoben, so dass kein Bruch in der Arbeitsweise des Framework-Entwicklers notwendig ist.
ANW-W	*Werkzeug-Entwickler* Die Informationen einer Framework-Beschreibung müssen für Werkzeug-Entwickler programmatisch auswertbar sein.	Viele Dokumentationen erfassen ihre Informationen in einem fortlaufenden Text, der z.B. mit einer Textverarbeitung erstellt wird. Der Text kann sehr gut sein und alle wichtigen Informationen enthalten, aber es ist nur sehr schwer möglich, gezielt Teile dieser Informationen aus dem Dokument zu extrahieren. Werkzeuge können die Informationen aus dem Dokument nicht auswerten, um z.B. automatische Analysen durchzuführen. Daher ist es wichtig, dass die Informationen in der Framework-Beschreibung so repräsentiert sind, dass sie automatisch ausgewertet werden können.

4.1.2 Ausdrucksfähigkeit

Im vorangegangenen Kapitel 2.3 ab Seite 36 haben wir eine Reihe von Merkmalen für Frameworks und deren Erweiterungspunkte erarbeitet. Diese Merkmale werden nun in Anforderungen an die Ausdrucksfähigkeit von Framework-Beschreibungssprachen aus Sicht der Framework-Benutzer überführt. Die Ausdrucksfähigkeit einer Sprache bestimmt was mit der Sprache inhaltlich beschrieben werden kann. Daher leiten sich diese Anforderungen aus Sicht der Framework-Benutzer ab, denn diese Zielgruppe muss inhaltlich mit der Framework-Beschreibung arbeiten. Wir stellen keine Anforderungen an die Ausdrucksfähigkeit aus Sicht der weiteren Benutzergruppen, da diese keine inhaltlichen Anforderungen beisteuern könnten.

Grundgedanke bei Aufstellung der Anforderung ist, dass die erwähnten Merkmale von einer Framework-Beschreibung erfasst werden können müssen, um aus Sicht der Framework-Benutzer zu einer vollständigen Beschreibung zu gelangen. Um die Anforderungen mit den bereits erfassten Framework- bzw. Erweiterungsmerkmalen in Relation zu bringen, führen wir Querverweise zu den Merkmalen aus Kapitel 2.3 in der Spalte »REF« (für Referenz) auf. Die Vollständigkeit der Anforderungen bemisst sich darüber, dass alle Framework- bzw. Erweiterungsmerkmale mit den Anforderungen abgedeckt sind.

Tabelle 4.2: Anforderungen an die Ausdrucksfähigkeit von Framework-Beschreibungen aus Sicht der Framework-Benutzer

ID	REF	Anforderung an die Ausdrucksfähigkeit Eine Framework-Beschreibungssprache muss...	Fragestellung
AUS-1	FM-1 FM-2 FM-3 FM-4 FM-5	... Beispiele für die Benutzung des Frameworks beschreiben können.	Gibt es Beispiele, die die Domäne des Frameworks veranschaulichen und so zeigen wie ein Feature des Frameworks benutzt werden kann und welche Probleme damit gelöst werden können? Anschauliche Beispiele und Tutorien helfen ein Framework korrekt zu benutzen [Gangopadhyay1995].
AUS-2	FM-1 FM-3 FM-4 FM-6	... statische und dynamische Architekturübersichten beschreiben können.	Wie ist die Gesamtarchitektur des Frameworks und an welchen Stellen befinden sich Erweiterungspunkte? Wie ist der Kontrollfluss des Frameworks und wo werden Erweiterungspunkte eingebunden? Um Erweiterungspunkte korrekt benutzen zu können, ist es unerlässlich die statische und dynamische Architektur zu kennen.
AUS-3	FM-5 FM-6	... die Erweiterungspunkte eines Frameworks beschreiben können.	Welche konzeptionelle Einheiten können sich an einen Erweiterungspunkt hängen? Einheiten könnten sein: Klassen, Komponenten gemäß eines Komponentenmodells (EJB, Corba), (Web) Services, aber auch deklarative Einheiten wie XML-Dateien, Modelle, Grafiken etc.
AUS-4	FM-7	... die Konfiguration und Integration des Frameworks beschreiben können.	Wie kann das Framework konfiguriert und in eine größere Softwarelandschaft, die ggf. aus mehreren Frameworks besteht, integriert werden?
AUS-5	FM-8	... die Implementierung einer Erweiterung beschreiben können.	Was muss für die Implementierung einer entsprechenden Erweiterung geleistet werden? Im Falle von Klassen etwa: Von welcher Vaterklasse muss geerbt werden? Welche Methoden müssen überschrieben werden oder welche Schnittstelle ist zu implementieren?
AUS-6	FM-8 EM-1 EM-2 EM-3	... das Kommunikationsprotokoll zwischen Framework und Erweiterung beschreiben können.	Welches Kommunikationsprotokoll im Sinne eines Schnittstellenprotokolls muss eingehalten und bei der eigenen Implementierung berücksichtigt werden? Wird ein Lebenszyklus im Sinne eines Protokolls für eine Erweiterung durch das Framework vorgegeben?

ID	REF	Anforderung an die Ausdrucksfähigkeit Eine Framework-Beschreibungssprache muss...	Fragestellung
AUS-7	FM-8 EM-2	... die Bindung von Erweiterungen an das Framework beschreiben können.	Wie entsteht die Bindung zwischen einer Erweiterung und dem Framework, so dass die Erweiterung entsprechend eingebunden und verwendet wird?
AUS-8	FM-8 EM-4	... das Format und die Installation einer Erweiterung in das Framework beschreiben können.	Wie muss eine Erweiterung im Framework installiert und in welchem Format muss sie ausgeliefert werden (engl. deployment)? Neben der korrekten Bindung erwarten Frameworks häufig, dass man bestimmte Vorgaben bezüglich Format und Paketierung der Erweiterungen einhält.
AUS-9	FM-8 EM-5	... einzuhaltende Bedingungen seitens einer Erweiterung beschreiben können.	Welche Bedingungen im Sinne von Einschränkungen sind an eine Implementierung einer Erweiterung geknüpft, um korrekt mit dem Framework zusammenzuarbeiten? Welche Eigenschaften muss die Implementierung erfüllen und welche darf sie nicht verletzen?
AUS-10	FM-8 EM-6	... Varianten von Erweiterungspunkten beschreiben können.	Welche Varianten eines Erweiterungspunktes gibt es? Gibt es zu einem Erweiterungspunkt z.B. eine speziellere Variante, die ein anderes Kommunikationsprotokoll oder andere einzuhaltende Bedingungen erfordert?

Mit Hilfe der Referenzen erkennen wir, dass alle Framework- bzw. Erweiterungsmerkmale mit den Anforderungen an die Ausdrucksfähigkeit abgedeckt sind. Somit stellen wir sicher, dass jedes relevante Merkmal von einer geeigneten Framework-Beschreibungssprache berücksichtigt wird.

4.1.3 Erweiterbarkeit

Auch wenn sich die beschriebenen Grundkonzepte der Framework-Entwicklung nicht verändern mögen, so ist es doch vorstellbar, dass neue Ideen und Sprachen bei der Entwicklung von Frameworks in zukünftigen Framework-Beschreibungen berücksichtigt werden müssen. Aus diesem Grund müssen Framework-Beschreibungssprachen auch für die Zukunft gewappnet sein und ihre Erweiterbarkeit gewährleistet werden.

Erweiterbarkeit bezieht sich darauf, dass z.B. für die Art und Weise wie Anforderungen aus Tabelle 4.2 umgesetzt werden, Alternativen angeboten werden. Neue Sprachen, die unabhängig von Framework-Beschreibungen zur Spe-

zifikation und Dokumentation von Software verwendet werden, sollten in Framework-Beschreibungen integriert werden können. Eine Framework-Beschreibungssprache sollte demnach so gestaltet sein, dass sie in der Lage ist, bestehende Sprachen für bestimmte Aspekte der Beschreibung wiederzuverwenden, so dass Sprachen verwendet werden können, mit denen die Beteiligten bereits vertraut sind. Ein Beispiel hierfür ist die Integration der UML [UML-SUP23], die sich zu einem Standard für die Spezifikation von Software etabliert hat.

Eine weitere Form der Erweiterbarkeit bezieht sich auf technische Details verschiedener Frameworks, die gegebenenfalls Erweiterungen der Beschreibungssprache erfordern. Ein Aspekt hierbei ist die Beschreibung von Erweiterungspunkten. Erweiterungspunkte können je nach Framework technisch unterschiedlich realisiert werden, so dass deren Beschreibung dementsprechend unterschiedlich ausfällt. Es gibt beispielsweise Frameworks, bei denen eine Erweiterung statt in einer universellen Programmiersprache in einer domänenspezifischen Sprache programmiert werden. Eine solche domänenspezifische Sprache könnte dementsprechend Anpassungen bei der Beschreibung von Erweiterungspunkten erfordern.

Wir betrachten die Erweiterbarkeit wieder aus Sicht der drei Zielgruppen Framework-Benutzer, -Entwickler und Werkzeug-Hersteller. Für Framework-Benutzer ist es wichtig, dass bereits vertraute Sprachen verwendet werden können. Framework-Entwickler hingegen legen Wert darauf, dass für zukünftige Frameworks die Beschreibungssprache angepasst werden kann. Die Gruppe der Werkzeug-Entwickler benötigt eine Form der Sprachdefinition der Framework-Beschreibungssprache, die es ermöglicht Veränderungen auf Basis der Sprachdefinition möglichst leicht in Werkzeugen berücksichtigen zu können.

Tabelle 4.3: Anforderungen an die Erweiterbarkeit von Framework-Beschreibungen

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Erläuterung
ERW-B	*Framework-Benutzer* ... existierende Sprachen, mit denen Framework-Benutzer bereits vertraut sind, zur Beschreibung einzelner Merkmale integrieren können.	Um das Rad in vielen Bereichen der Spezifikation und Dokumentation von Software nicht neu zu erfinden, muss eine Framework-Beschreibungssprache in der Lage sein, existierende und neue Sprachen zu integrieren und wiederzuverwenden.
ERW-E	*Framework-Entwickler* ... es den Framework-Entwicklern ermöglichen, die Framework-Beschreibung um neue Beschreibungsformen für Erweiterungspunkte zu erweitern.	Da es für Erweiterungspunkte viele Möglichkeiten der technischen Realisierung gibt, die ggf. spezielle Beschreibungsformen bedingen, muss eine Framework-Beschreibungssprache im Hinblick auf die Beschreibung von Erweiterungspunkten erweiterbar sein.

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Erläuterung
ERW-W	*Werkzeug-Entwickler* ... müssen in der Lage sein, Werkzeuge auf Basis der Sprachdefinition der Framework-Beschreibungssprache zu erweitern.	Verändert sich die Sprachdefinition der Framework-Beschreibungssprache, so müssen die Werkzeuge angepasst werden. Werkzeug-Entwickler muss es möglich sein, die Werkzeuge auf Basis der Sprachdefinition entsprechend anzupassen.

4.2 Existierende Ansätze zur Framework-Beschreibung

Dieser Abschnitt zeigt eine Übersicht bestehender Ansätze, um Frameworks mit ihren möglichen Erweiterungen zu spezifizieren und ihre Benutzung zu dokumentieren. Tabelle 4.4 enthält einen chronologischen Überblick der relevanten Beiträge und Ansätze im Kontext dieser Arbeit. Es fällt auf, dass es keine neueren Arbeiten zum Thema Framework-Beschreibungen gibt. Der Grund hierfür liegt darin, dass die Forschung im Bereich von Frameworks durch andere Themen, wie Service-Orientierung, abgelöst wurde. Somit ist unsere eigene Arbeit, der wir uns in Kapitel 6 ausführlich widmen werden, aus dem Jahr 2010 die aktuellste zu diesem Thema.

Im Verlauf dieses Abschnitts werden wir die verwandten Arbeiten eingehen und zentrale Ansätze anhand von Beispielen genauer auf ihre Anwendbarkeit auf das gestellte Problem der Framework-Beschreibung im Hinblick auf eine Framework-Aktualisierung analysieren.

Tabelle 4.4: Chronologie relevanter Beiträge und bestehender Ansätze

Jahr	Konzept / Ansatz	Autoren und Verweise
1988	Erste Definition des Framework-Begriffs	R. E. Johnson, B. Foote [Johnson1988]
	Kochbuch zur Beschreibung der Framework-Benutzung am Beispiel eines Model-View-Controller Frameworks in smalltalk-80	G. E. Krasner, S. T. Pope [Krasner1988]
1990	Kontrakte zur Beschreibung der Kommunikation zwischen Objekten – speziell für die Entwicklung von Frameworks	R. Helm, I. M. Holland, D. Gangopadhyay [Helm1990]
1992	Framework Dokumentation mit Hilfe von Patterns	Ralph E. Johnson [Johnson1992]

Jahr	Konzept / Ansatz	Autoren und Verweise
1993	Design Patterns zur Beschreibung von objekt-orientiertem Design	Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides [Gamma1993][Gamma1995]
1994	Meta-patterns: Hot-Spots in Verbindung mit Template- und Hook-Methoden	W. Pree [Pree1994][Pree1995]
1995	Design Patterns für die objekt-orientierte Softwareentwicklung	W. Pree, J. Kepler [Pree1995]
	Frameworks verstehen durch die Verwendung von Beispielen	D. Gangopadhyay, S. Mitra [Gangopadhyay1995]
	Active Cookbook als interaktive Anleitung zur Framework-Benutzung	A. Schappert, P. Sommerlad, W. Pree [Schappert1995]
1996	Eine Pattern Sprache für die Entwicklung objekt-orientierter Frameworks	D. Roberts, R. E. Johnson [Roberts1996] [Roberts1998]
	Hot Spot Cards	W. Pree, G. Pomberger [Pree1996]
1997	Hooks zur Beschreibung der Framework-Benutzung	G. Froehlich, H. J. Hoover, L. Liu, P. Sorenson [Froehlich1997]
1998	Framelets als Architekturbausteine als Gegenentwurf zur Verwendung von Frameworks	W. Pree, E. Althammer, H. Sikora [Pree1998]
1999	SmartBook zur automatisierten Unterstützung eines Entwicklers bei der Entwicklung einer neuen Anwendung auf Basis eines Frameworks.	A. Ortigosa, M. Campo, R. Moriyn [Ortigosa1999] [Ortigosa1999a]
2000	Hypermedia Links und Framework Kontrakte zur Beschreibung von Frameworks	S. Demeyer, K. De Hondt, P. Steyaert [Demeyer2000]
	UML-F: Ein UML-Profil für Framework-Architekturen	M. Fontoura,W. Pree, B. Rumpe [Fontoura2000]
	Verstehen von Frameworks durch Visualisierung	K. Jackson, R. Biddle, E. Tempero [Jackson2000]
	Hot-Spot getriebene Framework Entwicklung	W. Pree [Pree2000]
	Framework Beschreibung durch „concern-specific“ Design Patterns und deren Komposition	A. R. Silva, F. A. Rosa, T. Goncalves [Silva2000]
2001	Erweiterung der UML zur Beschreibung von Design Patterns	M. Fontoura, C. Lucena [Fontoura2001]

Jahr	Konzept / Ansatz	Autoren und Verweise
2003	Framework Design Sprache F-UML	N. Bouassida, H. Ben-Abdallah, F. Gargouri, A. B. Hamadou [Bouassida2003]
2004	UML basiertes Framework Design	H. Ben-Abdallah, N. Bouassida, F. Gargouri, A. Ben-Hamadou [Ben-Abdallah2004]
2006	Regelbasierte Weiterentwicklung von Frameworks	M. Cortés, M. Fontoura, C. Lucena [Cortes2006a]
2010	Meta-Modell einer ganzheitlichen Framework-Beschreibungssprache	F. Christ, J.-C. Bals, G. Engels, C. Gerth, M. Luckey [Christ2010]

In Abschnitt 2.2 haben wir Definitionen für Frameworks behandelt und die Eigenschaften von Frameworks herausgearbeitet. Eine zentrale Eigenschaft von Frameworks ist deren Fähigkeit durch Erweiterungen angepasst zu werden, siehe Kapitel 2.3 Merkmal FM-5. Betrachtet man objektorientierte Frameworks im Detail wird diese Anpassbarkeit auf technischer Ebene mit Hilfe so genannter Template- und Hook-Methoden realisiert. Da dieses Konzept zentral für einige verwandte Ansätze zur Framework-Beschreibung ist, wollen wir es hier erläutern.

Das Konzept von Template- und Hook-Methoden ist sehr verbreitet beim Einsatz von Whitebox-Frameworks. Betrachten wir zunächst eine Definition von Template-Methoden aus [Wirfs-Brock1990].

> *Template methods provide step-by-step algorithms. Each step can invoke an abstract method, which the subclass must define, or a base method. The purpose of a template method is to provide an abstract definition of an algorithm. The subclass must implement specific behavior to provide the services required by the algorithm.*
>
> **Template-Methode nach [Wirfs-Brock1990]**

Eine Template-Methode definiert einen komplexen Ablauf, wobei dieser komplexe Ablauf angepasst werden kann, indem bestimmte Methoden, die während des komplexen Ablaufs aufgerufen werden, ausgetauscht werden können. Der Austausch geschieht auf technischer Ebene dadurch, das die abstrakte Methode in einer Unterklasse überschrieben wird [Gamma1993]. Eine solche abstrakte Methode, die von einer Template-Methode aufgerufen wird, bezeichnet man als Hook-Methode [Pree1995].

Die Hook-Methoden sind die Methoden, die von einem Benutzer eines Frameworks zu implementieren sind. Die Template-Methoden spezifizieren an welcher Stelle und in welchem Zusammenhang die Hook-Methoden vom Framework aufgerufen werden. Da das Framework aus seinen Template-

Methoden heraus Hook-Methoden aufruft, die von einem Framework-Benutzer zu implementieren sind, ist das Framework der Auslöser für die Ausführung des benutzerdefinierten Programmcodes. Aus dieser Begebenheit entspringt die Metapher des »Hollywood-Prinzips« bei Frameworks mit dem Slogan »Don't call us – we call you!«. In der Fachliteratur nennt man dieses Prinzip »Inversion of Control«, da die Kontrolle der Anwendung dem Framework übergeben wird und Erweiterungen passiv aufgerufen werden.

4.2.1 Beschreibungsvorlagen und Kochbücher

Eine der ersten Varianten zur Beschreibung von Frameworks war der Einsatz von Beschreibungsvorlagen, die definieren, wie eine Framework-Beschreibung aufgebaut ist. Durch Ausfüllen der Vorlage entsteht eine standardisierte Framework-Beschreibung. Ein häufig verwendetes Konzept einer solchen Beschreibungsvorlage ist das so genannte »Kochbuch«. Es beschreibt in Anlehnung an eine Rezeptsammlung, wie man das Framework in unterschiedlichen Szenarien für unterschiedliche Problemstellungen verwendet. Hierbei liegt der Schwerpunkt meist auf den typischen Verwendungsszenarien des Frameworks, was für einen Framework-Benutzer genau dann zum Problem wird, wenn er sich einem Problem gegenübersieht, das kein typisches Szenario darstellt, so dass keine Lösung im Kochbuch vorhanden ist.

Eine der ersten Framework-Beschreibungen dieser Art wurde von Krasner und Pope veröffentlicht. Sie beschrieben das Model-View-Controller Design für Benutzungsoberflächen in Smalltalk-80 mit Hilfe einer solchen Rezeptsammlung [Krasner1988]. Die Struktur einer solchen Sammlung definiert Johnson über eine Sprache, die aus aufeinander aufbauenden Mustern (engl. pattern) besteht [Johnson1992]. Diese Muster sind nicht mit den Design-Pattern für objektorientiertes Design zu verwechseln, zu denen wir nachfolgend kommen werden. Jedes Muster ist nach [Johnson1992] problemorientiert aufgebaut. Es beschreibt informell, wie man einen kleineren abgeschlossenen Teil eines größeren Design-Problems löst. Das Muster verweist dabei auf verwandte Muster und leitet den Leser zu weiteren Mustern, die ihm helfen könnten. Ein Muster besteht nach [Johnson1992] aus einer Zusammenfassung zu Beginn und einem informellen Fließtext mit Beispielen und Verweisen auf verwandte Muster. Am Ende eines Musters folgt nochmals eine Liste mit Links zu weiterführenden Mustern.

Das Vorlagen-Prinzip wurde unter anderem in [Lajoie1994] erweitert. Hier verwendet man eine Kombination aus drei Vorlagen für unterschiedliche Zwecke. Die erste Vorlage bedient sich der Design-Patterns nach [Gamma1993] zur Beschreibung von Mikro-Architekturen des Frameworks. Als zweites wird eine Vorlage zur Beschreibung von Kontrakten zur Modellierung von Interaktionen zwischen Objekten verwendet, wie sie in [Helm1990] eingeführt wurden. Drittens kommt eine Vorlage zur Beschreibung so genannter »Motifs« zum Einsatz. Ein Motif ist sehr ähnlich zu Krasner & Popes Rezepten oder Johnsons Mustern. Es beschreibt eine Situation (eine Problemstellung), in der das Framework zur Lösung benutzt werden kann. In einem informellen Fließtext wird die

Situation beschrieben und die Lösung diskutiert. Außerdem kann ein Motif auf verwandte Motifs verlinken.

Eine Implementierung des Kochbuchprinzips wurde in [Schappert1995] unter dem Namen »Active Cookbook« vorgestellt. Es ist die Bezeichnung für ein interaktives Kochbuch in Form eines Hypertext Dokumentes, in dem zum Beispiel die Referenzen zwischen den Rezepten durch Links unterstützt werden. Aus heutiger Sicht entspricht ein »Active Cookbook« in etwa einer HTML Darstellung eines Kochbuches. So kann ein Benutzer leicht durch das Kochbuch navigieren. Außerdem können bestimmte Aspekte nicht nur textuell, sondern auch graphisch mit Hilfe von Diagrammen visualisiert werden. Aufbauend auf dem interaktiven Kochbuch kann auch abhängig von Benutzerentscheidungen Quellcode generiert werden, der quasi den Rumpf für eine Implementierung eines Erweiterungspunktes bildet.

Eine erweiterte Sicht auf die erwähnten Hooks verwenden Froehlich et al. in [Froehlich1997], indem sie einen Hook nicht als einzelne Methode auffassen, sondern einen Hook mit einem Hot-Spot (= Erweiterungspunkt) gleichstellen. Ein Hook ist in dieser allgemeinen Betrachtungsweise eine Stelle im Framework, an der man Anpassungen vornehmen kann. In [Froehlich1997] stellen die Autoren eine Vorlage vor, um diese Hooks zu spezifizieren. Eine Hook-Beschreibung besteht hiernach aus folgenden Feldern, die wir in Tabelle 4.5 zusammengefasst haben.

Tabelle 4.5: Vorlage zur Beschreibung von Hooks nach [Froehlich1997]

Feld	Erläuterung
Name	Ein sprechender Bezeichner.
Anforderung	Beschreibt das Problem, das mit diesem Hook gelöst werden soll.
Typ	Beschreibt die Art und Weise, wie man das Framework mit diesem Hook anpassen kann.
Anwendungs-bereich	Beschreibt die Teile des Frameworks, die mit diesem Hook in Verbindung stehen.
Benutzt	Beschreibt welche anderen Hooks benötigt werden, um diesen Hook anwenden zu können.
Teilnehmer	Beschreibt Komponenten, die bei der Umsetzung des Hooks teilnehmen.
Anwendung	Beschreibt wie der Hook verwendet werden muss und damit welche Anpassungen an teilnehmenden Komponenten notwendig sind.
Einschränkungen	Beschreibt Einschränkungen, die mit der Benutzung des Hooks einhergehen.
Kommentare	Für allgemeine Hinweise zum Hook.

Auch diese Form der Beschreibung fällt unter das Vorlagen-Prinzip und unterscheidet sich nicht von den Grundgedanken, die schon den frühen Koch-

büchern zugrunde lagen. Zur Veranschaulichung wollen wir nach der Vorlage in Tabelle 4.5 das Servlet-Beispiel aus Abschnitt 2.1.3 beschreiben. Hierzu greifen wir die Vorlage auf und füllen Sie mit den Informationen zu Servlets. Das Ergebnis zeigt Tabelle 4.6.

Tabelle 4.6: Servlet-Beispiel nach der Vorlage für Hooks aus [Froehlich1997]

Feld	Erläuterung
Name	`HTTP Servlet`
Anforderung	Eine Web-Anwendung soll nach dem Java Servlet Standard entwickelt werden.
Typ	Implementierung eines neuen Servlets.
Anwendungsbereich	Eine Servlet-Engine delegiert eingehende HTTP Anfragen gemäß eines zu konfigurierenden Musters an registrierte Servlets. Die Servlets verarbeiten die Anfrage und erzeugen eine entsprechende Antwort, die von der Servlet-Engine an den Client gesendet wird.
Benutzt	`Generic Servlet`
Teilnehmer	Die zu erstellende HTTP Servlet-Klasse sowie `HttpServletRequest` und `HttpServletResponse` Objekte.
Anwendung	Die neue HTTP Servlet-Klasse muss von der Klasse `HTTPServlet` erben und je nach benutzten HTTP Methoden die `do*` Methoden überschreiben. Zur Erzeugung der Antwort muss die neue Klasse das `HttpServletResponse` Objekt verändern und Daten in das `body` Feld dieses Objektes schreiben.
Einschränkungen	Die Verwendung dieses Servlets ist für das HTTP Protokoll implementiert. Bei Verwendung anderer Protokolle siehe `Generic Servlet`.
Kommentare	Zusätzlich zur Implementierung muss das Servlet in einer Verzeichnisstruktur für Web-Anwendung abgelegt und als WAR Archiv komprimiert werden. Außerdem muss das Servlet in der `web.xml` Datei eingetragen werden.

Aus diesem Beispiel in Tabelle 4.6 wird deutlich, dass die Möglichkeiten der Beschreibung sehr flexibel sind. Die Vorlage hilft insbesondere die wichtigen Fragen für die Dokumentation zu beantworten. Allerdings ist die Struktur wiederum so allgemein, dass unterschiedliche Benutzer die Vorlage durchaus leicht unterschiedlich benutzen könnten.

Bewertung

Wir bewerten die Verwendung informeller Framework-Beschreibungen auf Basis des vorgestellten Vorlagen-Prinzips anhand der aufgestellten Anforderungen an Framework-Beschreibungssprachen aus Kapitel 4.1. Wir beginnen mit den Anforderungen an die Anwendbarkeit in Tabelle 4.7.

Tabelle 4.7: Anwendbarkeit des Vorlagen-Prinzips

ID	Anforderung	Umsetzung mit Mustern	Bewertung
ANW-B	Eine Framework-Beschreibungssprache muss Konzepte unterstützen, die einen Benutzer bei Verwendung einer Framework-Beschreibung, d.h. beim Explorieren und Erlernen eines Frameworks unterstützen.	Vorlagen-basierte Beschreibungen unterstützen Framework-Benutzer beim Erlernen eines Frameworks durch ihren verlinkten Aufbau.	+
ANW-E	Die Erstellung der Framework-Beschreibung muss sich so in den Entwicklungsprozess des Frameworks integrieren, dass der Aufwand der Entwickler für die Erstellung möglichst gering ist.	Die Erstellung einer Framework-Beschreibung gehört zum Entwicklungsprozess. Durch die Vorgabe von Vorlagen wird diese Aufgabe standardisiert und damit erleichtert.	+
ANW-W	Die Informationen einer Framework-Beschreibung müssen für Werkzeug-Entwickler programmatisch auswertbar sein.	Die informellen Texte sind nur schwer automatisch auswertbar.	–

Wir sehen, dass sich Vorlagen eignen, wenn man mit ihrer Hilfe ein Framework erlernen muss. Die Vorgabe fester Strukturen für die Dokumentation wirkt sich positiv auf den Aufwand zur Erstellung der Framework-Beschreibung aus, da Software-Entwickler nicht überlegen müssen, wie sie etwas dokumentieren sollen. Leider lassen sich die Informationen aus einer Vorlagen-basierten Framework-Beschreibung nur sehr schlecht automatisch auswerten, da die informellen Texte keine formale Struktur aufweisen. Im Folgenden betrachten wir die Anforderungen an die Ausdrucksfähigkeit in Tabelle 4.8.

Tabelle 4.8: Ausdrucksfähigkeit des Vorlagen-Prinzips

ID	Anforderung der Framework-Benutzer Eine Framework-Beschreibungssprache muss...	Umsetzung mit Vorlagen	Bewertung
AUS-1	... Beispiele für die Benutzung des Frameworks beschreiben können.	Beispiele sind wichtiger Bestandteil bei der Beschreibung mit Mustern.	+
AUS-2	... statische und dynamische Architekturübersichten beschreiben können.	Im Freitext eines Feldes einer Vorlage lassen sich Übersichten nur in Form von Text beschreiben.	–

ID	Anforderung der Framework-Benutzer Eine Framework-Beschreibungssprache muss...	Umsetzung mit Vorlagen	Bewertung
AUS-3	... die Erweiterungspunkte eines Frameworks beschreiben können.	Erweiterungspunkte können durch Vorlagen beschrieben werden.	+
AUS-4	... die Konfiguration und Integration des Frameworks beschreiben können.	Konfigurationsparameter und Schnittstellen für die Integration können über eine Vorlage dokumentiert werden.	+
AUS-5	... die Implementierung einer Erweiterung beschreiben können.	Details der Implementierung können im Freitext wiedergegeben werden.	o
AUS-6	... das Kommunikationsprotokoll zwischen Framework und Erweiterung beschreiben können.	Die Kommunikation lässt sich etwa über Vorlagen für Kontrakte zumindest textuell beschreiben.	o
AUS-7	... die Bindung von Erweiterungen an das Framework beschreiben können.	Die Bindung kann im Freitext beschrieben werden.	o
AUS-8	... das Format und die Installation einer Erweiterung in das Framework beschreiben können.	Details zu Format und Installation können im Freitext beschrieben werden.	o
AUS-9	... einzuhaltende Bedingungen seitens einer Erweiterung beschreiben können.	Einschränkungen können im Freitext beschrieben werden.	o
AUS-10	... Varianten von Erweiterungspunkten beschreiben können.	Durch Verlinken von Vorlagen können auch Varianten ausgedrückt werden.	o

Dadurch, dass Vorlagen nur wenige Vorgaben machen welche Informationen man als Freitext aufnimmt, ist das Vorlagen-Prinzip im Hinblick auf die Ausdrucksfähigkeit natürlich sehr mächtig. Der zu zahlende Preis ist der erwähnte Mangel an automatischen Möglichkeiten zur Auswertung der Informationen. Daher können wir hier nur mittlere Bewertungen vergeben. Betrachten wir nun abschließend die Anforderungen zur Erweiterbarkeit in Tabelle 4.9.

Tabelle 4.9: Erweiterbarkeit des Vorlagen-Prinzips

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Umsetzung mit Vorlagen	Bewertung
ERW-B	... existierende Sprachen, mit denen Framework-Benutzer bereits vertraut sind, zur Beschreibung einzelner Merkmale integrieren können.	Vorlagen, die Freitext erlauben, machen keine Einschränkung welche Sprache man für einen bestimmten Aspekt verwendet. Allerdings führt diese Offenheit zu keiner Standardisierung und mangelnder Vergleichbarkeit.	O
ERW-E	... es den Framework-Entwicklern ermöglichen, die Framework-Beschreibung um neue Beschreibungsformen für Erweiterungspunkte zu erweitern.	Durch den Einsatz von Freitext können auch neue Formen von Erweiterungspunkten beschrieben werden. Allerdings führt diese Freiheit wiederum zu mangelnder Vergleichbarkeit.	O
ERW-W	... müssen in der Lage sein, Werkzeuge auf Basis der Sprachdefinition der Framework-Beschreibungssprache zu erweitern.	Vorlagen legen nur Felder fest, in denen Werte eingetragen werden können. Im Falle von Freitext-Felder gibt es keine auswertbare Sprachdefinition, die Werkzeug-Entwickler ausnutzen könnten.	–

Vorlagen sind sehr leicht erweiterbar und anpassbar, da ihre informelle Struktur so gut wie keine Einschränkungen macht. Jedoch gilt auch hier, dass es keinen Weg gibt, um diese informellen Dinge automatisch zugänglich zu machen. Insbesondere der Mangel an einer definierten Sprachdefinition erschwert die Entwicklung anpassbarer Werkzeuge.

Fazit

Vorlagen-basierte Ansätze eignen sich gut, um Dokumentationen zu erstellen, die von Menschen gelesen werden. Ein geschickter Aufbau der Vorlagen, etwa nach dem Pyramiden-Prinzip von allgemein zu speziell, wie in [Meusel1997] vorgeschlagen, erleichtert es ein Framework vom groben Überblick bis zu den Details zu durchdringen. Leider sind die erfassten Informationen kaum wiederverwendbar, da sie wenig strukturiert und nicht formal definiert sind. Damit sind sie nicht automatisch verarbeitbar.

4.2.2 SmartBooks

Die Benutzung so genannter SmartBooks [Ortigosa1999][Ortigosa1999a] ist ein Ansatz von Alvaro Ortigosa und weiteren Forschern, die den Ansatz der interaktiven Kochbücher (»Active Cookbooks«) erweitern. Hierzu machen

SmartBooks eine wichtige Unterscheidung bei der Erstellung der Framework-Beschreibung. SmartBooks unterscheiden zwischen der Dokumentation für Framework-Entwickler und für Framework-Benutzer. Sie betonen, dass für beide Benutzergruppen unterschiedliche Dokumentationen notwendig sind, da beide Benutzergruppen unterschiedliche Bedürfnisse an eine Framework-Beschreibung haben.

Die Unterscheidung von zwei Arten von Framework-Beschreibungen ist sehr wichtig und SmartBooks sind der einzige Ansatz, der diese Unterscheidung so explizit macht. Muster und Kochbücher haben ihren Schwerpunkt auf Seiten der Framework-Benutzer, aber bieten keine Unterstützung für Framework-Entwickler. Design- und Meta-Pattern, die wir im Kapitel 4.2.3 vorstellen werden, versuchen einen Spagat zwischen den Bedürfnissen beider Benutzergruppen, was allerdings zu Lasten der Framework-Benutzer geht, da diese mit zu vielen, aus Sicht des Benutzers unnötigen, Detailinformationen überfordert werden.

SmartBooks gehen davon aus, dass ein Framework-Entwickler den Aufbau des Frameworks mit Dokumentationsformen wie der UML, Design-Pattern und Beispielen z.B. durch ein interaktives Kochbuch dokumentieren. Diese Form der Dokumentation ist allerdings nicht primär für den Framework-Benutzer vorgesehen. Als Dokumentation für den Framework-Benutzer erstellt der Framework-Entwickler eine Beschreibung der Funktionalität des Frameworks, welche Komponenten für unterschiedliche Funktionalitäten zuständig sind und erstellt Regeln, die einschränken, wie ein Framework-Benutzer eigene Anpassungen auf Basis der angebotenen Funktionalität vornehmen kann. Diese Form der Dokumentation soll den Framework-Benutzer gezielt durch die Benutzung des Frameworks führen [Ortigosa1999].

Die Regeln bestimmen, wie so genannte Instantiierungs-Aufgaben (engl. instantiation tasks) zur Erreichung eines bestimmten Ziels komponiert werden können. Instantiierungs-Aufgaben bilden eine Hierarchie von Aufgaben, die ein Framework-Benutzer bei der Benutzung des Frameworks, mit anderen Worten bei der Instantiierung, abarbeiten muss. Hierzu gibt es Basisaufgaben, die zu komplexeren Aufgaben zusammengesetzt werden können. Durch den Einsatz von Handlungsplanungsalgorithmen können die notwendigen Schritte zu Erreichung eines vorgegebenen Zieles berechnet und dem Framework-Benutzer als Leitfaden bestehend aus Einzelaufgaben präsentiert werden [Ortigosa1999].

Basisaufgaben sind zum Beispiel das Erben von einer bestimmten Klasse, das Überschreiben bestimmter Methoden oder der Aufruf einer Funktion. Aufbauend auf derartigen Basisaufgaben können komplexere Benutzeraufgaben komponiert werden. Die Erstellung der Dokumentation für den Framework-Benutzer wird so zur Modellierung von Benutzeraufgaben, die beschreiben, wie Framework-Benutzer Funktionalitäten des Frameworks anpassen können. Durch die Angabe erwähnter Regeln, die auch als Instantiierungs-Regeln (engl. instantiation rules) bezeichnet werden, wird festgelegt, wie Aufgaben komponiert werden können. Auf Basis dieser Grundlage können dann automatisch Leitfäden für den Framework-Benutzer abgeleitet werden [Ortigosa1999].

Die automatische Planung der Aufgaben basiert darauf, dass jede Regel ein Ziel beschreibt, das bei Erfüllung einer Bedingung erreicht werden kann. Der

Planungsalgorithmus erhält als Eingabe die zu erreichenden Ziele und wählt Regeln entsprechend ihrer Bedingungen aus, um die Ziele zu erreichen. Jede Regel besteht aus einem Ziel, der so genannten Nachbedingung, die gilt, wenn bestimmte Vorbedingungen eingetreten sind. Schematisch wird eine Regel wie folgt ausgedrückt:

Liste mit Vorbedingungen ← Nachbedingung (Ziel)

In jedem Schritt versucht der Planungsalgorithmus die Vorbedingungen zu erfüllen, um eine angestrebte Nachbedingung als Ziel zu erfüllen. Wir wollen dieses Prinzip am Beispiel der Erstellung eines Servlets aus Abschnitt 2.1.3 verdeutlichen. Die folgenden Regeln definieren welche Schritte notwendig sind, um ein neues Servlet zu erstellen.

1. addServlet ← `initialFunctionality`(„Create new Servlet“)
2. defineComponent(„Servlet“), bindComponent(„Servlet“), deployComponent(„Servlet“) ← addServlet
3. `pendingTask`(„DefineClass“, [NewClass, Component]), `pendingTask`(„DocumentClass“, [NewClass]), `option`[inheritFromClass(„HttpServlet“, [NewClass]), inheritFromClass(„GenericServlet“, [NewClass])] ← defineComponent(Component)
4. `pendingTask`(„ImplementAbstractMethods“, [SuperClass, Class]) ← inheritFromClass(SuperClass, Class)
5. `pendingTask`(„AddServletToConfig“, [Component]) ← bindComponent(Component)
6. `pendingTask`(„CreateWebARchive“) ← deployComponent(Component)

Die Planung der Aufgaben beginnt mit dem `initialFunctionality` Ziel. Hierzu muss die Vorbedingung addServlet erfüllt werden. Zur Erfüllung dieser Vorbedingung wiederum müssen die drei Vorbedingungen aus Regel 2 erfüllt werden. Diese sind defineComponent, bindComponent und deployComponent.

Die Regel 3 für defineComponent verlangt die Erfüllung zweier `pendingTasks`. Bei einem `pendingTask` wartet der Aufgabenmanager so lange bis der Benutzer signalisiert hat, dass die Aufgabe abgeschlossen wurde. Hier müssen die beiden Aufgaben „DefineClass“ und „DocumentClass“ erledigt werden. Zusätzlich kann der Benutzer nun wählen (`option`) ob er für die neue Klasse von „HttpServlet“ oder von „GenericServlet“ gemäß der Bedingung inheritFromClass erben möchte. In den Regeln 4, 5 und 6 werden die Bedingungen inheritFromClass, bindComponent und deployComponent ebenfalls auf `pendingTasks` zurückgeführt, die vom Benutzer zu erfüllen sind. So würde der Planungsalgorithmus zum Beispiel folgenden Leitfaden für die Erstellung eines Servlets ausgeben.

Listing 4.1: Leitfaden für die Erstellung eines Servlets gemäß [Ortigosa1999]

```
1  DefineClass
2  DocumentClass
3  ImplementAbstractMethods from „HttpServlet“
4  AddServletToConfig
5  CreateWebARchive
```

Eine solche Aufgabenliste, die hier aus Gründen der Übersichtlichkeit nur wenige Informationen enthält, aber leicht erweitert werden kann, hilft einem Framework-Benutzer direkt zur Erreichung seines Zieles. Durch den Einbau von Alternativen können unterschiedliche Pfade zum Ziel eingeschlagen werden, wobei das Planungssystem die Aufgabenliste dynamisch anpassen kann.

Verwandt zur Idee von SmartBooks ist das in [Butler2000] vorgestellte Framework zur Dokumentation von Frameworks. Hier geht man von bestimmten Anwendungsfällen für eine Wiederverwendung aus und drückt diese durch eine Menge an elementaren Entwicklungsaufgaben für einen Framework-Benutzer aus. Diese elementaren Entwicklungsaufgaben werden in der Framework-Beschreibung erfasst, so dass die gesamte Beschreibung die Gesamtmenge der elementaren Entwicklungsaufgaben ist. Je nach Wiederverwendung können so die benötigten Entwicklungsaufgaben aus der Dokumentation berechnet werden.

Bewertung

Nachdem wir den SmartBook Ansatz vorgestellt haben, wollen wir ihn gemäß unserer aufgestellten Anforderungen bewerten. Wir beginnen mit der Bewertung der Anwendbarkeit von SmartBooks in Tabelle 4.10.

Tabelle 4.10: Anwendbarkeit von SmartBooks

ID	Anforderung	Umsetzung mit SmartBooks	Bewertung
ANW-B	Eine Framework-Beschreibungssprache muss Konzepte unterstützen, die einen Benutzer bei Verwendung einer Framework-Beschreibung, d.h. beim Explorieren und Erlernen eines Frameworks unterstützen.	Ein SmartBook leitet den Benutzer zu seinem Ziel, aber das Ziel muss vom Benutzer formuliert werden. SmartBooks geben nicht vor wie der Benutzer lernt welche Ziele man überhaupt formulieren kann.	—
ANW-E	Die Erstellung der Framework-Beschreibung muss sich so in den Entwicklungsprozess des Frameworks integrieren, dass der Aufwand der Entwickler für die Erstellung möglichst gering ist.	Die Modellierung der Instantiierungs-Regeln bedeutet zusätzlichen Aufwand. Es muss dabei sichergestellt sein, dass die Regeln in sich konsistent sind, was nicht immer trivial ist.	—

ID	Anforderung	Umsetzung mit SmartBooks	Bewertung
ANW-W	Die Informationen einer Framework-Beschreibung müssen für Werkzeug-Entwickler programmatisch auswertbar sein.	Die formellen Regeln sind für eine automatische Auswertung erstellt worden.	+

Es zeigt sich, dass die automatisch erzeugten Leitfäden eine Hilfe für den Framework-Benutzer darstellen können, aber um einen Leitfaden erzeugen zu können, muss der Benutzer zunächst verstehen welche Ziele überhaupt erreicht werden können. Der SmartBook Ansatz erklärt nicht, wie ein Framework-Benutzer ein unbekanntes Framework mit Hilfe des Ansatzes erlernen kann. Hierzu wird man vermutlich auf andere verfügbare Dokumentationsquellen, wie UML-Diagramme und Beispiele, die nicht Teil des SmartBook Ansatzes sind, zurückgreifen müssen. SmartBook Leitfäden sind erst sinnvoll anwendbar, wenn der Benutzer bereits sein Ziel kennt.

Darüber hinaus ist die Erstellung der Instantiierungs-Regeln keine triviale Aufgabe. Die Regeln müssen aufeinander aufbauen und in sich konsistent sein. Falls die Regeln keine innere Konsistenz hätten, würde der Planungsalgorithmus vermutlich in Sackgassen landen und dem Benutzer nicht helfen können. Wie diese Konsistenz sichergestellt wird und wie man systematisch Regeln aufbauen kann, wird im Ansatz nicht näher beschrieben. Aus Sicht der Anwendbarkeit bedeutet es in jedem Fall einen Mehraufwand bei der Erstellung.

Auf der anderen Seite sind die Regeln ideal um automatisch ausgewertet werden zu können, denn hierzu wurden sie spezifiziert. Es sollte daher möglich sein, die Unterschiede zwischen Instantiierungs-Regeln, die auftreten, wenn man zwei unterschiedliche Framework-Versionen betrachtet, berechnen zu können. Diese Unterschiede wiederum könnten Rückschlüsse auf Veränderungen am Framework zulassen. Betrachten wir nun die Ausdrucksfähigkeit von SmartBooks in Tabelle 4.11.

Tabelle 4.11: Ausdrucksfähigkeit von SmartBooks

ID	Anforderung der Framework-Benutzer Eine Framework-Beschreibungssprache muss...	Umsetzung mit SmartBooks	Bewertung
AUS-1	... Beispiele für die Benutzung des Frameworks beschreiben können.	Beispiele sind nicht Teil eines SmartBooks.	–
AUS-2	... statische und dynamische Architekturübersichten beschreiben können.	Derartige Übersichten lassen sich nicht in Regeln ausdrücken.	–

ID	Anforderung der Framework-Benutzer Eine Framework-Beschreibungssprache muss...	Umsetzung mit SmartBooks	Bewertung
AUS-3	... die Erweiterungspunkte eines Frameworks beschreiben können.	Die Benutzung eines Erweiterungspunktes ist ein Ziel, das mit Regeln ausgedrückt werden kann.	+
AUS-4	... die Konfiguration und Integration des Frameworks beschreiben können.	Konfigurationsparameter und die Integration des Frameworks sind nicht Teil des Regelwerks.	–
AUS-5	... die Implementierung einer Erweiterung beschreiben können.	Die einzelnen Aufgaben eines Leitfadens geben dem Benutzer vor, was für eine konkrete Implementierung zu tun ist.	+
AUS-6	... das Kommunikationsprotokoll zwischen Framework und Erweiterung beschreiben können.	Es können Regeln formuliert werden, die das einzuhaltende Protokoll beschreiben.	+
AUS-7	... die Bindung von Erweiterungen an das Framework beschreiben können.	Die Erstellung der Bindung kann durch Regeln ausgedrückt werden.	+
AUS-8	... das Format und die Installation einer Erweiterung in das Framework beschreiben können.	Die Erstellung der Installation kann durch Regeln ausgedrückt werden.	+
AUS-9	... einzuhaltende Bedingungen seitens einer Erweiterung beschreiben können.	Regeln für Einschränkungen sind eingeschränkt möglich.	o
AUS-10	... Varianten von Erweiterungspunkten beschreiben können.	Durch die Verwendung von Alternativen in den Regeln können Varianten eingeschränkt umschrieben werden.	o

SmartBooks besitzen eine sehr hohe Ausdrucksfähigkeit, wie man bereits am Servlet-Beispiel erkennen konnte. Mit ihnen lassen sich gut die einzelnen Schritte modellieren, die man zur Implementierung einer Erweiterung benötigt. Ihre Schwäche haben SmartBooks, wenn es um den Aspekt einer architektonischen Sicht auf das Framework geht. Außerdem sind konkrete Beispiele zur Veranschaulichung nicht vorgesehen. Betrachten wir nun abschließend die Erweiterbarkeit von SmartBooks in Tabelle 4.12.

Tabelle 4.12: Erweiterbarkeit von SmartBooks

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Umsetzung mit SmartBooks	Bewertung
ER W-B	... existierende Sprachen, mit denen Framework-Benutzer bereits vertraut sind, zur Beschreibung einzelner Merkmale integrieren können.	Es ist nicht vorgesehen existierende Sprachen wiederzuverwenden. Stattdessen müssen Framework-Benutzer alles in Regeln ausdrücken.	–
ER W-E	... es den Framework-Entwicklern ermöglichen, die Framework-Beschreibung um neue Beschreibungsformen für Erweiterungspunkte zu erweitern.	Neue Beschreibungsformen können durch neue Arten von Regeln und Aufgaben ausgedrückt werden. SmartBooks können jedoch nicht beliebig und nur durch neue Regeln erweitert werden.	o
ER W-W	... müssen in der Lage sein, Werkzeuge auf Basis der Sprachdefinition der Framework-Beschreibungssprache zu erweitern.	Die formale Struktur der Regeln ermöglicht es Werkzeuge zu entwickeln, die auf Basis sich ändernder Regeln auch erweiterbar sind.	+

SmartBooks sind durch neue oder erweiterte Regeln anpassbar. Allerdings sind Framework-Benutzer immer an die formale Regelstruktur gebunden. Es ist zu beachten, ob Erweiterungen Auswirkungen auf die automatische Auswertung der Informationen in Werkzeugen haben. Prinzipiell sollte es jedoch gut möglich sein, erweiterbare Werkzeuge auf der formalen Basis der Regeln zu entwickeln.

Fazit

SmartBooks stehen für die automatische Erstellung von Leitfäden, um ein bestimmtes Ziel im Sinne der Benutzung eines Frameworks zu erreichen. Damit die Leitfäden je nach Ziel automatisch abgeleitet werden können, ist ein relativ kompliziertes Regelwerk notwendig, deren Erstellung dem Framework-Entwickler auferlegt wird. Es ist nicht klar, wie komplex Regelwerke für realistische Frameworks werden können und wie die Konsistenz der Regeln gewahrt wird. Es ist in jedem Fall mit einem erhöhten Aufwand für den Framework-Entwickler zu rechnen. Das Konzept der Regeln ist ausdrucksstark, da leicht neue oder erweiterte Regeln aufgebaut werden können. Die Schwäche dieses Ansatzes liegt in der fehlenden architektonischen Betrachtung des Frameworks aus abstrakter Sicht. Die Anwendung der Regeln wird erst dann sinnvoll, wenn ein Framework-Benutzer das zu erreichende Ziel formulieren kann. Hierzu muss er jedoch zunächst das Konzept des Frameworks verstanden haben.

4.2.3 Design- und Meta-Pattern

Design-Pattern beschreiben Mikroarchitekturen [Gamma1993] und helfen wiederkehrende programmiertechnische Design-Probleme zu lösen. Ein Katalog solcher Design-Pattern wurde von der so genannten »Gang of Four«, als Anspielung auf die vier Autoren, in [Gamma1995] beschrieben. Ein Design-Pattern wird nach [Gamma1993] beschrieben durch:

1. eine abstrakte Beschreibung einer Klassen- oder Objektstruktur
2. das Design-Problem, das durch diese abstrakte Struktur angegangen wird
3. die Konsequenzen, die sich aus der Verwendung der abstrakten Struktur in einer Architektur ergeben

Zur Dokumentation eines Design-Pattern greift man auf einen Vorlagen-basierten Ansatz zurück. In [Gamma1993] wird folgende Vorlage für die Dokumentation von Design-Pattern vorgeschlagen.

Tabelle 4.13: Vorlage zur Beschreibung von Design-Pattern nach [Gamma1993]

Feld	Erläuterung
Design-Pattern Name	Ein sprechender Bezeichner.
Intention	Beschreibt das zu lösende Problem und das Ziel, das man mit der Anwendung des Design-Pattern verfolgt.
Motivation	Beschreibt ein Szenario, in dem das Design-Pattern gut angewendet werden kann, um die Intention besser zu verstehen.
Anwendbarkeit	Beschreibt wann das Design-Pattern angewendet werden kann.
Teilnehmer	Beschreibt die Klassen und Objekte und deren Rollen im Design-Pattern.
Zusammenarbeit	Beschreibt die Interaktion zwischen den teilnehmenden Klassen und Objekten.
Diagramm	Ein Klassen- oder Objektdiagramm, dass die strukturelle Beziehung zwischen teilnehmenden Klassen und Objekten zeigt.
Konsequenzen	Beschreibt wie die Ziele des Design-Pattern erreicht werden und mögliche Auswirkungen, die zu beachten sind.
Implementierung	Gibt Hinweise auf mögliche Implementierungen des Design-Pattern.
Beispiele	Zeigt einige Beispiele für die Anwendung des Design-Pattern.
Siehe auch	Verweise zu verwandten Design-Pattern.

Die Verwendung von Design-Pattern in Bibliotheken soll unter anderem zu einer reduzierten Lernzeit für die Verwendung der Bibliothek führen. Aus

[Gamma1993] entnehmen wir: „Design patterns help reduce the learning time for a class library.“ Wenn also die Lernzeit von Bibliotheken verringert werden kann, so könnte dies auch für Frameworks gelten. Durch die Verwandtschaft von Bibliotheken und Frameworks ist es daher naheliegend, Design-Pattern auch für Framework-Beschreibungen einzusetzen. Der bereits beschriebene »Motif« Ansatz [Lajoie1994] ist ein Beispiel dafür. Allerdings sind Design-Pattern nicht speziell für die Dokumentation von Frameworks entwickelt worden, sondern liefern abstrakte Design-Ratschläge für programmiertechnische Design-Probleme. Manche Forscher sind der Meinung, dass man zunächst die abstrakten Design-Pattern lernen solle, um dann ein Framework leichter verstehen zu können, das diese Design-Pattern einsetzt. Dem lässt sich entgegenhalten, dass man hierfür sehr viele Details der Framework-Implementierung betrachten und verstehen müsste. Diese Details, sind für einen Framework-Benutzer jedoch relativ uninteressant, wenn er das Framework nur benutzen möchte. Aus dieser Motivation wurden von Pree so genannte Meta-Pattern entwickelt [Pree1994]. Meta-Pattern sind Sammlungen von Design-Pattern, die beschreiben wie man ein Framework unabhängig von einer Domäne konstruiert.

> *We introduce the term meta patterns for a set of design patterns that describes how to construct frameworks independent of a specific domain.*
>
> **Meta-Pattern nach [Pree1994]**

Die beschriebenen Meta-Pattern sind sehr eng verwandt mit dem später erschienenen UML Profil für Frameworks, genannt UML-F, dem wir uns ab Seite 103 genauer widmen werden. In [Pree1994] wird das Konzept der Template- und Hook-Methoden als Meta-Pattern aufgefasst, das verwendet werden kann, um Frameworks zu spezifizieren. So kann man mit Hilfe des Meta-Pattern für Template- und Hook-Methoden die Erweiterungsmöglichkeiten eines Frameworks spezifizieren.

In [Richner1998] gehen die Autoren darauf ein, warum eine Framework-Beschreibung nicht nur mit Design-Patterns erfolgen kann. Sie kommen zu dem Schluss, dass (zusätzlich) eine Beschreibung der Architektur des Frameworks notwendig ist. Sie stützen ihre Argumentation auf vier Anforderungen, die nach Meinung der Autoren eine Framework-Beschreibung erfüllen muss [Richner1998]. Auch groß angelegte Studien wie in [Kirk2007] kommen zu dem Schluss, dass Design-Pattern und die Beschreibung von Mikro-Architekturen nur bedingt helfen, um insbesondere größere Zusammenhänge im Framework zu dokumentieren.

- *'selling' the framework*
  Benutzer wollen schnell verstehen wozu ein Framework eingesetzt werden kann und welchen Zweck es erfüllt.

- *reuse of architecture*
  Eine abstrakte Sicht auf die Architektur des Frameworks erleichtert es, dieses im Hinblick auf seine Wiederverwendung zu beurteilen.

- *integration of frameworks*
  Jedes Framework muss in den Kontext eines größeren Systems eingebettet werden. Eine Architekturbeschreibung erleichtert diese Aufgabe.

- *evolution and re-engineering*
  Eine Architekturbeschreibung ermöglicht es nachfolgende Versionen eines Frameworks mit der aktuellen zu vergleichen. So können Änderungen erkannt und nachvollzogen werden.

Würde man nur mit Design- und Meta-Pattern arbeiten, könnte man diese Anforderungen, die sich auch in den Anforderungen in dieser Arbeit wiederfinden, nicht erfüllen. Auf ein Beispiel wollen wir an dieser Stelle verzichten und verweisen auf das Beispiel zur Verwendung der UML-F in Kapitel 4.2.4 ab Seite 102, da dort die Konzepte der Design- und Meta-Pattern integriert wurden. Im Folgenden wollen wir die Möglichkeiten der Design- und Meta-Pattern in Bezug auf die gestellten Anforderungen dieser Arbeit bewerten.

Bewertung

Für unsere Bewertung betrachten wir zunächst die Anwendbarkeit von Design-Pattern und Meta-Pattern zur Framework-Beschreibung.

Tabelle 4.14: Anwendbarkeit von Design- und Meta-Pattern

ID	Anforderung	Umsetzung mit Pattern	Bewertung
AN W-B	Eine Framework-Beschreibungssprache muss Konzepte unterstützen, die einen Benutzer bei Verwendung einer Framework-Beschreibung, d.h. beim Explorieren und Erlernen eines Frameworks unterstützen.	Auf einem niedrigen Abstraktionsniveau helfen Design-Pattern den Aufbau eines Frameworks zu verstehen. Ein Benutzer erhält jedoch nur schwer einen Überblick über das Framework.	–
AN W-E	Die Erstellung der Framework-Beschreibung muss sich so in den Entwicklungsprozess des Frameworks integrieren, dass der Aufwand der Entwickler für die Erstellung möglichst gering ist.	Da das Framework gezielt Design-Pattern verwendet, ist der Aufwand diese an den verwendeten Stellen zu dokumentieren vertretbar. Die Struktur der Design-Pattern hilft die Dokumentation zu strukturieren.	+
AN W-W	Die Informationen einer Framework-Beschreibung müssen für Werkzeug-Entwickler programmatisch auswertbar sein.	Design-Pattern werden nach bestimmten Vorlagen (siehe Tabelle 4.13) dokumentiert. Eine automatische Auswertung ist so nur sehr eingeschränkt möglich.	–

Ein Manko der Design-Pattern ist ihr niedriges Abstraktionsniveau, so dass sie kaum dazu geeignet sind, um größere Zusammenhänge des Frameworks zu

dokumentieren. Ihre Stärken liegen eher in der technischen Dokumentation einzelner Erweiterungspunkte. Jedoch ist diese Art der Dokumentation für einen Framework-Benutzer nur dann sinnvoll, wenn unnötige Details der Implementierung weggelassen werden. Design-Pattern wollen aber gerade die Details einer Implementierung erfassen, so dass es hier zu einem Konflikt in der Anwendungsweise der Beschreibung mit Design-Pattern kommt.

Positiv ist, dass sich die Erstellung der Dokumentation gut in den Entwicklungsprozess integrieren lässt, wenn in der Entwicklung Design-Pattern eingesetzt werden. Die automatische Auswertbarkeit ist eingeschränkt gegeben. Sie hängt von der Art der verwendeten Vorlage ab, mit dem die Design-Pattern beschrieben werden. Betrachten wir nun in Tabelle 4.15 die Ausdrucksfähigkeit von Design-Pattern im Detail.

Tabelle 4.15: Ausdrucksfähigkeit von Design- und Meta-Pattern

ID	Anforderung der Framework-Benutzer Eine Framework-Beschreibungssprache muss...	Umsetzung mit Pattern	Bewertung
AUS-1	... Beispiele für die Benutzung des Frameworks beschreiben können.	Design-Pattern sehen die Beschreibung von Beispielen explizit vor. Allerdings beziehen sich diese Beispiele nicht unbedingt auf eine globale Sichtweise zur Benutzung des gesamten Frameworks.	o
AUS-2	... statische und dynamische Architekturübersichten beschreiben können.	Hier sind Design-Pattern weniger geeignet, da diese speziell Strukturen in Mikroarchitekturen beschreiben.	–
AUS-3	... die Erweiterungspunkte eines Frameworks beschreiben können.	Design-Pattern können Details eines Erweiterungspunktes beschreiben, sind aber weniger geeignet, um einen Überblick über alle verfügbaren Erweiterungspunkte zu erhalten.	–
AUS-4	... die Konfiguration und Integration des Frameworks beschreiben können.	Konfigurationsparameter und Schnittstellen für die Integration können Teil einiger Design-Pattern sein, so dass dies eingeschränkt berücksichtigt ist.	o
AUS-5	... die Implementierung einer Erweiterung beschreiben können.	Programmiertechnische Details eines Erweiterungspunktes lassen sich mit Design-Pattern beschreiben.	+
AUS-6	... das Kommunikationsprotokoll zwischen Framework und Erweiterung beschreiben können.	Ein Design-Pattern beschreibt auch die Interaktion der teilnehmenden Klassen, so dass auch die Kommunikation beschrieben werden kann. Allerdings ist dies nicht formal festgelegt.	o

ID	Anforderung der Framework-Benutzer Eine Framework-Beschreibungssprache muss...	Umsetzung mit Pattern	Bewertung
AUS-7	... die Bindung von Erweiterungen an das Framework beschreiben können.	Hierzu kann es Informationen bei den Hinweisen zur Implementierung eines Design-Pattern geben. Nicht explizit vorgesehen.	–
AUS-8	... das Format und die Installation einer Erweiterung in das Framework beschreiben können.	Hierzu kann es Informationen bei den Hinweisen zur Implementierung eines Design-Pattern geben. Nicht explizit vorgesehen.	–
AUS-9	... einzuhaltende Bedingungen seitens einer Erweiterung beschreiben können.	Diese können als Konsequenzen der Anwendung eines Design-Pattern im Freitext beschrieben werden.	O
AUS-10	... Varianten von Erweiterungspunkten beschreiben können.	Verschiedene Varianten können getrennt beschrieben und mit Verweisen verknüpft werden. So sind Alternativen eingeschränkt formulierbar.	O

Design-Pattern haben ihre Stärken logischerweise in den Bereichen, für die sie entworfen wurden, nämlich zur Beschreibung von Mikroarchitekturen und programmiertechnischen Design-Problemen. Bei der Implementierung von Template- und Hook-Methoden befindet man sich exakt auf dieser Ebene, so dass Design-Pattern hier hilfreich sind. Für abstraktere Sichten auf das Framework oder die Beschreibung spezifischer Eigenschaften, wie die Bindung und Installation von Erweiterungen, sind sie weniger gut geeignet.

Betrachten wir nun die Erweiterbarkeit des Ansatzes der Design- und Meta-Pattern gemäß unserer Anforderungen in Tabelle 4.16.

Tabelle 4.16: Erweiterbarkeit von Design- und Meta-Pattern

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Umsetzung mit Pattern	Bewertung
ERW-B	... existierende Sprachen, mit denen Framework-Benutzer bereits vertraut sind, zur Beschreibung einzelner Merkmale integrieren können.	Da Design-Pattern Vorlagen-basiert beschrieben werden, können innerhalb der Vorlage auch beliebige Sprachen eingesetzt werden. Allerdings kann hierdurch die Vergleichbarkeit eingeschränkt sein.	O

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Umsetzung mit Pattern	Bewertung
ERW-E	... es den Framework-Entwicklern ermöglichen, die Framework-Beschreibung um neue Beschreibungsformen für Erweiterungspunkte zu erweitern.	Design-Pattern haben ihren Fokus auf der Beschreibung von Mikroarchitekturen. Es sind keine Erweiterungsmöglichkeiten vorgesehen, die andere Beschreibungsformen vorsehen.	–
ERW-W	... müssen in der Lage sein, Werkzeuge auf Basis der Sprachdefinition der Framework-Beschreibungssprache zu erweitern.	Da es keine formale Sprachdefinition für Design-Pattern gibt, ist die Erstellung und insbesondere Erweiterung von Werkzeugen schwierig.	–

Design-Pattern werden mit Hilfe von Vorlagen beschrieben, in denen leicht andere Sprachen verwendet werden können. Allerdings ist nicht klar, wie sich dies auf die automatische Auswertbarkeit der Dokumentation auswirkt. Es existiert eine ähnlich hohe Flexibilität wie bei Verwendung von Mustern, zum Preis einer weniger formalen Beschreibung. Um eine bessere Formalisierung zu erreichen, entstanden Ansätze, die Design-Pattern zum Beispiel mit Hilfe einer erweiterten UML zu beschreiben [Fontoura2001]. Diese Herangehensweise führte zu UML-basierten Beschreibungssprachen wie der UML-F oder F-UML, die wir im nachfolgenden Abschnitt betrachten werden.

Fazit

Design-Pattern haben ihre Stärken in der Spezifikation von Mikroarchitekturen und damit im Bereich der Template- und Hook-Methoden. Aus Sicht des Framework-Benutzers offenbaren sie jedoch zu viele Details der Implementierung, die für einen Benutzer wenig von Interesse sind. Diese Details sind für einen Framework-Entwickler wiederum durchaus von Interesse, wenn dieser Interna des Frameworks verstehen muss. Für einen Framework-Benutzer jedoch abstrahieren Design-Pattern nicht in geeigneter Weise, wie man es für die Beschreibung der Benutzung eines Frameworks benötigen würde.

4.2.4 UML-basierte Framework-Beschreibungen

In diesem Abschnitt betrachten wir UML-basierte Sprachen zur Beschreibung von Frameworks. Die UML bietet im Gegensatz zu den bisher vorgestellten Verfahren eine formalere Grundlage für die Beschreibung, was einer automatischen Verarbeitung entgegen kommt. Wir betrachten hier die zwei UML Profile UML-F und F-UML. Diese Profile erweitern die UML um eigene Stereotypen, die man als Annotationen an Methoden oder Klassen verwenden kann, um eine

bestimmte Semantik zu transportieren. Durch diese Stereotypen können so framework-spezifische Eigenschaften in UML-Modellen ausgedrückt werden.

UML-F : The UML Profile for Framework Architectures

Im Jahr 2000 veröffentlichten Marcus Fontoura, Wolfgang Pree und Bernhard Rumpe einen Aufsatz mit dem Titel »UML-F: A Modeling Language for Object-Oriented Frameworks« [Fontoura2000]. In diesem Aufsatz verknüpften sie Forschungsarbeiten zur UML mit den Konzepten zur Framework-Beschreibung und schlugen als Ergebnis ein UML-Profil zur Framework-Modellierung namens »UML-F« vor. Zwei Jahre später erschien von diesen drei Autoren das Buch »The UML Profile for Framework Architectures« [Fontoura2002], in dem sie eine weiterentwickelte Version des UML-F Ansatzes ausführlich beschreiben. Die folgenden Anmerkungen beziehen sich im speziellen auf das später erschienene Buch.

Das UML-Profil UML-F erweitert die UML um einige Konstrukte, die es erlauben Variationspunkte (engl. variation points) in Klassen- und Zustandsdiagrammen explizit zu kennzeichnen. UML-F entstand, als die UML in Version 1.4 verfügbar war, so dass einige Erweiterungen, die mit UML 2.0 eingeführt wurden, von UML-F nicht berücksichtigt sind.

Das Konzept der Variationspunkte geht zurück auf die Beschreibung so genannter »Hot Spots« in Application Frameworks [Pree1994], [Pree1995], [Pree2000]. Ein Hot-Spot ist ein Variationspunkt in einem Framework, also ein Punkt im Framework, an dem ein Framework-Benutzer seine eigene Implementierung einhängen kann, die das Framework erweitert und zu einer vollständigen Anwendung macht.

In UML-F können Variationspunkte durch zusätzliche Angaben in Klassendiagrammen gekennzeichnet werden. Das Profil definiert eine Reihe von Stereotypen für UML-Elemente wie Klassen und Methoden, die logisch ausdrücken, ob ein Element eine bestimmte variable Eigenschaft hat. Die folgende Tabelle 4.17 gibt einen Überblick über die in UML-F verfügbaren Stereotypen und deren Bedeutung.

Tabelle 4.17: Stereotypen für Variationspunkte nach UML-F

Stereotyp	Element	Beschreibung
«application»	Klasse, Paket, Schnittstelle	Das Element gehört nicht zum Framework, sondern ist Teil der Anwendung, die das Framework verwendet.
«framework»	Klasse, Paket, Schnittstelle	Das Element ist Teil des Frameworks.
«utility»	Klasse, Paket, Schnittstelle	Das Element gehört zu einer Werkzeugbibliothek oder zur Laufzeitumgebung und ist selbst nicht Teil des Frameworks oder der Anwendung.

Stereotyp	Element	Beschreibung
«fixed»	Klasse, Methode, Vererbung	Das Element ist fixiert und darf in Unterklassen nicht überschrieben werden. Bei Vererbungen dürfen keine weiteren Erben hinzugefügt werden.
«adapt-static»	Schnittstelle, Klasse, Methode, Vererbung	Das Element kann zur Übersetzungszeit durch Vererbung oder Schnittstellenimplementierung angepasst werden. Neue Attribute oder Methoden dürfen hinzugefügt, bestehende überschrieben werden. Zur Ausführungszeit ist das Element fixiert.
«adapt-dyn»	Schnittstelle, Klasse, Methode, Vererbung	Das Element kann während der Laufzeit z.B. durch dynamisches Laden von Unterklassen angepasst werden. Typischerweise überschreiben diese Unterklassen bestehende Methoden.

Neben diesen Stereotypen erweitert UML-F auch die Möglichkeiten von Sequenzdiagrammen, um z.B. optionale Teile oder Schleifen ausdrücken zu können. Mit der Einführung der Combined-Fragments in UML 2.0 [OMG2005] sind diese UML-F Erweiterungen obsolet geworden. Ebenso macht UML-F einige Vorschläge, um die Darstellung in Klassendiagrammen zu verbessern. Die Erweiterungen sind jedoch für die Beschreibung von Frameworks nicht direkt notwendig, sondern eher kosmetischer Natur, um die Lesbarkeit von Diagrammen zu verbessern.

Zur Spezifikation von Erweiterungspunkten definiert die UML-F auf Basis der allgemeinen Stereotypen in Tabelle 4.17 weitere Stereotypen, die die Auszeichnung von Template- und Hook-Methoden unterstützen. Zur Erinnerung: Eine Template-Methode bezeichnet eine Methode im Framework, die Hook-Methoden aufruft. Hook-Methoden sind Methoden, die von Erweiterungen implementiert werden, um so das Framework durch neue Funktionalität zu erweitern. UML-F unterscheidet hierbei zwei Varianten bei der Verwendung von Template- und Hook-Stereotypen.

1. *Unification Tags*
   Hierbei werden Template- und Hook-Methoden gemeinsam in derselben Klasse des Frameworks definiert. Um die Hook-Methoden anzupassen erbt man von dieser Klasse und überschreibt die Hook-Methode.

2. *Separation Tags*
   Hierbei befinden sich die aufgerufenen Hook-Methoden und Definition der Template-Methode in unterschiedlichen Klassen. In dieser Variante können Hook-Methoden durch Komposition angepasst werden. Hierzu komponiert man die Klasse, in der die angepassten Hook-Methoden implementiert sind, mit der Klasse, in der die Template-Methoden definiert sind.

Die nachfolgende Tabelle 4.18 listet die in UML-F verfügbaren Stereotypen für Template- und Hook-Methoden mit ihren unterschiedlichen Ausprägungen auf und erläutert diese kurz.

Tabelle 4.18: Stereotypen für Template- und Hook-Methoden in UML-F

Stereotyp	Element	Beschreibung
«template»	Klasse, Methode	Kennzeichnet eine Methode als Template-Methode bzw., dass eine Klasse Template-Methoden enthält. Dieser Stereotyp erweitert den Stereotypen «fixed».
«hook»	Klasse, Methode	Kennzeichnet eine Methode als Hook-Methode bzw., dass eine Klasse Hook-Methoden enthält. Dieser Stereotyp ist eine Erweiterung von «adapt-static» bzw. «adapt-dyn».
«Unif-TH»	Klasse	Kennzeichnet, dass eine Klasse sowohl Template- als auch zugehörige Hook-Methoden implementiert.
«Unif-t»	Methode	Kennzeichnet eine Methode als Template-Methode, wobei zugehörige Hook-Methoden ebenfalls innerhalb der Klasse definiert sind. Dieser Stereotyp ist eine Erweiterung von «template».
«Unif-h»	Methode	Kennzeichnet eine Methode als Hook-Methode, wobei die aufrufende Template-Methode ebenfalls innerhalb der Klasse definiert ist. Dieser Stereotyp ist eine Erweiterung von «hook».
«Sep-T: ID»	Klasse	Kennzeichnet, dass eine Klasse Template-Methoden enthält, wobei die aufgerufenen Hook-Methoden nicht innerhalb dieser Klasse definiert sind. ID ist ein frei wählbarer Bezeichner, um die separierten Templates und Hooks zuordnen zu können.
«Sep-H: ID»	Klasse	Kennzeichnet, dass eine Klasse Hook-Methoden enthält, wobei die aufrufenden Template-Methoden nicht innerhalb dieser Klasse definiert sind. ID ist ein frei wählbarer Bezeichner, um die separierten Templates und Hooks zuordnen zu können.
«Sep-t: ID»	Methode	Kennzeichnet eine Methode als Template-Methode, deren aufgerufene Hook-Methoden in einer anderen Klasse definiert sind. Dieser Stereotyp ist eine Erweiterung von «template». ID ist ein frei wählbarer Bezeichner, um die separierten Templates und Hooks zuordnen zu können.
«Sep-h: ID»	Methode	Kennzeichnet eine Methode als Hook-Methode, deren aufrufende Template-Methode in einer anderen Klasse definiert ist. Dieser Stereotyp ist eine Erweiterung von «hook». ID ist ein frei wählbarer Bezeichner, um die separierten Templates und Hooks zuordnen zu können.

Aufbauend auf den hier vorgestellten Stereotypen lassen sich mit UML-F nun spezifischere Stereotypen erstellen. So gibt es zum Beispiel Stereotypen, um die Template- und Hook-Methoden und ihre Rollen in den Design-Patterns nach [Gamma1995] zu spezifizieren [Fontoura2002].

Im Folgenden werden wir den UML-F Ansatz auf seine Anwendbarkeit und Vollständigkeit im Hinblick auf die Framework-Merkmale in Kapitel 2.3 prüfen, indem wir versuchen das in Abschnitt 2.1.3 beschriebene Beispiel zur Servlet-Implementierung mit UML-F zu spezifizieren. Hierzu nehmen wir noch einmal die drei Schritte auf, die wir für die Implementierung eines Servlets benötigen:

1. Implementierung der Servlet-Schnittstelle aus Abbildung 2.7
2. Anlage der Verzeichnisstruktur gemäß Listing 2.1
3. Anlage bzw. Erweiterung einer `web.xml` als Instanz des Schemas in Listing 2.2

Im ersten Schritt modellieren wir die Servlet-Schnittstelle aus Abbildung 2.7 mit Hilfe von UML-F. Das Ergebnis zeigt Abbildung 4.1.

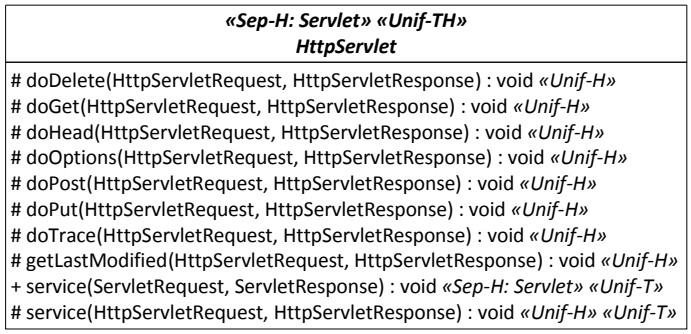


Abbildung 4.1: `HttpServlet` in UML-F

Die Modellierung der `HttpServlet` Klasse wurde um die UML-F spezifischen Stereotypen für Template- und Hook-Methoden erweitert. Es ist zu erkennen, dass diese Klasse sowohl die Rolle einer Hook-Klasse einnimmt (`Sep-H: Servlet`), als auch eigene Template- und Hook-Methoden implementiert (`Unif-TH`).

An dieser Stelle wird bereits eine Schwachstelle in UML-F ersichtlich, da es bereits in kleinen Beispielen geschieht, dass eine Klasse unterschiedliche Rollen einnimmt und in dem einen Zusammenhang bestimmte Template-Methoden definiert und in einem anderen Zusammenhang gleichzeitig Hook-Methoden implementiert. Dies ist ausdrücklich gewollt in UML-F, aber es wird nicht spezifiziert, wie man das Darstellungsproblem löst. Aus der reinen Darstellung im Klassendiagramm ist nicht unbedingt ersichtlich welche Methoden zusammen zu einem Erweiterungspunkt gehören. Der Grund hierfür liegt in der geringen Abstraktion des Template- und Hook-Konzeptes vom eigentlichen Programmcode.

Vergleicht man die Darstellung in Abbildung 4.1 mit der Darstellung in Abbildung 2.7 sind wenig Vorteile der Darstellung mit UML-F ersichtlich. Um den Ablauf zwischen Template- und Hook-Methoden nachvollziehen zu können, benötigt man zusätzlich ein Sequenzdiagramm, wie wir es in Abbildung 2.8 verwendet haben. Die UML-F Annotationen transportieren lediglich die Information, dass besagte Hook-Methoden von der eigenen Implementierung überschrieben werden können. Ein geübter Programmierer leitet diese Information jedoch bereits daraus ab, dass die Klasse `HttpServlet` eine abstrakte Klasse ist, woran man erkennt, dass diese Klasse für weitere Spezialisierung vorgesehen ist.

Zur vollständigen Dokumentation des Beispiels müssen nun noch die Schritte 2 und 3 berücksichtigt werden, denn neben der eigentlichen Implementierung im Programmcode sind diese Information ebenso wichtig, um das programmierte Servlet benutzen zu können. Leider bietet UML-F für diese beiden Schritte keine Unterstützung. In UML-F ist es weder möglich eine bestimmte Verzeichnisstruktur zu modellieren, noch die benötigten Anpassungen an der `web.xml` Datei zu spezifizieren.

F-UML : Eine Sprache für Framework-Design

Eine weitere Sprache für die Spezifikation von Frameworks ist die UML-basierte Sprache F-UML [Bouassida2001]. Diese Sprache fokussiert auf die Spezifikation eines Frameworks im Zuge einer Framework-Entwicklung und weniger auf die Dokumentation der Benutzung eines Frameworks. Nichtsdestotrotz ist die Sprache wie die UML-F geeignet, um auch die Benutzung eines Frameworks aus der Spezifikation herauszulesen. Die Sprache F-UML wurde in [Bouassida2003] auf Basis von »Object-Z« formalisiert. Passend zur Spezifikationssprache F-UML wurde auch eine UML-basierte Framework-Entwicklungsmethode in [Bouassida2002] und [Ben-Abdallah2004] vorgestellt, die wir in Kapitel 6.5 noch näher betrachten werden. Unterstützt wird der Ansatz durch das F-UMLTool [Bouassida2004], in dem die verwendeten Konzepte umgesetzt wurden.

F-UML ist wie UML-F ein UML-Profil, das Stereotypen für bestimmte Diagramme definiert. Die UML-F beschränkt ihre Erweiterungen im Wesentlichen auf Stereotypen in UML-Klassendiagrammen. Mit der F-UML verfolgen die Autoren Nadia Bouassida, Hanêne Ben-Abdallah und Faiez Gargouri einen umfassenderen Ansatz und legen Annotationen für verschiedene Diagrammarten fest. In der F-UML besteht eine Framework-Spezifikation aus vier Diagrammarten [Bouassida2004].

1. Anwendungsfalldiagramme bestimmen den Anwendungsbereich, die Ziele und die Domäne des Frameworks. Durch zusätzliche Annotationen können Anwendungsfälle ausgezeichnet werden, die Erweiterungspunkte enthalten.

2. Klassendiagramme zur Spezifikation der statischen Struktur des Frameworks. Die verfügbaren Stereotypen in F-UML sind nicht so detailliert wie in UML-F. Es lassen sich Klassen auszeichnen, die einen Erweiterungspunkt bereitstellen. Außerdem lässt sich ausdrücken, dass

bestimmte Klassen erweiterbar sind, so dass man eigene spezialisierte Implementierungen vornehmen kann. Das aus der UML-F bekannte Konzept der Template- und Hook-Methoden wird von der F-UML nicht unterstützt.

3. Muster-Diagramme, die die Klassenstruktur des Frameworks zeigen, wobei die Klassen mit Rollenbezeichnern versehen werden. Die Rollenbezeichner stammen von den verwendeten Design-Pattern, die in der Klassenstruktur verwendet wurden. So erkennt man welche Rollen im Sinne der verwendeten Design-Pattern bestimmte Klassen einnehmen.

4. Sequenzdiagramme zeigen mögliche Interaktionen von Objekten mit Erweiterungspunkten innerhalb des Frameworks. So erkennt man welche Methoden auf erweiterbaren Klassen vom Framework aufgerufen werden. Dies ist eine Alternative zur Auszeichnung von Template- und Hook-Methoden, wie es die UML-F vorsieht.

Wir sehen, dass die F-UML Annotationen in unterschiedlichen Diagrammarten anbietet. Hierdurch wird es möglich das Framework auch auf höherer Abstraktion zu beschreiben, indem man zum Beispiel bereits an Anwendungsfällen erkennt, dass diese Erweiterungspunkte enthalten. Damit hebt die F-UML das Manko der UML-F auf, die mit Template- und Hook-Methoden auf einem sehr niedrigen Abstraktionsniveau agiert. Umgekehrt bietet die F-UML nicht die detaillierten Informationen, die die UML-F in Klassendiagrammen ausdrücken kann, da die F-UML das Konzept der Template- und Hook-Methoden nicht unterstützt. Stattdessen werden in der F-UML die Informationen auf verschiedene Diagrammarten verteilt.

Bewertung

Die beiden vorgestellten UML-basierten Ansätze UML-F und F-UML sind UML-Profile, die es erlauben framework-spezifische Informationen in UML-Diagrammen auszudrücken. Die Sprachen gehen davon aus, dass eine Framework-Spezifikation allein dadurch ausgedrückt werden kann, dass man ein paar wenige Zusatzinformationen zu einem objektorientierten Entwurf hinzufügt. Um dies im Detail beurteilen zu können, bewerten wir im Folgenden die einzelnen Merkmale einer Framework-Beschreibung im Hinblick auf deren Erfüllung mit UML-basierten Beschreibungssprachen. Wir beginnen mit der Bewertung der Anwendbarkeit in Tabelle 4.19.

Tabelle 4.19: Anwendbarkeit UML-basierter Sprachen

ID	Anforderung	Umsetzung mit der UML	Bewertung
ANW-B	Eine Framework-Beschreibungssprache muss Konzepte unterstützen, die einen Benutzer bei Verwendung einer Framework-Beschreibung, d.h. beim Explorieren und Erlernen eines Frameworks unterstützen.	Die UML Stereotypen helfen eine Übersicht über das Framework zu erhalten und Erweiterungspunkte zu identifizieren. Insbesondere die F-UML bietet Unterstützung durch Anwendungsfalldiagramme.	+
ANW-E	Die Erstellung der Framework-Beschreibung muss sich so in den Entwicklungsprozess des Frameworks integrieren, dass der Aufwand der Entwickler für die Erstellung möglichst gering ist.	Falls das Framework zuvor mit Hilfe der UML spezifiziert wurde, ist die Anreicherung dieser Modelle um Stereotypen mit vertretbarem Aufwand möglich.	+
ANW-W	Die Informationen einer Framework-Beschreibung müssen für Werkzeug-Entwickler programmatisch auswertbar sein.	Stereotypen sind leichtgewichtige Erweiterungen des UML Meta-Modells und damit automatisch auswertbar.	+

Wir sehen, dass die UML aus Sicht der Anwendbarkeit alle Kriterien erfüllt. Neben der Anwendbarkeit bewerten wir die UML-basierten Ansätze nun im Hinblick auf ihre Ausdrucksfähigkeit.

Tabelle 4.20: Ausdrucksfähigkeit UML-basierter Sprachen

ID	Anforderung der Framework-Benutzer Eine Framework-Beschreibungssprache muss...	Umsetzung mit der UML	Bewertung
AUS-1	... Beispiele für die Benutzung des Frameworks beschreiben können.	Keine Integration von Beispielen vorgesehen.	–
AUS-2	... statische und dynamische Architekturübersichten beschreiben können.	Stereotypen können sowohl in UML Struktur- als auch in Verhaltensdiagrammen eingesetzt werden.	+
AUS-3	... die Erweiterungspunkte eines Frameworks beschreiben können.	Durch entsprechende Stereotypen können speziell in der F-UML Erweiterungspunkte ausgezeichnet werden.	+
AUS-4	... die Konfiguration und Integration des Frameworks beschreiben können.	Konfigurationsparameter und Schnittstellen für die Integration können mit der UML modelliert werden.	+

ID	Anforderung der Framework-Benutzer Eine Framework-Beschreibungssprache muss...	Umsetzung mit der UML	Bewertung
AUS-5	... die Implementierung einer Erweiterung beschreiben können.	Template- und Hook-Methoden der UML-F beschreiben die Implementierung.	+
AUS-6	... das Kommunikationsprotokoll zwischen Framework und Erweiterung beschreiben können.	Mit UML (Protokoll-) Zustandsdiagrammen möglich.	+
AUS-7	... die Bindung von Erweiterungen an das Framework beschreiben können.	Nicht Teil der UML.	–
AUS-8	... das Format und die Installation einer Erweiterung in das Framework beschreiben können.	Nicht Teil der UML.	–
AUS-9	... einzuhaltende Bedingungen seitens einer Erweiterung beschreiben können.	UML OCL Constraints können verwendet werden, um gültige Objektmodelle zu einem Klassenmodell einzuschränken. Sie eignen sich jedoch nicht, um gültige Implementierungen einer Hook-Methode einzuschränken. Diese Art der Einschränkung ist nicht Teil der UML.	–
AUS-10	... Varianten von Erweiterungspunkten beschreiben können.	Verschiedene Varianten können durch mehrere Modelle umgesetzt werden. Allerdings kann nicht modelliert werden, dass ein Modell eine Variante eines anderen ist.	–

Aus der Bewertung der Ausdrucksfähigkeit geht hervor, dass die UML-basierten Sprachen Schwächen im Bereich bestimmter Abstraktionen haben, die für eine Framework-Beschreibung wichtig sind. Insbesondere fehlt eine Abstraktion im Hinblick auf die Erweiterungspunkte eines Frameworks. Möchte ein Benutzer einen guten Überblick über die Erweiterungsmöglichkeiten des Frameworks erhalten, so ist die Darstellung der Template- und Hook-Methoden nur eingeschränkt hilfreich. Die Verwendung von Anwendungsfalldiagrammen in der UML-F ist hier ein richtiger Schritt, aber es fehlt die Verbindung zur Architektur des Frameworks. Neben den erweiterbaren Anwendungsfällen muss ein Framework-Benutzer erkennen können, wo im Gesamtbild des Frameworks Erweiterungen möglich sind. Dies ist insofern wichtig, als dass Frameworks durchaus in einer Art und Weise benutzt werden,

die nicht von den beschriebenen Anwendungsfällen abgedeckt sind. Bewerten wir nun die Erweiterbarkeit der UML.

Tabelle 4.21: Erweiterbarkeit UML-basierter Sprachen

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Umsetzung mit der UML	Bewertung
ERW-B	... existierende Sprachen, mit denen Framework-Benutzer bereits vertraut sind, zur Beschreibung einzelner Merkmale integrieren können.	Da mit UML Profilen gearbeitet wird, stehen alle Erweiterungsmöglichkeiten der UML zur Verfügung. Andere Sprachen können in die UML integriert werden, wenn diese kompatibel zum UML Meta-Modell sind.	+
ERW-E	... es den Framework-Entwicklern ermöglichen, die Framework-Beschreibung um neue Beschreibungsformen für Erweiterungspunkte zu erweitern.	Die UML ist nicht auf die Beschreibung von Frameworks und ihrer Erweiterungspunkte spezialisiert. Die Profile schaffen in einigen Teilen Abhilfe, aber sind nicht ausreichend, um neue Formen von Erweiterungspunkten, die sich z.B. nicht durch Klassen und Methoden ausdrücken lassen, zu modellieren. Hierfür kann jedoch bei Bedarf eine Erweiterung des UML Meta-Modells definiert werden, was jedoch einen tiefen Eingriff in das UML Meta-Modell erfordert.	+
ERW-W	... müssen in der Lage sein, Werkzeuge auf Basis der Sprachdefinition der Framework-Beschreibungssprache zu erweitern.	Werkzeuge, die auf Meta-Modellen zur Sprachdefinition operieren, sind sehr gut erweiterbar, da mit Erweiterung des Meta-Modells auch das Werkzeug erweitert werden kann.	+

Die Erweiterbarkeit UML-basierter Sprachen ist gekoppelt an die Erweiterungsmöglichkeiten der UML. Diese bietet in ihrer aktuellen Version [UMLINF23] einige Möglichkeiten der Erweiterung, die damit auch den Profilen zur Verfügung stehen. Allerdings ist die UML nicht auf die Modellierung von Frameworks beschränkt, so dass so spezifische Erweiterungen wie neue Beschreibungsformen für Erweiterungspunkte schwerer umzusetzen sind. Hierfür wären tiefgreifende Veränderungen am UML Meta-Modell notwendig, was zwar möglich ist, jedoch die Wiederverwendung verfügbarer UML Editoren einschränkt. Allerdings ermöglicht es die Meta-Modell-basierte Sprachdefinition der UML, Werkzeuge so zu gestalten, dass Erweiterungen des Meta-Modells leicht im Werkzeug umzusetzen sind.

Fazit

UML-basierte Sprachen bieten mit der Annotation von Erweiterungspunkten, Template- und Hook-Methoden eine brauchbare Erweiterung der UML. In der Praxis ist jedoch fraglich, ob durch die Fülle an Annotationen in realistischen Beispielen die Übersichtlichkeit nicht derart leidet, dass ein Framework-Benutzer wenig Nutzen daraus ziehen kann. Diagramme wie UML-Komponentendiagramme auf höherem Abstraktionsniveau sind nicht auf Frameworks ausgelegt, da das Konzept von Erweiterungspunkten hier nicht berücksichtigt ist. Die vorgestellten UML-basierten Sprachen zeigen keine zufriedenstellende Lösung für dieses Problem. Außerdem sind UML-basierte Sprachen in erster Linie für die Spezifikation eines Frameworks gedacht. Eine Framework-Spezifikation richtet sich jedoch in erster Linie an Software-Entwickler, die das Framework selbst implementieren müssen. Die Framework-Benutzer werden, wenn überhaupt, nur am Rande berücksichtigt. Aus diesem Grund werden Framework-Benutzer häufig mit zu vielen internen Implementierungsdetails konfrontiert, die sie für die reine Benutzung des Frameworks nicht benötigen.

4.2.5 Auswertung

Die Analyse existierender Ansätze zur Framework-Beschreibung aus Kapitel 4.2 hat ergeben, dass kein Ansatz allein alle geforderten Merkmale erfüllen kann. Hierfür lassen sich zwei Gründe identifizieren:

1. Die existierenden Ansätze haben eine einschränkende Sicht auf Application Frameworks und deren Implementierung. Ihnen fehlt eine ganzheitliche Sicht auf das Framework-Konzept, das allen Frameworks zugrunde liegt. Dadurch werden wichtige Merkmale wie die Bindung oder die Installation von Erweiterungspunkten nicht berücksichtigt.

2. Die existierenden Ansätze fokussieren zu wenig auf die Bedürfnisse der Framework-Benutzer. Häufig wird der Entwurf des Frameworks dokumentiert, aber nicht wie Framework-Benutzer es anwenden können. Eine zu geringe Abstraktion von Implementierungsdetails erschwert es Framework-Benutzern die für sie relevanten Informationen zu extrahieren.

Allerdings hat jeder beschriebene Ansatz in bestimmten Bereichen Stärken, die sinnvoll in eine Framework-Beschreibung eingebunden werden sollten. Um die Stärken der einzelnen Ansätze zu vereinen, benötigt man einen vereinheitlichenden Ansatz zur Framework-Beschreibung. Durch die Vereinigung bewährter, existierender Konzepte zur Framework-Beschreibung plus Erweiterungen um bisher unberücksichtigte Aspekte kann der existierende Mangel behoben werden. Zusätzlich muss die vereinigte Framework-Beschreibungssprache so aufgebaut sein, dass deren Informationen automatisch ausgewertet werden können, um sich in das Konzept der automatischen Kompatibilitätsprüfung aus Kapitel 3 einbetten zu lassen.

Im Gebiet der Softwaretechnik haben sich Meta-Modelle als ein geeignetes Mittel zur Spezifikation von Sprachen etabliert. Das UML Meta-Modell ist ein

klassisches Beispiel für die Definition eines Meta-Modells zur Spezifikation der Sprache UML. Im Meta-Modell werden alle Sprachkonstrukte spezifiziert, mit deren Hilfe sich Sätze der Sprache bilden lassen. Im Falle der UML sind die Sätze der Sprache Modelle, die sich mit Hilfe der UML ausdrücken lassen. Jedes UML Modell ist dabei eine gültige Instanz des UML Meta-Modells.

Mit der F-UML und der UML-F haben wir bereits UML-basierte und damit auch Meta-Modell-basierte Sprachen betrachtet. Diese beiden Sprachen sind jeweils als UML-Profile definiert, was bedeutet, dass sie das UML Meta-Modell um neue Stereotypen erweitern. Beide Sprachen enthalten jedoch die volle Ausdrucksfähigkeit der UML. Wir haben jedoch gesehen, dass die UML-basierten Sprachen nicht alle Merkmale einer Framework-Beschreibungssprache unterstützen können. Daher benötigen wir ein Meta-Modell, unabhängig von der UML, das alle Merkmale in einer vereinheitlichten Framework-Beschreibungssprache umsetzt. Hierbei ist zu untersuchen, welche Rolle die UML in diesem Meta-Modell einnimmt. Im folgenden Abschnitt 4.3 wollen wir das Konzept einer ganzheitlichen Framework-Beschreibung betrachten und daraus resultierende Anforderungen an eine Umsetzung ableiten.

4.3 Konzept einer ganzheitlichen Framework-Beschreibung

Eine ganzheitliche Framework-Beschreibung muss in der Lage sein, alle geforderten Merkmale umzusetzen. Darüber hinaus sollte es möglich sein, dass für bestimmte Aspekte bereits etablierte existierende Sprachen wiederverwendet werden können. So können Sprachen und auch deren Werkzeuge direkt wiederverwendet werden. So gibt es etwa eine ganze Reihe von Architektur-Beschreibungssprachen [Clements1996], die eventuell wiederverwendet werden könnten.

Betrachten wir konkret das Konzept des Hook-Protokolls, das beschreibt in welcher Reihenfolge Hook-Methoden einer Erweiterung vom Framework aufgerufen werden. Zur Erinnerung: Eine Hook-Methode ist eine Methode, die vom Framework definiert wird, von einer Erweiterung implementiert werden muss und nach dem Hollywood-Prinzip vom Framework aufgerufen wird. Ein Hook-Protokoll beschreibt somit ein Protokoll für Methodenaufrufe, das man auch als »Kommunikationsprotokoll« zwischen Framework und Erweiterung bezeichnet.

Zur Beschreibung von Kommunikationsprotokollen existieren eine Reihe von etablierten Ansätzen und Sprachen. Beispiele hierfür sind Petri-Netze, UML-Protokollzustandsautomaten [UMLSUP23] oder W3C State Chart XML [W3CSC]. Bei der Definition einer vereinheitlichenden Framework-Beschreibungssprache stellt sich unmittelbar die Frage, ob man in dieser nicht eine der genannten Sprachen wiederverwenden könnte, um Hook-Protokolle zu beschreiben. Neben der reinen Wiederverwendung der Sprache ist auch die Wiederverwendung existierender Werkzeuge und Infrastrukturen von Interesse. Der Grund ist, dass die Entwicklung einer Sprache bereits eine aufwändige

Aufgabe ist, aber die Entwicklung zusätzlicher Werkzeuge und Editoren für diese Sprachen sind eine Investition, die es durch Wiederverwendung zu nutzen gilt.

Es stellt sich die Frage, wie man eine existierende Meta-Modell-basierte Sprache formal korrekt in andere Sprachen integrieren kann. Dabei will man nicht beliebige Sprachen integrieren, sondern wie im Protokoll-Beispiel nur Sprachen, die einen ganz bestimmten Aspekt ausdrücken können. Unterschiedliche Benutzer bevorzugen eventuell unterschiedliche Sprachen, weil sich Benutzer zum Beispiel mit einer Sprache schon gut auskennen und im Besitz passender Werkzeuge sind. Da es für manche Aspekte durchaus verschiedene Alternativen geben kann, welche Sprache man hierfür integrieren möchte, sollte das Meta-Modell so flexibel sein, dass man leicht eine andere Sprache verwenden kann. Eine solche Flexibilität setzt jedoch voraus, dass keine Sprache so fest in das Meta-Modell integriert ist, dass man sie nicht leicht wieder entfernen könnte. Insbesondere die automatische Kompatibilitätsprüfung muss unabhängig von den verwendeten Sprachen spezifiziert und implementiert werden können. Dies verlangt nach einem stabilen Sprachkern, der für Spezifikation und Implementierung unverändert bleibt. In diesen stabilen Sprachkern werden die existierenden Sprachen für bestimmte Aspekte flexibel integriert.

Konzeptionell stellt sich der Ansatz einer ganzheitlichen Framework-Beschreibungssprache wie in Abbildung 4.2 gezeigt dar. Die neue Sprache integriert verschiedene Aspekte, die für eine ganzheitliche Framework-Beschreibung notwendig sind. Diese Aspekte sind zum Beispiel die Architektur des Frameworks, existierende Erweiterungspunkte oder die Bindung von Erweiterungen an das Framework. Jeder Aspekt wird nun durch eine eigene Sprache ausgedrückt. So gibt es Sprachen für die Architektur, für Erweiterungspunkte und für die Bindung. Es muss jedoch sichergestellt sein, dass die jeweils verwendete Sprache zum Aspekt passt. Der Aspekt bestimmt die Eigenschaften, die eine Sprache erfüllen muss. Durch die Spezifikation des Aspektes bleibt der Kern der Framework-Beschreibungssprache wie gewünscht stabil und verwendete Sprachen können flexibel eingehängt werden.

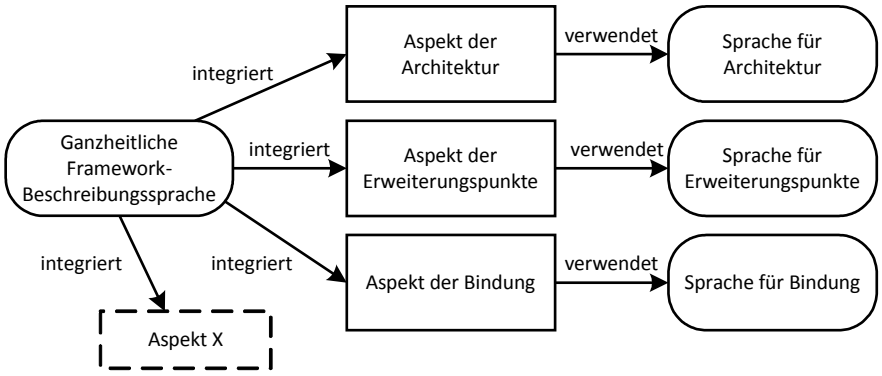


Abbildung 4.2: Konzept einer ganzheitlichen Framework-Beschreibungssprache

Zusammenfassend bleibt das Problem, ein Meta-Modell für Framework-Beschreibungssprachen entsprechend des in Abbildung 4.2 skizzierten Konzep-

tes zu spezifizieren, das alle geforderten Merkmale ganzheitlich abdeckt und dabei existierende Sprachen für bestimmte Teilaspekte flexibel integriert. Hieraus entstehen eine Reihe von Anforderungen an die Wiederverwendung von Sprachen.

4.4 Zusammenfassung

In diesem Kapitel haben wir gesehen, dass bestehende Ansätze zur Framework-Beschreibung die gestellten Anforderungen nicht vollständig abdecken können. Um die existierenden Lücken zu schließen, werden wir in einem ganzheitlichen Ansatz die bewährten Konzepte einzelner Ansätze in einer neuen Framework-Beschreibungssprache vereinen. Eine solche Vereinigung hat zur Konsequenz, dass für unterschiedliche Teile einer Framework-Beschreibung existierende Ansätze wiederverwendet werden. Auf Sprachebene bedeutet dies, dass existierende Sprachen zu einer gemeinsamen Sprache integriert werden. Durch die Integration existierender Sprachen sind wir in der Lage eine ganzheitliche Beschreibung für Frameworks zu definieren, die darüber hinaus den Anforderungen an eine automatische Auswertbarkeit im Hinblick auf die automatische Kompatibilitätsprüfung genügt.

5

Wiederverwendung von Sprachen

The mediocre code compiles.
The good code runs.
The superiorcode passes
tests and inspections.
The great code inspires.

-- William Ward

Um das Konzept einer ganzheitlichen Framework-Beschreibungssprache aus Abschnitt 4.3 umsetzen zu können, benötigen wir ein Vorgehen, das es erlaubt existierende Sprachen als Teile einer neuen Sprache wiederzuverwenden. Dabei ist das Ziel, einen praktisch anwendbaren Ansatz zu verwenden und weniger eine theoretische Beweisbarkeit für die Wiederverwendung von Sprachen zu zeigen. Es ist nicht der Anspruch für die Problemstellung dieser Arbeit eine formale Sprachdefinition zu etablieren, auf der wir die korrekte Wiederverwendung existierender Sprachen beweisen könnten. Vielmehr benötigen wir eine Mischung aus intuitiver Modellierung gestützt durch formale Eigenschaften.

In diesem Kapitel stellen wir mit dem neuen Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen solch einen praktischen Ansatz vor.

Der Ansatz beschreibt ein Vorgehen, das durch geeignete Modellierungs- und Verfahrenstechniken unterstützt wird. Bevor wir die Lösung aus Vorgehen und Techniken vorstellen, analysieren wir zunächst das gestellte Problem.

Zur Problemanalyse modellieren wir die Situation in Abbildung 5.1. In diesem Modell verallgemeinern wir die Forderung nach einer ganzheitlichen Framework-Beschreibungssprache auf die Forderung nach einer ganzheitlichen Sprache, die aus einzelnen Teilaspekten besteht. Dabei kann für jeden Teilaspekt eine existierende Sprache wiederverwendet werden, wenn die existierende Sprache dieselben Sprachkonzepte bereitstellt, wie sie von der Sprachdefinition des Teilaspekts gefordert werden. Hierzu muss die Sprachdefinition eines Teilaspektes in *geeigneter Relation* zur Sprachdefinition der existierenden Sprache stehen.

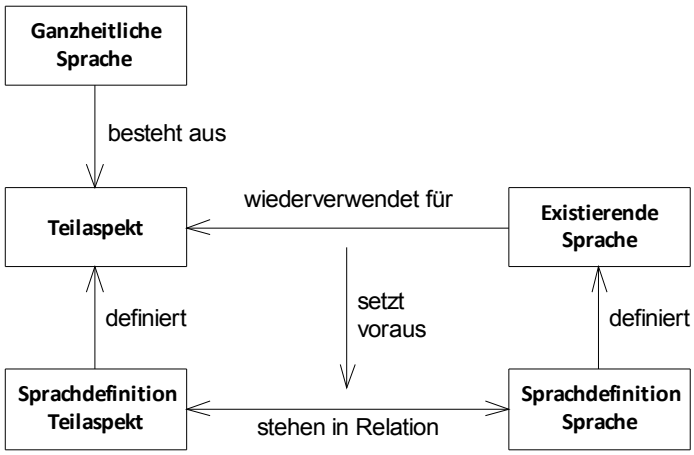


Abbildung 5.1: Wiederverwendung von Sprachen für Teilaspekte einer ganzheitlichen Sprache

Wir wollen eine Sprachdefinition genauer betrachten. Im Allgemeinen spezifiziert eine Sprachdefinition die Syntax und die Semantik einer Sprache. Insbesondere die Betrachtung der Semantik einer Sprache hat dabei eine Reihe von Facetten. Während die Syntax einer Sprache keine Bedeutung transportiert, besitzen die verwendeten Konzepte innerhalb der Syntax eine Semantik.

Diesen Sachverhalt können wir veranschaulichen, wenn wir uns eine Sprachdefinition mit Hilfe eines Meta-Modells vor Augen führen. Für eine Einführung in die Meta-Modellierung sei auf die Grundlagen in Abschnitt 5.3.1 verwiesen. In einem Meta-Modell definieren Klassen und Assoziationen die Syntax einer Sprache. Die verwendeten Konzepte in Form von Klassen- und Rollennamen im Meta-Modell transportieren dabei die Semantik der syntaktischen Elemente. Entfernt man die Konzepte aus einem Meta-Modell wird deutlich, dass die verbleibende Syntax keinerlei Bedeutung hat. Die Syntax definiert lediglich wie man die Konzepte syntaktisch korrekt anordnet.

Die Semantik der in einem Meta-Modell verwendeten Konzepte ist häufig nicht formal spezifiziert. Stattdessen nimmt man eine implizite Semantik an, die auf der intuitiven Sprachbedeutung der Konzepte beruht. Eine Klasse mit

dem Namen »Zustand« wird nach diesem intuitiven Verständnis das Konzept eines Zustands repräsentieren.

Um die Begrifflichkeiten voneinander zu trennen, sprechen wir in dieser Arbeit von der Syntax einer Sprache, wenn wirklich nur die Syntax ohne Semantik gemeint ist. Die Syntax ist definiert als die Struktur einer Sprache. Beide Begriffe verwenden wir synonym.

Zusammenfassend besteht eine Sprachdefinition im Sinne dieser Arbeit aus der Definition der Syntax und der Semantik einer Sprache. Beide Aspekte sind in einem Meta-Modell definiert. Den schematischen Zusammenhang einer Sprachdefinition zeigen wir in Abbildung 5.2. Die Syntax wird dabei durch die Struktur vorgegeben. Die Semantik erhält eine Sprache durch die Bedeutung der verwendeten Konzepte.

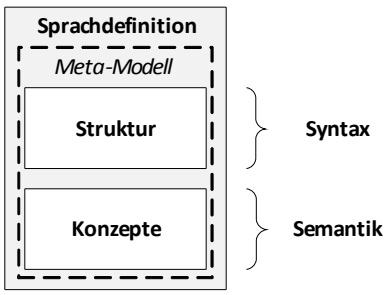


Abbildung 5.2: Inhalt einer Sprachdefinition

Mit diesen Überlegungen wollen wir die in Abbildung 5.1 angedeutete Relation zwischen zwei Sprachdefinitionen genauer untersuchen. Um zwei Sprachdefinitionen in Relation zu setzen, müssen wir die Struktur und die Konzepte beider Sprachen betrachten. Die Wiederverwendung einer Sprache setzt voraus, dass die geforderte Struktur und die Konzepte eines Teilaspektes von einer wiederverwendeten Sprache zur Verfügung gestellt werden. So entsteht eine Relation auf Basis der Meta-Modelle beider Sprachen.

Die gesuchte Relation zwischen den Meta-Modellen muss sicherstellen, dass die geforderte Sprache des Teilaspektes von der existierenden Sprache zur Verfügung gestellt wird. Dies bedeutet, dass alles, was in der Sprache des Teilaspektes ausgedrückt werden kann, auch in der existierenden Sprache ausgedrückt werden kann. In dieser Richtung muss die Sprachdefinition des Teilaspektes auf die Sprachdefinition der existierenden Sprache *abbildbar* sein.

Angewandt auf eine Sprachdefinition basierend auf Meta-Modellen, wie wir sie in dieser Arbeit verwenden, muss das Meta-Modell der geforderten Sprache auf das Meta-Modell der existierenden Sprache abgebildet werden können. Diese Abbildung muss sicherstellen, dass alle Elemente des geforderten Meta-Modells eine Entsprechung im Meta-Modell der existierenden Sprache haben. Betrachten wir zur Veranschaulichung ein kleines Beispiel.

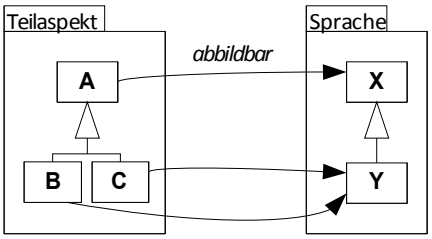


Abbildung 5.3: Gefordertes Meta-Modell abbildbar auf das Meta-Modell einer Sprache

Im Beispiel in Abbildung 5.3 sehen wir ein Meta-Modell mit einer Vererbungshierarchie zwischen den Klassen A, B und C. Damit das Meta-Modell des Teilaspektes wie skizziert auf das Meta-Modell der Sprache abgebildet werden kann, darf die Struktur der Vererbungshierarchie nicht gebrochen werden. Dies wird sichergestellt, da die Elternklasse A links auf die Elternklasse X rechts abgebildet wird, zusätzlich werden die Kinder B und C links beide auf die Kindklasse Y rechts abgebildet. Die Hierarchie bleibt logisch erhalten. Damit sind die Anforderungen an die Struktur erfüllt, aber auf der semantischen Ebene muss zusätzlich gelten, dass das Konzept, das hinter A steht auf das Konzept Y abgebildet werden kann. Ebenso müssen die Konzepte der Kindklassen B und C auf das Konzept Y semantisch vereinigt werden können. Nur wenn Syntax und Semantik abbildbar sind, ist auch das Meta-Modell abbildbar.

In der einen Richtung muss eine Sprachdefinition somit im beschriebenen Sinne abbildbar sein. Widmen wir uns nun der Rückrichtung dieser Relation. In der Regel ist die wiederverwendete Sprache mächtiger als die Sprache des Teilaspektes in dem Sinne, dass die konkrete Sprache mehr Sprachkonzepte bereitstellt als vom Teilaspekt gefordert. Diese Sprachkonzepte sind nicht durch die Abbildung der Sprache des Teilaspekts auf die konkrete Sprache erfasst. Die Relation muss sicherstellen, dass bei Übersetzung von der konkreten Sprache in die Sprache des Teilaspekts diese weiteren Sprachkonzepte ignoriert werden können. Die Sprachdefinition der existierenden Sprache muss in dieser Richtung auf die Sprachdefinition des Teilaspektes *reduzierbar* sein. Betrachten wir auch hierzu ein veranschaulichendes Beispiel.

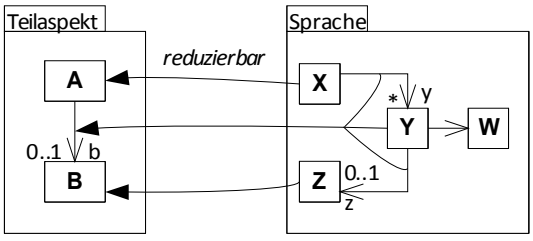


Abbildung 5.4: Meta-Modell einer Sprache reduzierbar auf gefordertes Meta-Modell

Das Beispiel in Abbildung 5.4 zeigt ein gefordertes Meta-Modell des Teilaspektes im linken Teil, das weniger Klassen enthält als das Meta-Modell der

Sprache rechts. Das Meta-Modell der Sprache ist in diesem Sinne mächtiger und muss auf das Meta-Modell des Teilaspektes reduziert werden können. Demzufolge muss es syntaktisch und semantisch möglich sein Klassen, wie im Beispiel die Klasse W, auszublenden, da diese nicht gefordert wird. Darüber hinaus müssen komplexe Assoziationen wie zwischen den Klassen X, Y und Z auf der rechten Seite auf einfache Assoziationen wie zwischen A und B auf der linken Seite reduziert werden können. Hierbei sind wiederum die strukturellen Eigenschaften wie Kardinalitäten und Richtungen von Assoziationen zu beachten, wie auch die Semantik der zu reduzierenden Konzepte.

Zusammenfassend halten wir fest: Kann ein Teilaspekt auf eine existierende Sprache abgebildet und in Rückrichtung reduziert werden, sprechen wir davon, dass eine existierende Sprache an den Teilaspekt *gebunden* ist. Daher bezeichnen wir die Relation als *Bindung* zwischen Teilaspekt und existierender Sprache.

Die Bindung muss gemäß unserer oben angestellten Überlegungen sicherstellen, dass die Struktur und die Konzepte des Teilaspektes auf die Struktur und die Konzepte der existierenden Sprache abbildbar und, dass die Struktur und Konzepte der existierenden Sprache auf die Struktur und Konzepte des Teilaspektes reduzierbar sind. Wir veranschaulichen dies in Abbildung 5.5.

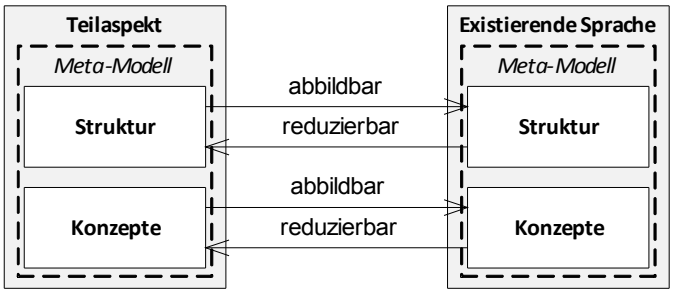


Abbildung 5.5: Bindung zwischen Sprachdefinition eines Teilaspekts und Sprachdefinition einer existierenden Sprache

Mathematisch gesehen fordern wir, dass die Relation zwischen beiden Meta-Modellen *rechtseindeutig* ist, so dass keinem Element im Meta-Modell des Teilaspektes mehr als ein Element im Meta-Modell der Sprache zugeordnet ist. Dies entspricht einer Bindung, bei der alle Elemente des Teilaspektes abgebildet sind, aber die existierende Sprache mächtiger sein kann und somit reduziert wird.

Es stellt sich jedoch die Frage, wann Strukturen und Konzepte gebunden werden können. Intuitiv muss für jedes Element des Teilaspektes eine Entsprechung in der existierenden Sprache gefunden werden. Bei Abbildung der Struktur kann eine Entsprechung durch eine syntaktische Überprüfung gefunden werden. Da die Struktur keine Semantik hat, ist es immer möglich eine Abbildung zwischen zwei Strukturen zu definieren. Erst bei der zusätzlichen Betrachtung der Bindung von Konzepten kommt die Semantik ins Spiel.

Bei Bindung der semantikbehafteten Konzepte ist entscheidend, dass man sicherstellt, dass die Bedeutung der verwendeten Konzepte auf beiden Seiten gleich verstanden wird. Nur gleiche Konzepte können aneinander gebunden

werden. In der Kombination aus gebundener Struktur unter Beachtung gebundener Konzepte liegt das Wesen der Wiederverwendung von Sprachen. Beide Aspekte sind zu beachten.

In Abschnitt 5.6 werden wir uns einer Reihe verwandter Arbeiten zum Thema der Wiederverwendung von Sprachen widmen. Wir werden sehen, dass bestehende Ansätze sich in der Regel nur darauf beschränken, die Wiederverwendung als Abbildung der Struktur zu verstehen. Häufig werden die Konzepte dabei nur implizit behandelt. Einige Ansätze gehen zum Beispiel davon aus, dass nur Elemente mit gleichem Namen aufeinander abzubilden sind.

Das Problem hierbei ist die Möglichkeit sprachlicher Mehrdeutigkeiten, die zu Missverständnissen und letztendlich zu einer fehlerhaften Bindung führen können. Beispielsweise können Elemente mit gleichem Namen für unterschiedliche Konzepte stehen. Diese Unterscheidung ist aus einem Meta-Modell aber unter Umständen nicht ersichtlich.

Damit erfassen Ansätze, die sich nur auf Namen von Elementen stützen, logischerweise nicht wirklich die Semantik der benutzten Konzepte. Die Konsequenz eines solchen Vorgehens ist, dass das Vertrauen in die Korrektheit einer Bindung zwischen zwei Sprachen relativ gering ist. Ohne tiefere Beachtung der Semantik, könnten bei der Bindung unentdeckte Fehler gemacht werden.

Wir erweitern daher zur tieferen Betrachtung der Semantik einer Sprache die Sprachdefinition aus Abbildung 5.2 in Abbildung 5.6 um den semantischen Aspekt des Ausführungsverhaltens. In unserer bisherigen Auffassung einer Sprachdefinition war die Semantik einer Sprache nur durch ihre Konzepte festgelegt. Es kann jedoch eine ganze Reihe weiterer semantischer Eigenschaften geben. Beispiele für weitere Eigenschaften sind die Definition semantisch äquivalenter Sprachkonstrukte oder die Festlegung von semantischen Bedingungen für die Verwendung. Wir beziehen uns für unseren Ansatz auf die Eigenschaft des Ausführungsverhaltens einer Sprache.

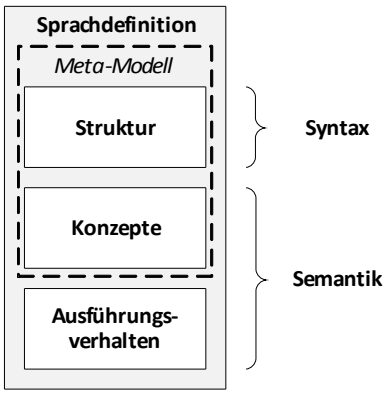


Abbildung 5.6: Sprachdefinition erweitert um die semantische Eigenschaft des Ausführungsverhaltens

Die semantische Eigenschaft des Ausführungsverhaltens werden wir einsetzen, um die Korrektheit einer Bindung auf semantischer Ebene testen zu können. Hierzu führen wir in Abschnitt 5.1 einen innovativen Ansatz zur *anforderungsbasierten Wiederverwendung von Sprachen* ein. In diesem Ansatz

leiten wir aus Anforderungen an eine gesuchte Sprache die geforderte Struktur und Semantik ab. Dabei modellieren wir die Bindung zwischen zwei Sprachen mit Hilfe der neu entwickelten Modellierungstechnik der *parametrisierten Meta-Modelle*.

Um diese Bindung auf ihre semantische Korrektheit prüfen zu können, führen wir den *semantischen Bindungstest* als neues Testverfahren ein, das hilft das Vertrauen in die Korrektheit der Bindung zu stärken. Der semantische Bindungstest basiert auf dem Ausführungsverhalten der geforderten Sprache und ist eine Neuerung, die bestehende Ansätze nicht bieten können. Er prüft, ob die Strukturen und Konzepte korrekt gebunden wurden. Die Annahme ist, dass Strukturen und Konzepte nur dann korrekt gebunden sind, wenn sich das erwartete Ausführungsverhalten des Teilaspektes mit dem tatsächlichen Ausführungsverhalten der existierenden Sprache deckt. Dies testet der semantische Bindungstest.

Als Folge der Festlegung auf die Eigenschaft des Ausführungsverhaltens einer Sprachdefinition in Abbildung 5.6 kann unser Ansatz für Sprachen, die kein Ausführungsverhalten besitzen, nicht verfolgt werden. Für Sprachen ohne Ausführungsverhalten können wir, wie in verwandten Ansätzen auch, die Bindung von Konzepten nur auf die intuitive Bedeutung von Bezeichnern für Konzepte stützen, ohne auf weitere semantische Eigenschaften zurückzugreifen.

Wir werden im nachfolgenden Abschnitt 5.1 den neuen Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen vorstellen. Dabei gehen wir in Abschnitt 5.3 auf die Modellierung mit parametrisierten Meta-Modellen ein und erläutern in Abschnitt 5.4 das Testverfahren des semantischen Bindungstest.

5.1 Anforderungsbasierte Wiederverwendung von Sprachen

Wir haben in Abbildung 5.1 die Problemsituation der Wiederverwendung von Sprachen für bestimmte Teilaspekte einer ganzheitlichen Sprache skizziert. Wir können dieses Problem so auffassen, dass bei Wiederverwendung einer Sprache zu prüfen ist, ob die gestellten Anforderungen an eine gesuchte Sprache von einer existierenden Sprache erfüllt werden.

Wir wollen das Vorgehen für die Wiederverwendung in Abbildung 5.7 zunächst skizzieren bevor wir dieses im weiteren Verlauf konkretisieren. Beim Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen gehen wir in fünf Schritten vor.

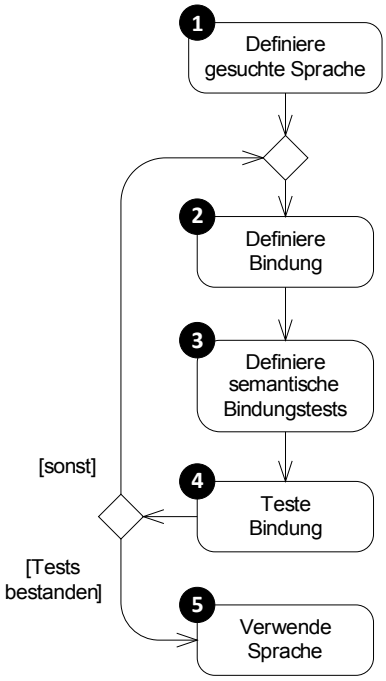


Abbildung 5.7: Vorgehen bei anforderungsbasierter Wiederverwendung von Sprachen

In Schritt 1 definieren wir die gesuchte Sprache, die für einen Teilaspekt einer Sprache wiederverwendet werden soll. Um eine konkrete Sprache wiederzuverwenden, müssen wir prüfen, ob diese der gesuchten Sprache entspricht. Hierzu definieren wir in Schritt 2 eine Bindung zwischen gesuchter Sprache und einer existierenden Sprache, die wiederverwendet werden soll.

Um zu überprüfen, ob die existierende Sprache der gesuchten auch auf semantischer Ebene entspricht, definieren wir in Schritt 3 eine Reihe von semantischen Bindungstest, die die Bindung aus Schritt 2 testen. In Schritt 4 werden diese Tests ausgeführt.

Für den Fall, dass alle Tests bestanden werden, kann die existierende Sprache in Schritt 5 tatsächlich wiederverwendet werden. Werden nicht alle Tests bestanden, ist zu prüfen, ob bei der Definition der Bindung Fehler gemacht wurden. Falls bei diesem iterativen Vorgehen keine Bindung gefunden werden kann, die zu bestandenen Tests führt, entspricht die existierende Sprache nicht der gesuchten Sprache und kann somit nicht verwendet werden.

Für eine konkrete Umsetzung des skizzierten Vorgehens müssen wir klären, wie die gesuchte Sprache definiert werden kann. Ein möglicher Ausgangspunkt sind Anforderungen, die man an die gesuchte Sprache stellt. Analog zum Softwareentwurf definieren wir diese Anforderungen in Form von Anwendungsfällen. Ein Anwendungsfall einer Sprache bezieht sich auf die Ausdrucksfähigkeit der Sprache. Der Sprachdesigner stellt sich die Frage, was ein Benutzer der Sprache mit der Sprache ausdrücken können soll.

Mit Anwendungsfällen als Ausgangspunkt zur Definition der Anforderungen an eine gesuchte Sprache, wollen wir das Vorgehen aus Abbildung 5.7 in vier

Bereiche unterteilen, aus denen sich die konkrete Umsetzung des Vorgehens ergibt. Wir zeigen diese Unterteilung in Abbildung 5.8.

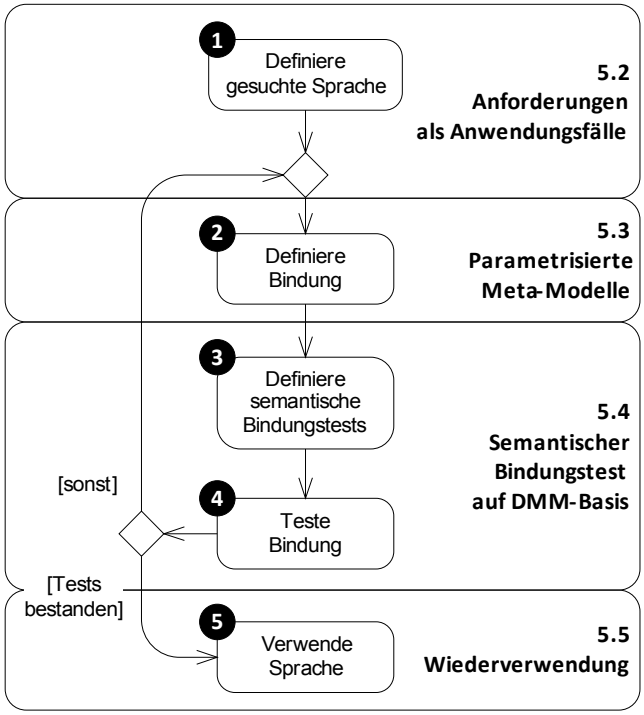


Abbildung 5.8: Konkrete Umsetzung des anforderungsbasierten Vorgehens

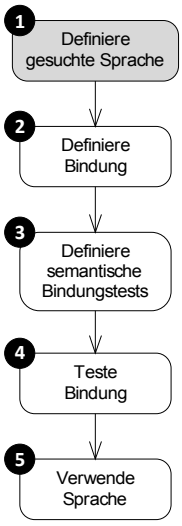
Die Definition der gesuchten Sprache in Schritt 1 setzen wir, wie erläutert, mit Hilfe von Anwendungsfällen um. Ausgehend von definierten Anwendungsfällen kann ein Sprachentwickler die geforderte Struktur und Konzepte einer Sprache in Form eines Meta-Modells modellieren. Ebenso kann das geforderte Ausführungsverhalten in Form so genannter *Verhaltensmuster* definiert werden. Das genaue Vorgehen beschreiben wir im Abschnitt 5.2.

Für Schritt 2, der Bindung von geforderten Strukturen und Konzepten, setzen wir die neu entwickelte Modellierungstechnik der *parametrisierten Meta-Modelle* ein, auf die wir in Abschnitt 5.3 im Detail eingehen werden. Diese Technik drückt die geforderte Struktur und die Konzepte in Form eines Meta-Modells aus, das an das Meta-Modell einer existierenden Sprache gebunden wird. Die Bindung zwischen diesen Meta-Modellen wird im semantischen Bindungstest überprüft.

Schritt 3 und Schritt 4 fassen wir im Bereich des *semantischen Bindungstest auf DMM-Basis* zusammen. Der Bindungstest basiert auf einer formalen Spezifikation des Ausführungsverhaltens von Sprachen. Wir setzen hierfür einen Ansatz mit dem Namen »Dynamic Meta-Modeling« [Engels2000], kurz DMM, ein. Wir verwenden die Verhaltensmuster, um das geforderte Verhalten gegen eine DMM-Spezifikation der existierenden Sprache im Bezug auf die definierte Bindung zu testen. Die Details dieses Testverfahrens beschreiben wir in Abschnitt 5.4.

Im letzten Schritt 5 wird die Sprache bei bestandenen Tests wiederverwendet. In der Umsetzung ist der Bereich der *Wiederverwendung* eng verknüpft mit der Technik der parametrisierten Meta-Modelle. Für den Fall, dass alle Tests bestanden werden, kann das geforderte Meta-Modell durch das konkrete Meta-Modell der existierenden Sprache ersetzt werden. Diese Ersetzung ist nur möglich und valide, wenn die Bindung die entsprechenden Tests bestanden hat. Wir werden auf die Ersetzung in Abschnitt 5.5 eingehen.

5.2 Anforderungen als Anwendungsfälle



In Schritt 1 unseres Vorgehens aus Abbildung 5.8 widmen wir uns der Definition von Anforderungen. Wir wollen Anforderungen an eine gesuchte Sprache mit Hilfe von Anwendungsfällen betrachten. Durch Anwendungsfälle beschreiben wir, was ein Benutzer der Sprache mit der Sprache ausdrücken können möchte. Es wird die Frage gestellt, was die Sprache aus Sicht eines Benutzer können soll. Zur Notation der Anwendungsfälle verwenden wir die etablierte Sprache der UML-Anwendungsfalldiagramme [UMLSUP23] (engl. use case diagram).

Aus Anwendungsfällen lassen sich indirekt Eigenschaften einer gesuchten Sprache ableiten. Diese Ableitung ist ein kreativer Prozess, der die Erfahrung eines Sprachentwicklers voraussetzt. Ein Sprachentwickler ist in der Lage die Anwendungsfälle zu interpretieren und daraus ein Modell der geforderten Strukturen, der Konzepte sowie des geforderten Verhaltens abzuleiten. Die Stützung auf Anwendungsfälle ist eine Variante, um dem Sprachentwickler Detailwissen für seine Arbeit an die Hand geben zu können. Wir wollen dies an einem konkreten Beispiel veranschaulichen.

Aus den Anforderungen an die Ausdrucksfähigkeit für eine Framework-Beschreibungssprache in Tabelle 4.2 wählen wir die Anforderung AUS-2, die besagt, dass eine Framework-Beschreibungssprache statische und dynamische Architekturübersichten beschreiben können muss. Wir konzentrieren uns hier auf den Aspekt einer dynamischen Architekturübersicht, die das interne Verhalten eines Frameworks beschreiben kann. Dieses Beispiel ist bereits ein Vorgriff auf die Einführung des »Framework Description Meta-Modells« (FDMM) in Kapitel 6, das unter anderem diesen Aspekt enthält.

Wir spezifizieren die Anwendungsfälle des Aspektes in Abbildung 5.9. Der Aspekt besteht darin, dass ein Framework-Entwickler das Verhalten eines von ihm entwickelten Frameworks dokumentieren will. Das Verhalten eines Frameworks soll dabei durch mögliche Abfolgen von Schritten im Framework ausgedrückt werden. Dabei können Abläufe auch Hierarchien bilden, so dass ein Schritt eines Ablaufs durch einen Unterablauf beschrieben werden kann. In jedem Schritt eines Ablaufs soll es möglich sein, den Aufruf von Erweiterungspunkten des Frameworks zu beschreiben. So kann ein Framework-Benutzer erkennen, wo in einem Ablauf Erweiterungspunkte aktiv werden.

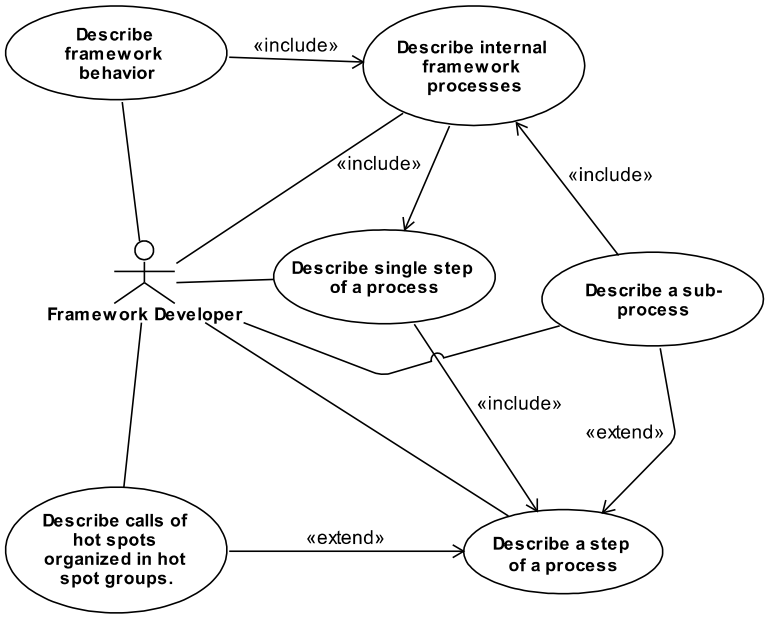


Abbildung 5.9: Anwendungsfälle für die Beschreibung des Verhaltens eines Frameworks

Aus den gezeigten Anwendungsfällen kann ein Sprachentwickler sowohl Anforderungen an die Struktur und Konzepte in Form eines Meta-Modells ableiten als auch Anforderungen an das Ausführungsverhalten durch Verhaltensmuster definieren. Diese Anforderungen bilden zusammengenommen eine Sprachanforderung deren Aufbau wir schematisch in Abbildung 5.10 zeigen. Für jede gesuchte Sprache müssen wir eine solche Sprachanforderung spezifizieren.

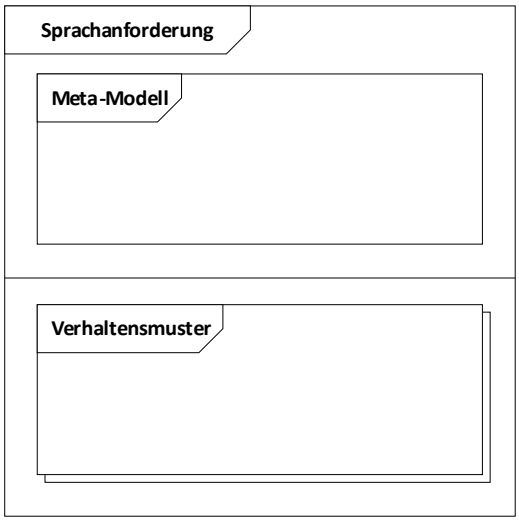


Abbildung 5.10: Struktur einer Sprachanforderung

Ein Sprachanforderung besteht aus einem Meta-Modell, das die Anforderungen an die geforderte Struktur und geforderter Konzepte enthält, und aus einer Menge von Verhaltensmustern, die die Anforderungen an das Verhalten ausdrücken.

Nachfolgend gehen wir in Abschnitt 5.2.1 auf die Erstellung von Meta-Modellen aus Anwendungsfällen und in Abschnitt 5.2.2 auf die Erstellung von Verhaltensmustern aus Anwendungsfällen ein.

5.2.1 Meta-Modelle aus Anwendungsfällen

Aus dem Anwendungsfalldiagramm in Abbildung 5.9 kann ein Sprachentwickler ein Meta-Modell für die gesuchte Sprache ableiten. Zu diesem kreativen Prozess gibt es kein festes Verfahren. Eine einfache Heuristik besteht darin, die verwendeten Substantive als Konzepte zu betrachten und strukturelle Beziehungen zwischen diesen aus den Anwendungsfällen abzuleiten. Dabei ist jedoch die Erfahrung des Sprachentwicklers entscheidend.

In unserem Beispiel erstellt ein Sprachentwickler etwa das Meta-Modell in Abbildung 5.11 für eine Sprache zur Beschreibung des Verhaltens eines Frameworks, genannt BehaviorDL für »Behavior Description Language«.

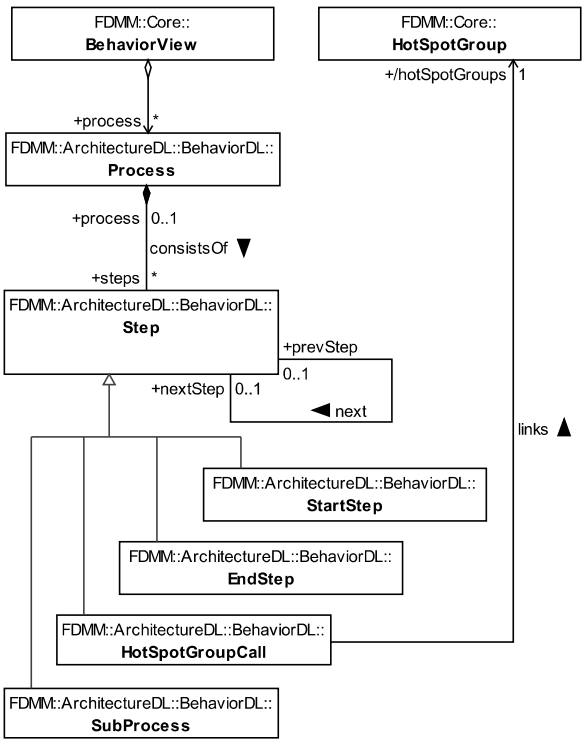


Abbildung 5.11: Meta-Modell der BehaviorDL zur Beschreibung des Verhaltens eines Frameworks

Der Sprachentwickler legt fest, dass das Verhalten eines Frameworks durch eine Menge von Abläufen, repräsentiert durch die Klasse Process in Abbildung

5.11 beschrieben wird. Ein Process besitzt einen startStep, der definiert, mit welchem Schritt (Klasse Step) ein Ablauf startet. Zu jedem Step kann es einen Nachfolgeschritt (nextStep) geben und jeder Step kann eine Unterabfolge (subProcess) enthalten. Außerdem kann zu jedem Step eine Menge von HotSpotGroupCalls zugeordnet werden, die ausdrücken, dass in einem Step bestimmte Erweiterungspunkte (engl. hot spots) aufgerufen werden. So kann beschrieben werden bei welchen Abläufen im Framework welche Erweiterungspunkte aktiv werden.

Neben einem Meta-Modell für Struktur und Konzepte einer Sprachanforderung erstellt der Sprachentwickler außerdem eine Menge von Verhaltensmustern, die wir in Abschnitt 5.2.2 erläutern.

5.2.2 Verhaltensmuster aus Anwendungsfällen

In diesem Abschnitt widmen wir uns der Beschreibung des geforderten Ausführungsverhaltens der Sprache eines Aspektes. Wir wollen hierzu auf die Festlegung unserer Sprachdefinition bestehend aus Struktur, Konzepten und dem Ausführungsverhalten in Abbildung 5.6 zurückkommen. Für eine Umsetzung müssen wir diese Sprachdefinition konkretisieren. Dabei haben wir die Möglichkeit uns im Spektrum zwischen einer intuitiven und damit nicht formalen Sprachdefinitionen und einer formalen Sprachdefinitionen zu bewegen. Für die Umsetzung in dieser Arbeit streben wir einen Mittelweg an, der auch in der Praxis Anwendung findet.

Mit diesen Überlegungen zeigen wir in Abbildung 5.12 eine Konkretisierung der eingesetzten Techniken im Spektrum der Möglichkeiten zwischen einer nicht formalen und einer formalen Sprachdefinition.

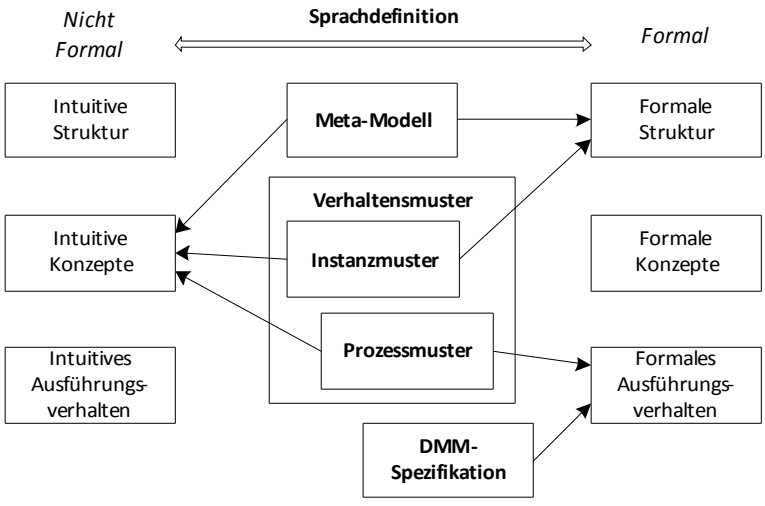


Abbildung 5.12: Eingesetzte Techniken zur Sprachdefinition

Verhaltensmuster teilen wir in Instanzmuster mit Bezug zum Meta-Modell und Prozessmuster mit Bezug zu einer formalen Definition des Ausführungsverhaltens auf. Ziel der Verhaltensmuster ist es, den Übergang von intuitiven Konzepten zu einem formal definierten Ausführungsverhalten vollziehen zu

können. So können wir intuitiv festgelegtes erwartetes Verhalten von Konzepten in einer formalen Spezifikation des Ausführungsverhaltens testen. Zur Spezifikation des formalen Ausführungsverhaltens setzen wir den »Dynamic Meta-Modeling« (DMM) Ansatz ein, den wir in Abschnitt 5.4.1 erläutern werden.

Wir spezifizieren für alle Sprachelemente ihr erwartetes Verhalten bei Ausführung mit Hilfe von Verhaltensmustern. Ein Verhaltensmuster beschreiben wir, indem wir typische Instanzmuster der Sprache betrachten und für diese ein Prozessmuster definieren, das dem erwarteten Verhalten entspricht.

Ein Instanzmuster wird definiert durch eine Objektstruktur, die konform zum geforderten Meta-Modell ist. Das Prinzip ist, dass immer, wenn eine Instanz entsprechend der definierten Objektstruktur auftritt, das erwartete Verhalten definiert durch ein Prozessmuster eingehalten werden muss. Prozessmuster spezifizieren wir mit der »Extended Process Pattern Specification Language« (EPPSL), die wir nachfolgend einführen.

Grundlagen zu Prozessmustern

Eine Spezifikationssprache für Prozessmuster wurde in der Dissertation von Alexander Förster [Forster2009] als »Process Pattern Specification Language« (PPSL) vorgestellt. Diese Sprache wurde in [Khaluf2010][Khaluf2011] erweitert zur »Extended Process Pattern Specification Language« (EPPSL). Wir wollen die EPPSL hier als Grundlage einführen, um sie für die Spezifikation des erwarteten Verhaltens von Sprachelementen einsetzen zu können.

Mit Hilfe von EPPSL können wir Muster für Ausführungspfade eines Prozesses beschreiben. Ein Prozessmuster beschreibt eine erwartete Abfolge von Aktionen. Die erwartete Abfolge wird über temporale Beziehungen zwischen den Aktionen spezifiziert. So können wir mit der EPPSL ausdrücken, dass nach der Ausführung einer bestimmten Aktion A, irgendwann die Ausführung einer Aktion B erwartet wird. Was nach der Ausführung von A und vor der Ausführung von B im Prozess geschieht, ist aus Sicht des Prozessmusters nicht von Interesse. Entscheidend ist die zeitliche Abfolge und damit die Reihenfolge, in der die Aktionen ausgeführt werden.

Die Syntax der EPPSL ist angelehnt an die Syntax von UML-Aktivitätendiagrammen [UMLSUP23]. Die Übersicht in Abbildung 5.13 zeigt die EPPSL Syntax der Sprachelemente.

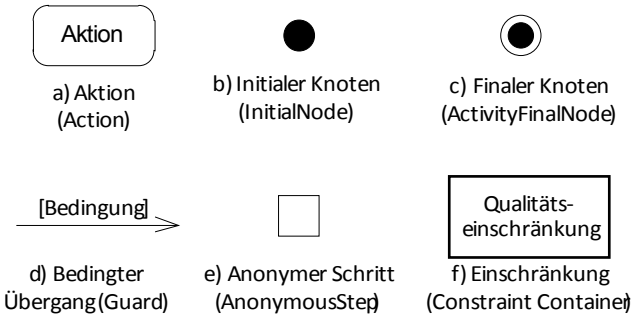


Abbildung 5.13: EPPSL Syntaxelemente

Für die Anwendung der EPPSL zur Beschreibung des erwarteten Verhaltens werden wir die Syntaxelemente a) bis c) benötigen. Mit diesen Dreien können wir Aktionen spezifizieren, die bei der Ausführung auftreten werden. Außerdem können wir den Start (InitialNode) und das Ende (ActivityFinalNode) einer Ausführung festlegen.

Um unterschiedliche temporale Beziehungen zwischen Aktionen ausdrücken zu können, bietet die EPPSL acht verschiedene Operatoren, die wir in Abbildung 5.14 darstellen.

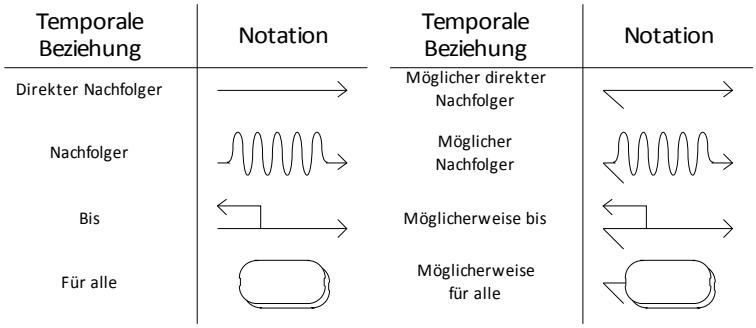


Abbildung 5.14: EPPSL Syntax für temporale Beziehungen

Die einfachste temporale Beziehung zwischen zwei Aktionen A und B ist, dass B ein direkter Nachfolger von A ist. Weicht man diese Eigenschaft etwas auf und fordert nicht, dass B ein direkter Nachfolger von A ist, sondern, dass B irgendwann nach A ausgeführt wird, so kann die Schlängelnotation verwendet werden. Mit der Bis-Beziehung können wir spezifizieren, dass eine Aktion wiederholt ausgeführt werden muss bis eine bestimmte andere Aktion eintritt. Die Für-alle-Notation ist hilfreich, um zu bestimmen, dass eine Beziehung für alle Vorkommen einer Aktion in einer konkreten Ausführung gelten soll.

Für die vier Beziehungen auf der linken Seite in Abbildung 5.14 gibt es jeweils eine erweiterte Variante auf der rechten Seite. Die rechte Variante ist dabei eine Abschwächung zur linken Seite. Man kann ausdrücken, dass eine Aktion B ein möglicher direkter Nachfolger von Aktion A ist. Dies lässt somit die Möglichkeit zu, dass B als direkter Nachfolger existiert oder auch nicht. Analog verhält es sich mit den weiteren Beziehungen. Die Möglicherweise-für-alle-Notation bedeutet somit, dass etwas nicht für alle Aktionen, sondern für mindestens eine Aktion in der gesamten Ausführung gelte.

Für die Anwendung der EPPSL zur Beschreibung des erwarteten Verhaltens werden wir hauptsächlich auf die Schlängelnotation zurückgreifen, um auszudrücken, dass das erwartete Verhalten einer bestimmten zeitlichen Abfolge von Schritten entspricht.

Die temporalen Beziehungen der EPPSL entsprechen den Möglichkeiten, die als temporal-logische Formeln mit der »Computation Tree Logic« (CTL) [Clarke1986] ausgedrückt werden können. Daher können EPPSL-Ausdrücke, wie in [Khaluf2010] gezeigt, direkt in CTL-Formeln übersetzt werden. Diese Eigenschaft werden wir im Testverfahren ausnutzen, um aus der Spezifikation des erwarteten Verhaltens durch Prozessmuster auswertbare CTL-Formeln abzuleiten.

Nach Einführung der EPPSL für Prozessmuster können wir im nachfolgenden Abschnitt die Definition von Verhaltensmustern als neue Modellierungstechnik einführen.

Spezifikation von Verhaltensmustern

Verhaltensmuster spezifizieren Anforderungen an das geforderte Ausführungsverhalten einer gesuchten Sprache. Wir wollen Verhaltensmuster an einem Beispiel einführen. Abbildung 5.15 zeigt das Verhaltensmuster für den Beginn der Ausführung einer Instanz der BehaviorDL, dessen Meta-Modell wir in Abbildung 5.11 gezeigt haben. Im oberen Teil eines Verhaltensmusters ist das Instanz-, im unteren Teil das Prozessmuster definiert.

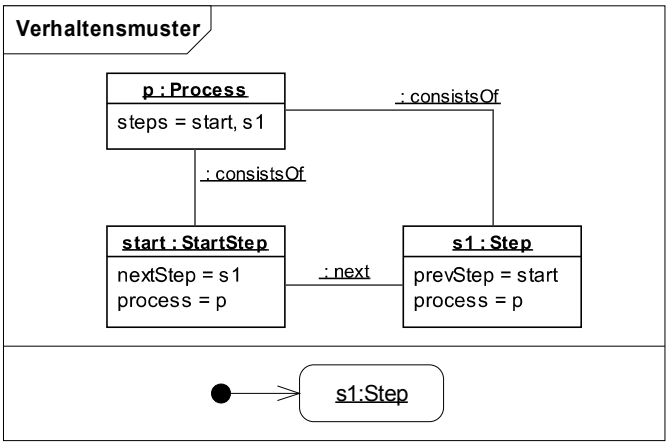


Abbildung 5.15: Verhaltensmuster für Ausführungsbeginn der BehaviorDL

Das Instanzmuster in Abbildung 5.15 beschreibt die Situation, in der ein Process p mit einem StartStep start und einem Step s1 verlinkt ist. Das Muster sagt nichts darüber aus, welche Links und Objekte darüber hinaus noch vorhanden sein könnten. Es stellt eine minimale Objektkonstellation dar, in der immer ein bestimmtes Verhalten erwartet wird. Im Beispiel ist der StartStep start über den next Link mit dem Step s1 verknüpft. Für das Verhalten stellt sich die Frage, wie sich Instanzen, die diesem Instanzmuster entsprechen, bei Ausführung verhalten.

Das zugehörige Prozessmuster im unteren Teil von Abbildung 5.15 zeigt, dass der strukturelle Zusammenhang zwischen Process p, StartStep start und dem Step s1 dazu führt, dass die Ausführung einer solchen Instanz mit der Ausführung des Step s1 beginnt. Das Verhaltensmuster definiert, dass direkt nach Initialisierung die Ausführung des Elementes Step s1 angestoßen wird. Das Prozessmuster beschreibt die erwartete Reihenfolge, in der die Ausführung von Sprachelementen erfolgt. Dabei konzentrieren sich Prozessmuster auf die Ausführung von relevanten Sprachelementen. Im Beispiel in Abbildung 5.15 ist nicht definiert, wann die Elemente Process p und StartStep start ausgeführt werden. Es ist jedoch festgelegt, dass bei Auftreten eines solchen Instanzmusters in jedem Fall das Element Step s1 am Beginn der Ausführung steht.

Betrachten wir ein weiteres Beispiel in Abbildung 5.16. In diesem Beispiel beschreibt das Instanzmuster die Situation, in der zwei Steps s2 und s3 über einen next Link miteinander verbunden sind.

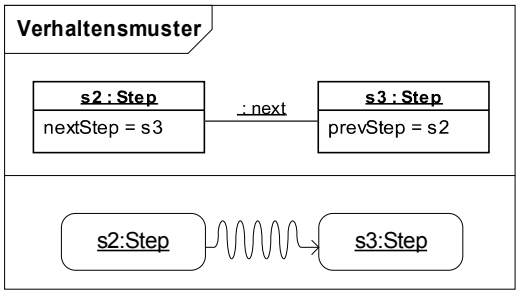


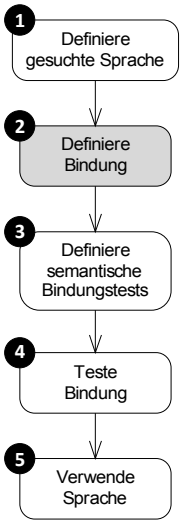
Abbildung 5.16: Verhaltensmuster für Schritte der BehaviorDL

In der durch das Instanzmuster beschriebenen Situation erwarten wir, dass bei Ausführung zunächst Step s2 und irgendwann danach Step s3 ausgeführt wird. In EPPSL können wir dieses erwartete Verhalten durch die Schlängelnotation ausdrücken. Das Prozessmuster legt fest, dass in einer solchen Konstellation immer der Step s2 vor einem Step s3 ausgeführt wird. So haben wir indirekt die erwartete Semantik der next Beziehung durch ein Prozessmuster festgelegt.

Durch die Beschreibung weiterer Verhaltensmuster können wir das erwartete Verhalten einer gesuchten Sprache weiter festlegen. Analog zum Testen von Software bedeuten mehr Verhaltensmuster mehr Tests. Mehr Tests von guter Qualität bedeuten ein größeres Vertrauen darauf, dass eine konkrete Sprache die gestellten Anforderungen erfüllt.

Wir haben in diesem Abschnitt gezeigt, wie man ausgehend von Anwendungsfällen Meta-Modelle als Anforderungen an Struktur und Konzepte einer gesuchten Sprache erstellen kann. Außerdem haben wir mit der Definition von Verhaltensmustern eine Möglichkeit geschaffen, um Anforderungen an das Ausführungsverhalten zu spezifizieren. Damit haben wir Schritt 1 unseres Vorgehens aus Abbildung 5.8 abgeschlossen und widmen uns in Abschnitt 5.3 dem Schritt 2 zur Definition der Bindung mittels parametrisierter Meta-Modelle.

5.3 Parametrisierte Meta-Modelle



Im Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen benötigen wir in Schritt 2 aus Abbildung 5.8 eine Modellierungstechnik, die es erlaubt die Bindung zwischen zwei Sprachdefinitionen, in unserem Fall zwischen zwei Meta-Modellen, zu definieren. Es gibt für diese Aufgabe bereits eine Reihe von Ansätzen insbesondere aus dem Bereich der Modelltransformation, bei der ebenfalls eine Bindung benötigt wird, um die Transformation durchzuführen. Uns ist jedoch kein Ansatz bekannt, der sich nahtlos in den Problemkontext dieser Arbeit eingliedern würde, so dass wir mit den parametrisierten Meta-Modellen eine neue Modellierungstechnik entwickelt haben, die eine konzeptionell neue Sicht auf die Bindung und Wiederverwendung von Sprachen ermöglicht. Für eine Betrachtung verwandter Arbeiten in diesem Kontext sei auf Abschnitt 5.6 verwiesen.

In Abschnitt 5.3.1 werden wir die Sichtweise von Meta-Modellen als Parameter einführen und neben einigen Grundlagen zur Meta-Modellierung auf die konkrete Technik der Meta-Modellierung mit Parametern eingehen. Im Abschnitt 5.3.2 zeigen wir anschließend wie eine Bindung zwischen zwei Meta-Modellen basierend auf dem Konzept der Parametrisierung konkret umgesetzt werden kann.

5.3.1 Meta-Modelle als Parameter

Parametrisierte Meta-Modelle sind eine neue Modellierungstechnik zur Definition von Sprachen, bei denen für bestimmte Teilaspekte existierende Sprachen wiederverwendet werden können. In einem parametrisierten Meta-Modell wird der Teilaspekt, für den eine andere Sprache wiederverwendet werden soll, als *formaler Parameter* des Meta-Modells aufgefasst. Ein formaler Parameter ist selbst wieder ein Meta-Modell. Durch Bindung des formalen Parameters an eine konkrete Sprache, kann das Meta-Modell des formalen Parameters durch das Meta-Modell der konkreten Sprache ersetzt werden. So wird eine Sprache auf Ebene der Meta-Modelle wiederverwendet.

Bevor wir auf die Technik der Meta-Modellierung mit Parametern genauer eingehen, betrachten wir im Folgenden einige Grundlagen zur Meta-Modellierung, auf die wir aufbauen werden.

Grundlagen zur Meta-Modellierung

Meta-Modelle dienen in der Softwaretechnik unter anderem zur Definition der abstrakten Syntax einer Sprache. Die Definition von Modellierungssprachen mittels Meta-Modellierung ist mit der Verbreitung der UML [UML-SUP23] eine etablierte Technik zur Sprachdefinition. Im Zuge der Forschung

rund um die modellgetriebene Softwareentwicklung haben sich Meta-Modelle als eine Standardtechnik in der Softwaretechnik für die Spezifikation domänenspezifischer Sprachen etabliert (vergl. [Emerson2006]).

Meta-Modelle werden typischerweise als Klassenmodell in einer graphischen Repräsentation mit objektorientierten Konzepten spezifiziert. So besteht ein Meta-Modell aus benannten Klassen und benannten Assoziationen zwischen diesen. Die Benennung der Klassen verweist auf die eingesetzten Konzepte innerhalb des Meta-Modells. In einem Meta-Modell werden somit sowohl Strukturen als auch eingesetzte Konzepte definiert. Spricht man von der abstrakten Syntax einer Sprache ist hiermit die Kombination aus Struktur und eingesetzten Konzepten gemeint.

Zusätzlich zur abstrakten Syntax, hat eine Sprache in der Regel eine konkrete Syntax, die jedoch nicht Teil des Meta-Modells ist. Ein Satz, der in der definierten Sprache ausgedrückt wird, ist eine Instanz des Meta-Modells, das die Sprache definiert. Daher liegen das Meta-Modell einer Sprache und die konkreten Sätze einer Sprache auf unterschiedlichen linguistischen Ebenen. Abbildung 5.17 zeigt schematisch die Beziehung zwischen linguistischer Definitions- und Instanzebene.

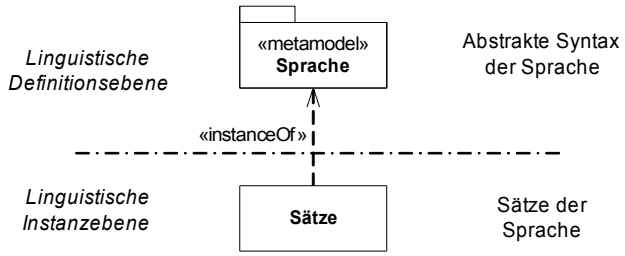


Abbildung 5.17: Linguistische Definitions- und Instanzebene

Das Grundprinzip der Trennung in Ebenen zwischen Sprachdefinition und Instanz kann man wiederholt anwenden, um Sprachdefinitionen aufeinander aufbauen zu lassen. Die UML verwendet für ihre Sprachdefinition ein Vier-Ebenen-Konzept [UMLSUP23] zu sehen in Abbildung 5.18. Auf der obersten vierten Ebene, die als M3 bezeichnet wird, steht eine Meta-Sprache, genannt »Meta Object Facility« (MOF) [MOF20], die dazu in der Lage ist, die Sprachdefinition für das UML-Meta-Modell auf der darunter liegenden Ebene M2 auszudrücken. Dabei ist die MOF über sich selbst definiert, so dass das Meta-Modell der MOF die MOF selbst ist.

Das UML-Meta-Modell auf der dritten Ebene (M2) definiert die Sprachkonstrukte, die man mit der UML ausdrücken kann. Zur Definition des UML-Meta-Modells wird die MOF der vierten Ebene (M3) verwendet, so dass das UML-Meta-Modell eine Instanz der MOF ist. Im UML-Meta-Modell wird beschrieben aus welchen Sprachelementen z.B. ein UML-Klassendiagramm oder Sequenzdiagramm besteht und wie die Sprachelemente miteinander in Beziehung stehen.

Auf der zweiten Ebene (M1) finden sich die konkreten Modelle, die Instanzen der UML sind. Ein konkretes Aktivitätendiagramm ausgedrückt in seiner konkreten Syntax befindet sich auf dieser Ebene.

Die unterste Ebene M0 enthält keine Sprachdefinitionen, sondern steht für die reale Welt, von der man mittels der UML ein Modell erstellt. Auf dieser Ebene findet sich also zum Beispiel der real gelebte Geschäftsprozess innerhalb eines Unternehmens, der dann mit Hilfe der UML in ein Aktivitätendiagramm auf Ebene M1 abgebildet werden kann. Die folgende Abbildung 5.18 zeigt die vier Ebenen von M3 bis M0.

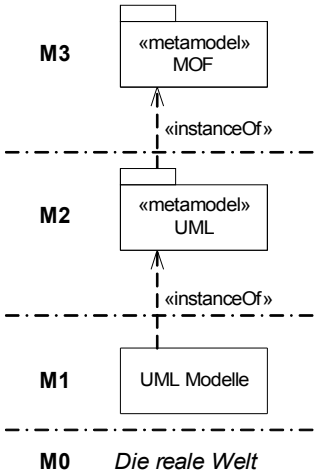


Abbildung 5.18: Meta-Ebenen der UML

Mit der MOF als Meta-Sprache können Meta-Modelle definiert werden. Wir betrachten daher die Möglichkeiten der MOF ein wenig genauer. Die MOF bietet drei Sprachkonstrukte für die Modularisierung und entsprechender Zusammenführung von Meta-Modellen. In der MOF werden Meta-Modelle in Pakete gekapselt. Durch die Aufteilung eines komplexen Meta-Modells in Pakete, kann das Meta-Modell in Einheiten gegliedert werden. Jedes Paket steht dabei für eine Untermenge von Sprachelementen, die in der Lage ist, einen bestimmten Teilaspekt der Gesamtsprache auszudrücken. In diesem Sinne enthält ein Paket ein (Teil-) Meta-Modell. Um die einzelnen Teilaspekte in Form von Paketen zu einem gesamten Meta-Modell wieder zusammenzusetzen und dabei Pakete wiederzuverwenden stellt die MOF die zwei Operationen «import» und «merge» zur Verfügung, die im weiteren Verlauf noch im Detail vorstellen werden. Betrachten wir jedoch zunächst den neuen Ansatz der Meta-Modellierung mit Parametern.

Meta-Modellierung mit Parametern

Parametrisierte Meta-Modelle bilden eine Erweiterung der Meta-Modellierung auf der M3-Ebene (vergl. Abbildung 5.18). In den Grundlagen zur Meta-Modellierung haben wir beschrieben, was ein einzelnes Meta-Modell zur Definition einer Sprache ausmacht. Nun wollen wir die Technik der Meta-Modelle so erweitern, dass ein Meta-Modell darüber hinaus andere Meta-Modelle im Sinne der Wiederverwendung integrieren kann. Auf abstrakter Ebene zeigt Abbildung 5.19 in Anlehnung an Abbildung 4.2 das gedachte Konzept.

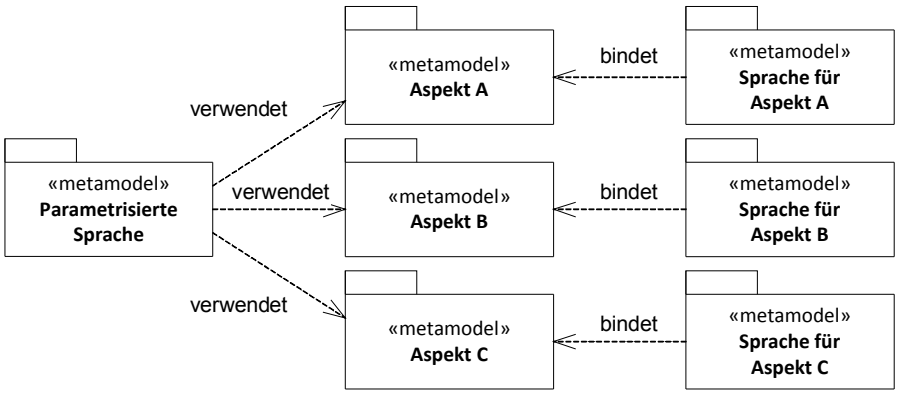


Abbildung 5.19: Konzept eines parametrisierten Meta-Modells

In Abbildung 5.19 sehen wir, dass eine parametrisierte Sprache als Paket modelliert werden kann, das unterschiedliche Aspekte in Form weiterer Pakete einbezieht. Ergebnis ist ein modularisierter Aufbau der parametrisierten Sprache, bei dem Teilaspekte, im Beispiel die Aspekte *A* bis *C*, in eigene Meta-Modelle ausgegliedert und durch die Beziehung verwendet im Meta-Modell Parametrisierte Sprache wieder kombiniert werden.

Die wichtige Neuerung aus dem Konzept parametrisierter Sprachen ist die Sichtweise, dass die Meta-Modelle der Teilaspekte als formale Parameter interpretiert werden. Das Meta-Modell des formalen Parameters fungiert als Platzhalter für eine Sprache, die den gewünschten Aspekt ausdrücken kann. Um eine Sprache für einen bestimmten Aspekt zu verwenden, muss diese Sprache an den formalen Parameter gebunden werden. In Abbildung 5.19 wird die Bindung durch die Beziehung bindet ausgedrückt, die besagt, dass zum Beispiel Sprache für Aspekt A an den Aspekt A gebunden wird. Ein formaler Parameter legt fest, welche anderen Meta-Modelle für eine Bindung an diesen Parameter zugelassen sind. Ist ein anderes Meta-Modell an einen Parameter gebunden, kann der formale Parameter durch dieses andere Meta-Modell ersetzt werden.

Zur Umsetzung des Konzeptes aus Abbildung 5.19 verwenden wir die MOF. Wir werden hierzu das Meta-Modell der MOF um die neuen Stereotypen «parameter» und «bindingOf» erweitern. Die verwendete Version der MOF ist 2.0 [MOF20]. Die Teile des MOF-Meta-Modells, die für unsere Erweiterung von Interesse sind, sind in der UML-Infrastruktur Spezifikation definiert, die wir in Version 2.3 [UMLINF23] betrachten.

Abbildung 5.20 zeigt den für unsere Erweiterung relevanten Ausschnitt aus dem Meta-Modell der MOF bzw. der UML-Infrastruktur [UMLINF23]. Der Ausschnitt zeigt die relevanten Klassen für die Verwendung von Paketen (»Package Diagram«). Es wird dort definiert, dass in einem Paket (Package) paketierbare Elemente (PackageableElement) abgelegt werden können, wozu Klassen, Assoziationen und Pakete selbst gehören. Ein Paket kann somit Unterpakete in Form von eingebetteten Paketen (nestedPackage) besitzen.

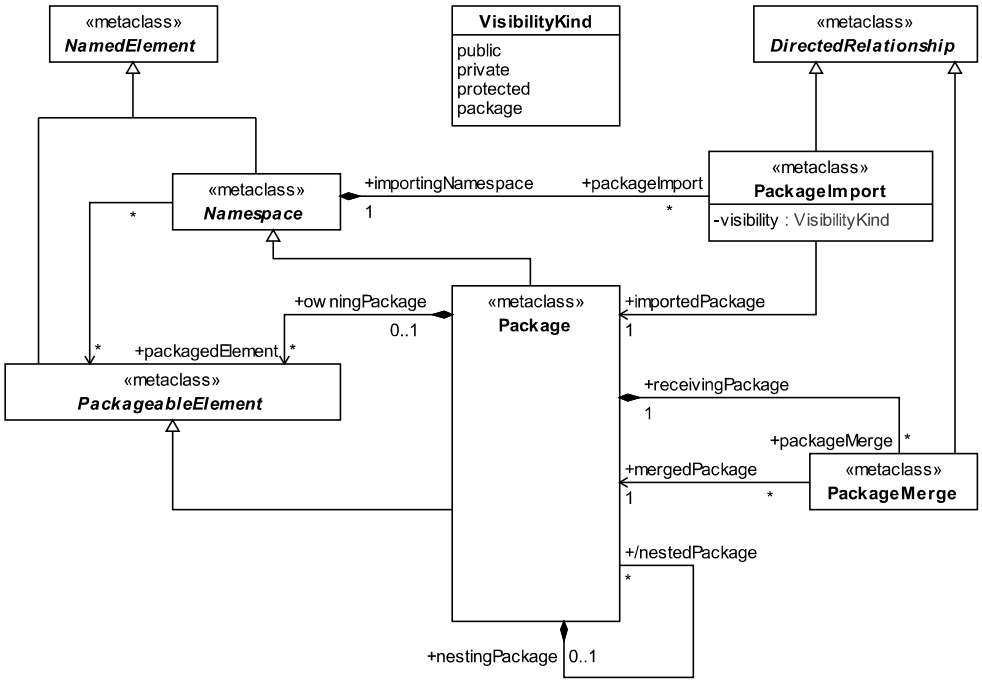


Abbildung 5.20: MOF-Meta-Modell für »Package Diagram« aus [UMLINF23]

Die UML-Infrastruktur definiert für Pakete zwei Arten von gerichteten Beziehungen (DirectedRelationship), nämlich einen Paketimport (PackageImport, «import») und eine Paketmischung (PackageMerge, «merge»). Wir wollen untersuchen, warum diese beiden Beziehungen für die Umsetzung des Konzeptes der Parametrisierung nicht ausreichen.

Der Paketimport bildet eine direkte Verbindung zwischen einem Namensraum (Namespace) und einem Paket. Da ein Paket selber durch Vererbung wieder ein Namensraum ist, kann ein Paketimport nur zwischen zwei Paketen erfolgen. Semantisch ist der Import über einen Import von Namensräumen definiert. Wenn ein Paket A ein anderes Paket B importiert, bedeutet dies, dass alle Elemente aus B in A bekanntgemacht werden und somit für die Modellierung zur Verfügung stehen. Anders ausgedrückt wird in Paket A ein Verweis auf Elemente aus B angelegt, wobei die Elemente, auch bei Namensgleichheit, durch ihre Herkunft eindeutig unterscheidbar bleiben.

Aus der Spezifikation der UML-Infrastruktur entnehmen wir bezüglich der Semantik für Namensräume folgendes:

> „A namespace provides a container for named elements. It provides a means for resolving composite names, such as name1::name2::name3. The member association identifies all named elements in a namespace called N that can be referred to by a composite name of the form N::<x>.“ [UMLINF23]

Ein Namensraum bildet einen Container für benannte Elemente, wobei ein Namensraum wieder andere Namensräume enthalten kann. So entsteht eine zusammengesetzte Namensgebung, bei der die Namen der benannten Elemente durch »::« aneinandergefügt werden, wie im Beispiel »name1::name2::name3«. So können alle benannten Elemente eindeutig identifiziert werden, indem man den Namen aus dem Namensraum, dem »::« Trenner und dem Elementnamen zusammensetzt (»N::<x>«).

Pakete können über die abgeleitete Assoziation nestedPackage ineinander geschachtelt werden. Diese Schachtelung über nestedPackage ist eine Untermenge aller zugehörigen Elemente (ownedElement) eines Namensraumes. Durch die Schachtelung kennt das übergeordnete Paket die Elemente der enthaltenen Pakete.

> „The elements that can be referred to using non-qualified names within a package are owned elements, imported elements, and elements in enclosing (outer) namespaces. Owned and imported elements may each have a visibility that determines whether they are available outside the package.“ [UMLINF23]

Ein Import von Paketen wird semantisch erklärt durch den Import aller Elementnamen des importierten Paketes in den Namensraum des importierenden Paketes. Durch den Import werden die importierten Elemente über ihren Namen per Referenz im importierenden Paket bekannt gemacht und können so verwendet werden. Konzeptionell führt die Semantikdefinition für einen Paketimport diesen auf den Import einzelner Elemente zurück.

> „A package import is a relationship between an importing namespace and a package, indicating that the importing namespace adds the names of the members of the package to its own namespace. Conceptually, a package import is equivalent to having an element import to each individual member of the imported namespace, unless there is already a separately-defined element import.“ [UMLINF23]

Importe sowie Schachtelungen von Paketen werden mit einem Attribut für die Sichtbarkeit (visibility) ausgezeichnet. Durch das Attribut visibility der Klasse PackageImport unterscheidet die MOF zwei Arten von Imports. Diese beziehen sich auf die Sichtbarkeit importierter Pakete für Dritte. Importiert ein Paket A ein Paket B und wird A gleichzeitig von C importiert, so lässt sich mit dem visibility Attribut steuern, ob Elemente aus B für C sichtbar werden oder nicht. Hat visibility den Wert public, bedeutet dies, dass importierte Elemente auch für Dritte sichtbar werden. Dies wird in der konkreten Syntax durch eine «import» Beziehung ausgedrückt und ist der Standard. Sollen die importierten Elemente nicht für Dritte sichtbar sein, so wird visibility mit private belegt. Daher nennt man dies auch einen privaten Import. In der konkreten Syntax wird dies durch eine «access» Beziehung ausgedrückt. Wir werden die «access» Beziehung noch ausgiebig bei der Verwendung von Parametern verwenden. Die «access» Bezie-

hung hat den Vorteil, das Parameter mehrfach verwendet werden können, ohne dass es zu Konflikten kommt.

Im Gegensatz zu einem Paketimport werden bei einer Paketmischung die Elemente aus beiden Paketen miteinander kombiniert, so dass bei einer Paketmischung, bei der ein Paket A mit einem Paket B gemischt wird, implizit ein neues Paket entsteht, das die Mischung der Elemente beider Pakete enthält. Im Kern besagt die Semantik der Paketmischung, dass beim Mischen Klassen, die den gleichen Namen haben, zu einer neuen Klasse kombiniert werden.

Wir erkennen, dass wir die Techniken des Paketimports und der Paketmischung für die angestrebte Parametrisierung nicht verwenden können. Wir benötigen vielmehr eine neue Beziehung zwischen Paketen, die besagt, dass ein Paket als formaler Parameter fungieren soll, der durch ein anderes Meta-Modell als konkreten Parameter ersetzt werden kann. Um parametrisierte Meta-Modelle mit der MOF beschreiben zu können, erweitern wir daher das Meta-Modell der MOF aus Abbildung 5.20 um zwei neue Stereotypen. Das erweiterte MOF-Meta-Modell für »Package Diagram« ist in Abbildung 5.21 zu sehen.

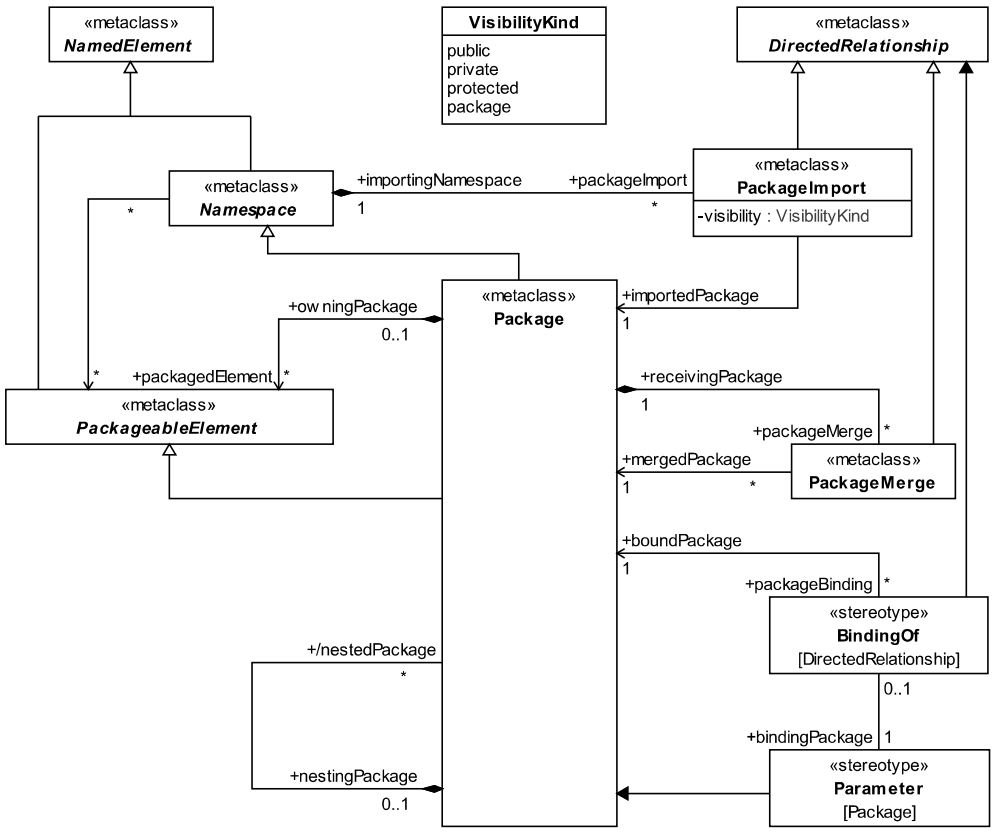


Abbildung 5.21: Erweitertes MOF-Meta-Modell für parametrisierte Pakete

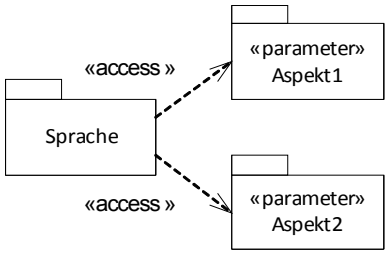
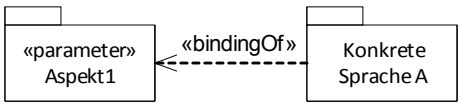
In dieser erweiterten Definition des MOF-Meta-Modells in Abbildung 5.21 führen wir für das Konzept die neue Stereotypen-Klassen Parameter und BindingOf ein. Der Stereotyp Parameter wird von der Meta-Klasse Package abgeleitet, so dass wir Pakete als Parameter deklarieren können.

Der neue Stereotyp BindingOf soll als Beziehung zwischen zwei Paketen verwendet werden können. Daher wird er von der Meta-Klasse DirectedRelationship abgeleitet, die die Oberklasse für alle gerichteten Beziehungen in der MOF bildet. Im Falle von BindingOf ist dies eine gerichtete Beziehung zwischen einem Paket, das eine konkrete Sprache spezifiziert, und einem Paket, das einen Parameter spezifiziert. Objekte des Stereotyps Parameter nehmen hierbei die Rolle des bindenden Paketes (bindingPackage) und Objekte vom Typ Package die Rolle des gebundenen Paketes (boundPackage) ein. Das bindende Paket ist das Meta-Modell des Parameters und das gebundene Paket ist das Meta-Modell der konkreten Sprache. So können wir festlegen welcher Parameter an welche konkrete Sprache gebunden wird.

Es bleibt festzuhalten, dass wir durch die Kombination einer Parametrisierung eines Meta-Modells mit Bindung an ein anderes Meta-Modell das Konzept der parametrisierten Meta-Modelle mit Hilfe des erweiterten MOF-Meta-Modells ausdrücken können. Bisher haben wir hierzu das Meta-Modell, das die abstrakte Syntax darstellt, betrachtet.

Wir führen nun in Tabelle 5.1 eine konkrete Syntax für parametrisierte Meta-Modelle ein, die es uns erlaubt, die Parametrisierung in Modellen graphisch darzustellen. In der konkreten Syntax sehen wir, dass die Neuerung die Verwendung der Stereotypen «parameter» und «bindingOf» ist. Pakete können mit dem Stereotyp «parameter» als formale Parameter eines Meta-Modells ausgezeichnet werden. Möchte man modellieren, dass das Paket einer konkreten Sprache an einen Parameter gebunden werden kann, so verwendet man die «bindingOf» Beziehung.

Tabelle 5.1: Konkrete Syntax für parametrisierte Pakete

Konkrete Syntax	Beschreibung
	Die Teilaspekte Aspekt1 und Aspekt2 einer Sprache werden als formale Parameter deklariert. Die Sprache greift über die «access» Beziehung auf die Parameter zu, wodurch die Meta-Modelle der Parameter für die Sprache sichtbar werden.
	Das Meta-Modell für die Konkrete Sprache A wird an den formalen Parameter Aspekt1 gebunden. Dies setzt eine gültige Bindung zwischen konkreter Sprache und Parameter voraus.

Mit dieser Erweiterung der MOF können wir ein parametrisiertes Meta-Modell modellieren. Im nachfolgenden Abschnitt 5.3.2 befassen wir uns mit den Details einer Bindung, die sich hinter dem syntaktischen Element «bindingOf» verbirgt.

5.3.2 Bindung von Parametern

Wir definieren die Bindung eines formalen Parameters als rechtseindeutige Abbildung zwischen dem Meta-Modell des gebundenen und des bindenden Paketes. Um diese Abbildung zu finden, müssen verwendete Konzepte des formalen Parameters auf Konzepte der konkreten Sprache abgebildet werden können.

Wir verwenden keine formale Semantikspezifikation für Konzepte, sondern gehen von einer impliziten Semantikspezifikation innerhalb des Meta-Modells aus, die sich aus dem intuitiven Sprachgebrauch von Bezeichnern ergibt. Die verwendeten Bezeichner für Klassen, Rollen und Assoziationen interpretieren wir als Verweise auf verwendete Konzepte. Eine Klasse mit dem englischen Namen »State« bezieht sich demnach auf das Konzept eines Zustands. In einer validen Bindung werden Konzepte abgebildet, die sich gemäß ihrer intuitiven Bedeutung entsprechen. Da wir uns nur auf die intuitive Semantik der Konzepte beziehen, kann es zum beschriebenen Problem der Mehrdeutigkeiten kommen, die sich wiederum als semantische Fehler in der Bindung manifestieren. Aus diesem Grund benötigen wir den semantischen Bindungstest.

Zusätzlich zu abbildbaren Konzepten müssen auch strukturelle Eigenschaften des formalen Parameters erfüllt sein. Verfügen formaler Parameter und konkrete Sprache über abbildbare Konzepte, aber lassen sich diese Konzepte nicht gemäß der Struktur aufeinander abbilden, kann die konkrete Sprache nicht verwendet werden. Nur die Kombination aus sich entsprechenden Konzepten unter Einhaltung der Struktur ergibt eine valide Bindung.

Wir wollen im Folgenden auf die Abbildung der Struktur näher eingehen. Hierzu betrachten wir typische Bindungsmuster, die verdeutlichen, welche Fälle bei Abbildung der Struktur zu beachten sind. Wir nehmen in unseren Bindungsmustern an, dass die abgebildeten Klassen gemäß ihrer Konzepte, für die sie stehen, valide abgebildet sind und konzentrieren uns auf die Abbildung der Struktur. Um nicht durch konkrete Bezeichner für Konzepte zu verwirren, verwenden wir daher abstrakte Bezeichner.

In unserem ersten Bindungsmuster für Kardinalitäten in Abbildung 5.22 haben wir ein Meta-Modell MM mit einem Parameter P und das Meta-Modell einer Sprache L, das an den Parameter P gebunden wird. Die Abbildung zwischen P und L zeigen wir in Abbildung 5.22.

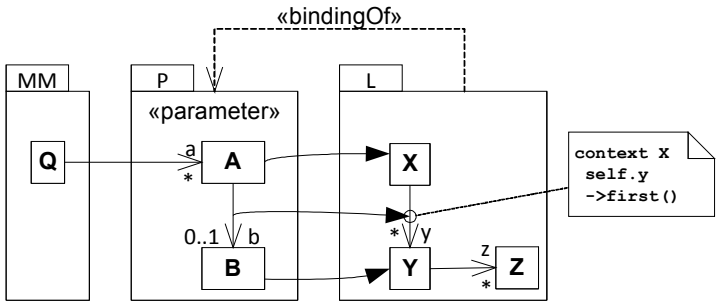


Abbildung 5.22: Bindungsmuster für Kardinalitäten

Das Meta-Modell MM definiert die Klasse Q, die auf Klasse A des Parameters P verweist. Klasse A im Parameter ist über die Rolle b mit Kardinalität 0..1 wiederum assoziiert mit Klasse B. Für eine valide Bindung müssen wir das Meta-Modell P mit den Klassen A und B auf das Meta-Modell in L mit den Klassen X, Y und Z abbilden. Hierzu bilden wir A auf X und B auf Y ab. Um nun die Assoziation zwischen A und B korrekt abzubilden, müssen wir die Kardinalitäten beachten. Zwischen X und Y hat die Assoziation mit Rolle y die Kardinalität *, was gröber ist als die Kardinalität 0..1 zwischen A und B. Daher benötigt die Bindung der Assoziation gemäß eine zusätzliche Einschränkung, die wir mit der »Object Constraint Language« (OCL) [OCL20] in Listing 5.1 spezifizieren.

Listing 5.1: Einschränkung eines Mappings mittels OCL

```
1  context X
2     self.y
3     -> first()
```

Die Einschränkung besagt, dass im Kontext von X die Assoziation mit der Rolle y immer so ausgewertet wird, dass aus der möglichen Menge der Objekte von Y immer das erste Element (first()) gewählt wird. So stellen wir sicher, dass die Kardinalität 0..1 korrekt abgebildet wird. Außerdem zeigt das Beispiel, dass das Meta-Modell L mehr Elemente enthalten kann, als von der Bindung benutzt. Durch diese partielle Abbildung können so auch nur Teile einer Sprache wiederverwendet werden, denn aus Sicht des Parameters P ist die Klasse Z nicht relevant und somit nicht Teil der Abbildung. Betrachtet man umgekehrt die Situation aus Sicht von L, so wird L auf P reduziert.

Im gezeigten Bindungsmuster für Kardinalitäten wurden mögliche Vererbungshierarchien innerhalb der Meta-Modelle nicht berücksichtigt. Bei Verwendung von Vererbungen zwischen Klassen ist zu beachten, dass ein vorhandener Vererbungsbaum auf einen kompatiblen Vererbungsbaum abgebildet werden muss. Wir wollen dies in einem Bindungsmuster für Vererbungshierarchien in Abbildung 5.23 verdeutlichen.

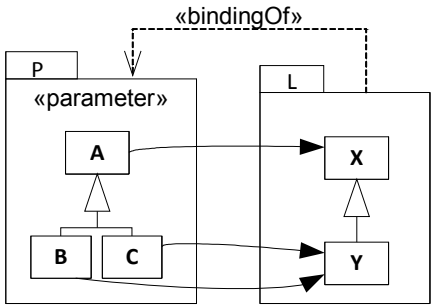


Abbildung 5.23: Bindungsmuster für Vererbungshierarchien

Im Bindungsmuster für Vererbungshierarchien muss der Vererbungsbaum bestehend aus den Klassen A, B und C des formalen Parameters P auf das Meta-Modell L abgebildet werden. Um die Vererbungshierarchie zu erhalten, bilden wir nun A auf X ab und beide Unterklassen von A, also B und C, auf die einzige

Unterklasse von X nämlich Y. Durch Abbildung der beiden Klassen B und C auf Y stellen wir sicher, dass die Vererbungshierarchie korrekt erhalten bleiben.

Für den Fall, dass auf der rechten Seite keine Vererbungshierarchie vorhanden ist, kann es vorkommen, dass eine Vererbungshierarchie in einer einzelnen Klasse vereinigt wird. Wir betrachten dieses Bindungsmuster in Abbildung 5.24.

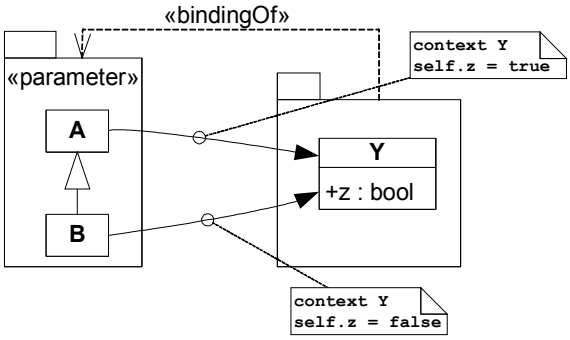


Abbildung 5.24: Bindungsmuster für vereinigende Vererbungshierarchien

Bei Vereinigung einer Vererbungshierarchie in einer einzelnen Klasse, muss die Klasse Y in Abbildung 5.24 ein Attribut anbieten, dass es erlaubt, die Abbildung der Klassen A und B zu unterscheiden. Nur so bleibt die geforderte Rechtseindeutigkeit der Abbildung gewahrt. Dieses Bindungsmuster für vereinigende Vererbungshierarchien kommt zur Anwendung, wenn eine Vererbung in einem Meta-Modell über Attribute modelliert wurde.

Betrachten wir ein weiteres Bindungsmuster, das entsteht, wenn eine Assoziation des Parameters nicht eins-zu-eins auf eine Assoziation der Sprache abgebildet werden kann. In der Praxis sind die Strukturen der Meta-Modelle selten so identisch, dass eine eins-zu-eins Ersetzung möglich wäre. Aus diesem Grund erlauben wir, einzelne Assoziationen des Parameters auf eine hintereinander geschaltete Kette von Assoziationen in der Sprache abzubilden.Wir zeigen dieses Bindungsmuster in Abbildung 5.25.

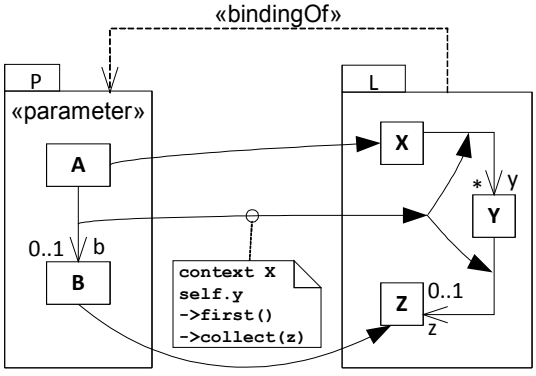


Abbildung 5.25: Bindungsmuster für Assoziationsketten

Bei diesem Bindungsmuster ist zu beachten, dass Anfang und Ende der einzelnen Assoziation im Parameter korrekt auf Anfang und Ende der Assoziationskette in der Sprache abgebildet werden. Die Idee ist, einzelne Kanten, also Assoziationen des Parameters, auf azyklische Pfade, genannt Assoziationsketten im Meta-Modell der Sprache, abzubilden.

Die Klassen A und B des Parameters P werden auf X und Z des Meta-Modells L abgebildet. Es existiert jedoch keine direkte Assoziation in L zwischen X und Z, um die vorhandene Assoziation von A nach B in P eins-zu-eins abbilden zu können. Wir können jedoch die Assoziation von A nach B auf die Kette der Assoziationen von X über Y nach Z abbilden. Um hierbei die Kardinalitäten korrekt einzuhalten, benötigen wir für diese Abbildung wiederum eine zusätzliche Einschränkung, die wir mittels OCL in Listing 5.2 formulieren.

Listing 5.2: Einschränkung der Abbildung einer Assoziation

```
1  context X
2     self.y
3     -> first()
4     -> collect(z)
```

Um die Assoziationskette in L auf die Kardinalität 0..1 zu beschränken, selektieren wir im Kontext von X die Rolle y, wählen hiervon mit first() das erste Element und schauen dann nach einem assoziierten Element vom Typ Z über collect(z). So erhalten wir maximal ein Element über die Assoziationskette und beschränken die Kardinalität damit wie gefordert.

Abschließend betrachten wir in Abbildung 5.26 ein Bindungsmuster für die Abbildung von komplexen Attributkonstellationen.

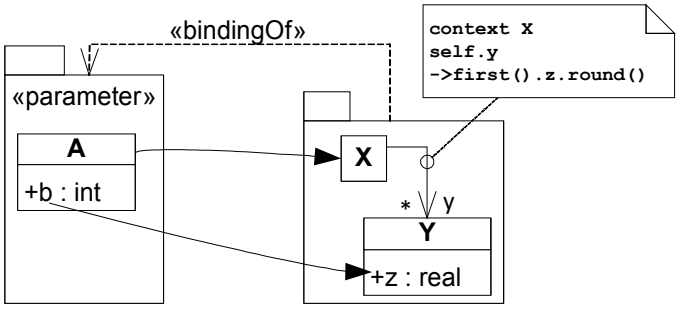


Abbildung 5.26: Bindungsmuster für Attribute

Beim Bindungsmuster für Attribute müssen wir mit Hilfe von OCL-Ausdrücken beschreiben, was bei der Umrechnung der Werte eines Attributs geschehen soll. In Abbildung 5.26 sehen wir das Beispiel einer Umrechnung von real zu int durch den Rundungsoperator round(). Außerdem sind Kardinalitäten zu beachten, wenn ein Attribut, wie im Bindungsmuster gezeigt, in einer assoziierten Klasse abgebildet wird.

In den gezeigten Bindungsmustern haben wir die Bindung zwischen Parameter und Sprache nur schematisch in einer Grafik skizziert. Für die Formulierung einer Bindung, definieren wir daher eine Syntax, die es uns erlaubt, jede Bindung in einer extensionalen Notation zu spezifizieren. Die grafischen Visuali-

sierungen dienen lediglich der Veranschaulichung. Wir definieren die Bindungs-Syntax durch die Grammatik in Listing 5.3.

Listing 5.3: Bindungs-Syntax in EBNF

```
1  mapping = "{" tuples "}" ;
2  tuples  = tuple ["," tuple] ;
3  tuple   = "(" IDENT "," expr ")" ;
4  expr    = ( IDENT | "[" ilist "]" ) [ constr ] ;
5  ilist   = IDENT ["," ilist] ;
6  constr  = "<" OCL ">" ;
```

Ein `IDENT` kann der Name einer Klasse oder auch der qualifizierte Name einer Assoziation als Punktnotation nach dem Schema <Klassenname>.<Rolle der Assoziation> sein. Der Platzhalter `OCL` steht für einen beliebigen OCL-Ausdruck, über den, wie in den Beispielen zu sehen, weitere Einschränkungen formuliert werden können.

Mit Hilfe dieser Syntax können wir das in Abbildung 5.25 zu sehende Bindungsmuster für Assoziationsketten wie folgt ausdrücken:

Listing 5.4: Spezifikation einer Bindung für Beispiel 3

```
1  {
2     (A, X),
3     (B, Z),
4     (A.b, [X.y, Y.z] <context X
5                          self.y
6                          -> first()
7                          -> collect(z)> )
8  }
```

Wir definieren die Bindung als eine Menge von Tupeln, wobei jedes Tupel ein Element der linken Seite auf ein Konstrukt der rechten Seite gemäß der Definition für ein valides Mapping abbildet. So wird `A` auf `X` und `B` auf `Z` abgebildet. Zur Abbildung einer Assoziation `A.b` auf eine Assoziationskette, wird die rechte Seite in eckigen Klammern als Liste definiert. Im Beispiel ist die Assoziationskette durch die Liste `[X.y, Y.z]` festgelegt. Werden für die korrekte Umsetzung von Kardinalitäten weitere Einschränkungen benötigt, so können diese als OCL-Ausdruck in spitzen Klammern angefügt werden.

Nach diesen theoretischen Überlegungen zur Spezifikation der Bindung wollen wir im Folgenden ein konkretes Beispiel einer Bindung zwischen zwei Meta-Modellen beschreiben.

Beispiel einer Bindung

In Abbildung 5.11 haben wir das Meta-Modell der BehaviorDL definiert, die das Verhalten eines Frameworks beschreiben kann. Ein möglicher Kandidat für eine Sprache, die diesem geforderten Meta-Modell entsprechen könnte, sind UML-Aktivitätendiagramme. Wir wollen in diesem Beispiel zeigen, wie das geforderte Meta-Modell und Meta-Modell für UML-Aktivitätendiagramme gebunden werden können. Zunächst betrachten wir in Abbildung 5.27 das UML::Activity Meta-Modell für UML-Aktivitätendiagramme.

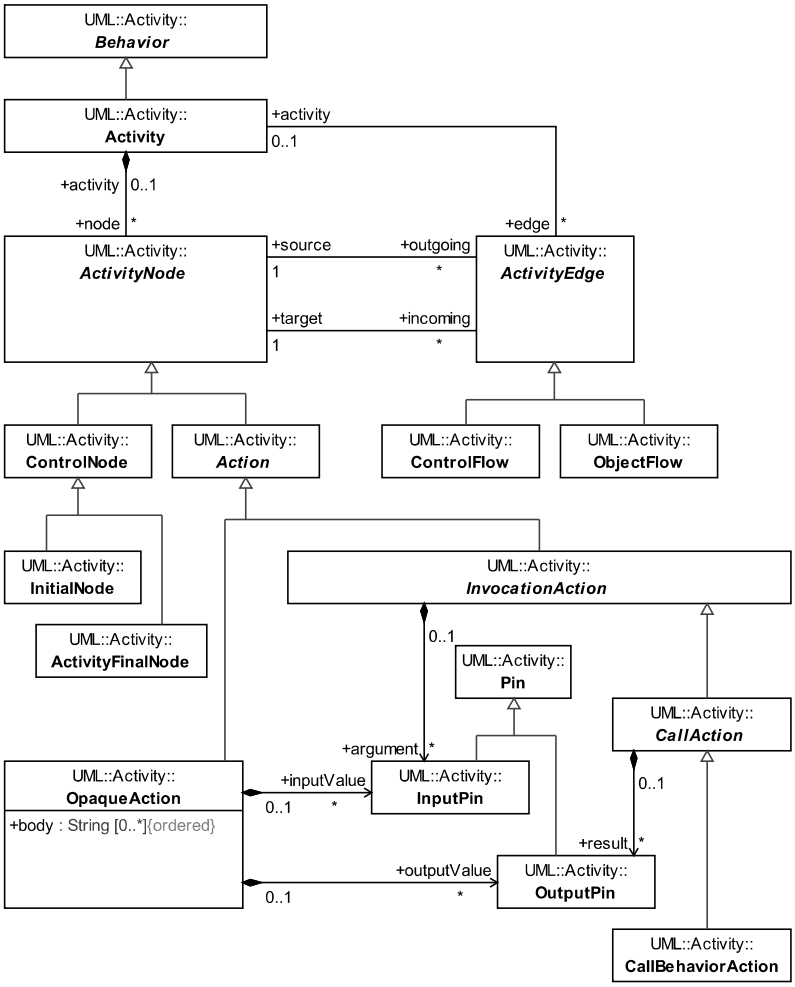


Abbildung 5.27: UML::Activity Meta-Modell

Ein UML-Aktivitätendiagramm (Activity) besteht aus Knoten (ActivityNode), die über Kanten (ActivityEdge) miteinander verbunden sind. Es werden verschiedene Arten von Knoten unterschieden, um unter anderem Kontrollstrukturen in einem Aktivitätendiagramm auszudrücken. Beispiele für Kontrollknoten (ControlNode) sind der initiale Knote (InitialNode), der bestimmt, wo die Ausführung eines Aktivitätendiagramms beginnt und der finale Knoten (ActivityFinalNode), der bestimmt, wo die Ausführung terminiert. Eine andere Form von Knoten sind Aktionen (Action), die in einem Aktivitätendiagramm ausgeführt werden. Auch hier gibt es wieder verschiedene Unterarten von Aktionen. Die allgemeinste Art von Aktion ist die OpaqueAction, die für jede Art von Aktion stehen kann.

Wir wollen nun das Meta-Modell der BehaviorDL aus Abbildung 5.11 auf das UML::Activity Meta-Modell in Abbildung 5.27 rechtseindeutig abbilden. Hierzu verwenden wir die Bindungs-Syntax aus Listing 5.3. Die resultierende Bindung zeigt Listing 5.5. Eine grafische Darstellung der Bindung zur Veranschaulichung zeigt Abbildung 5.28.

Listing 5.5: Bindung zwischen BehaviorDL und UML::Activity

```
1  {
2     (Process, Activity),
3     (Step, ActivityNode
4          <context ActivityNode
5             self.oclIsTypeOf(OpaqueAction)>),
6     (StartStep, InitialNode),
7     (FinalStep, ActivityFinalNode),
8     (SubProcess, CallBehaiorAction),
9     (HotSpotGroupCall, OpaqueAction
10         <context OpaqueAction
11            self.body.indexOf("HotSpotGroupCall") = 1> ),
12    (Process.steps, Activity.node),
13    (Step.process, ActivityNode.activity),
14    (Step.nextStep, [ActivityNode.outgoing, ControlFlow.target]
15         <context ActivityNode
16            self.outgoing
17            -> first()
18            -> collect(target)
19            -> first()> ),
20    (Step.prevStep, [ActivityNode.incoming, ControlFlow.source]
21         <context ActivityNode
22            self.incoming
23            -> first()
24            -> collect(source)
25            -> first()> )
26 }
```

Die Bindung stellt die Verbindung zwischen einem Process der BehaviorDL und einer Activity in UML-Aktivitätendiagrammen her. Ausgehend von diesen beiden Elementen ergeben sich die weiteren Bindungen. Einige Bindungen sind relativ einfach, andere erfordern eine Reihe komplexerer Einschränkungen, die mit Hilfe von OCL definiert sind. Einschränkungen sind insbesondere dann notwendig, wenn bestimmte Assoziationen wie Step.nextStep und Step.prevStep abgebildet werden, bei denen die Kardinalitäten beachtet werden müssen.

In der grafischen Darstellung in Abbildung 5.28 erkennt man die Zusammenhänge ein wenig leichter, wenn auch Details der Einschränkungen mittels OCL in dieser Darstellung verloren gehen. Man erkennt, dass ein Step auf einen ActivityNode abgebildet wird. Analog verfahren wir mit den spezialisierten Klassen von Step und bilden diese auf die entsprechenden Spezialisierungen von ActivityNode ab. Hierdurch ist sichergestellt, dass die Vererbungshierarchie während der Abbildung eingehalten wird. Eine Besonderheit betrifft den HotSpotGroupCall. Dieser wird auf eine OpaqueAction abgebildet, deren Attribut body mit der Zeichenkette »HotSpotGroupCall« beginnt.

Mit der Bindung auf Basis eines parametrisierten Meta-Modells als Schritt 2 unsere Vorgehens in Abbildung 5.8 haben wir die Voraussetzung geschaffen, um eine Sprache wiederzuverwenden. Im nächsten Abschnitt 5.4 wird diese Bindung im semantischen Bindungstest überprüft.

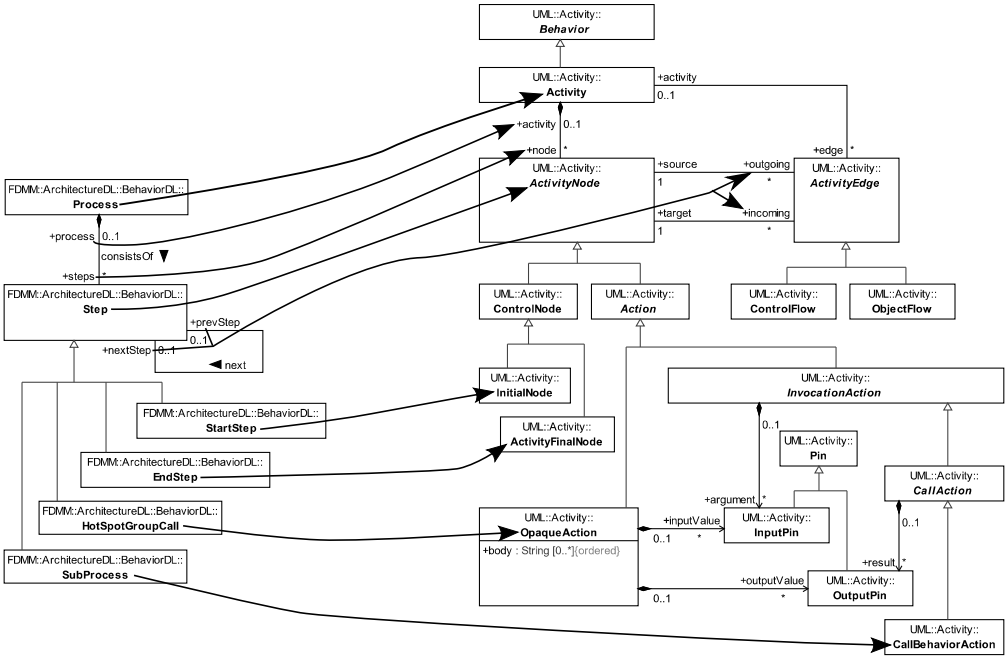
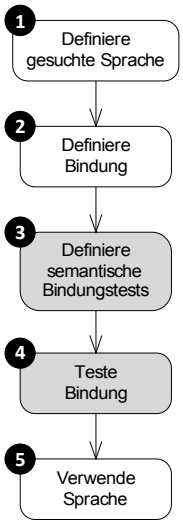


Abbildung 5.28: Mapping zwischen BehaviorDL und UML::Activity

5.4 Semantischer Bindungstest auf DMM-Basis



Der semantische Bindungstest umfasst die Schritte 3 und 4 im anforderungsbasierten Vorgehen in Abbildung 5.8. Der semantische Bindungstest testet die als Verhaltensmuster gestellten Anforderungen im Bezug auf das formal spezifizierte Ausführungsverhalten einer existierenden Sprache.

Konkret basiert der Test darauf, dass das Ausführungsverhalten der existierenden Sprache per »Dynamic Meta-Modeling« (DMM) [Engels2000] spezifiziert ist. Um die Details des semantischen Bindungstest nachvollziehen zu können, führen wir zunächst den DMM-Ansatz in den nachfolgenden Grundlagen zum Ausführungsverhalten in Abschnitt 5.4.1 ein. Im Anschluss daran widmen wir uns ab Abschnitt 5.4.2 der konkreten Umsetzung des Testverfahrens auf DMM-Basis.

5.4.1 Grundlagen zum Ausführungsverhalten

Im diesem Abschnitt führen wir als Grundlage für eine formale Spezifikation des Ausführungsverhaltens einer Sprache den »Dynamic Meta-Modeling« Ansatz ein, den wir voraussetzen, um das Ausführungsverhalten einer wiederverwendeten Sprache zu definieren. Insbesondere beziehen wir uns auf ein testgetriebenes Verfahren zur Erstellung von DMM-Spezifikationen.

Dynamic Meta-Modeling

»Dynamic Meta-Modeling« [Engels2000], kurz DMM, ist eine bestehende Technik zur Erstellung einer formalen Spezifikation der Ausführungssemantik einer ausführbaren visuellen Modellierungssprache. Die ursprüngliche Motivation für die Entwicklung von DMM war die Semantik der UML durch einen formalen Ansatz präzise zu beschreiben. Die bisherigen Versionen der UML-Spezifikation beschreiben die Semantik nur textuell, was Raum für Ungenauigkeiten und unterschiedliche Interpretationen lässt. Eine formale und damit präzise Spezifikation hingegen löst dieses Problem.

In Abbildung 5.29 zeigen wir eine Übersicht über den DMM-Ansatz. Der Ansatz setzt voraus, dass die abstrakte Syntax der visuellen Modellierungssprache als Meta-Modell definiert ist. Dieses Meta-Modell wird zu einem Laufzeit-Meta-Modell erweitert. Das Laufzeit-Meta-Modell besteht dabei aus dem ursprünglichen Meta-Modell erweitert um zusätzliche Elemente, die benötigt werden, um den Ausführungszustand zu spezifizieren.

Zusätzlich zum Laufzeit-Meta-Modell besteht eine DMM-Spezifikation aus einer Menge von Graph-Transformationsregeln, die Zustandsübergänge während der Ausführung einer Instanz der Sprache über dem Laufzeit-Meta-Modell definieren. In den Graph-Transformationsregeln wird so die Ausführungssemantik der Sprache formal erfasst, da diese Regeln eindeutig festlegen, in welchen Zuständen welche Folgezustände erreicht werden können.

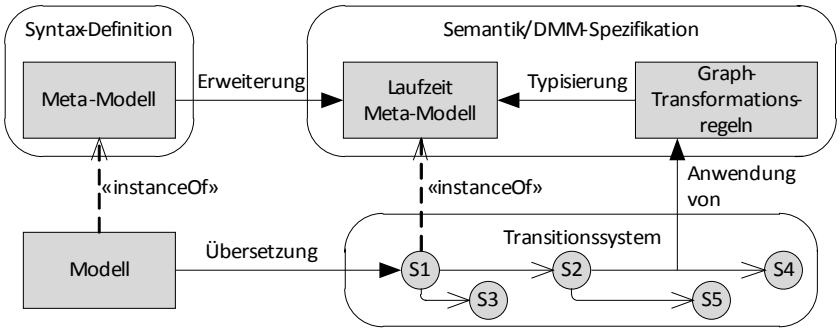


Abbildung 5.29: Übersicht über den DMM-Ansatz

Mit Hilfe der DMM-Spezifikation kann die Ausführungssemantik eines gegebenen Modells als Instanz des Meta-Modells der Modellierungssprache in Form eines Transitionssystems berechnet werden. Das Transitionssystem repräsentiert dabei ausgehend vom initialen Zustand des Modells alle möglichen Ausführungspfade. Hierzu wird das Modell auf Basis der Erweiterung von Meta-Modell zu Laufzeit-Meta-Modell in eine Instanz des Laufzeit-Meta-Modells übersetzt. Diese Übersetzung liefert gleichzeitig den initialen Zustand (S1) des Modells für die Berechnung aller möglichen Ausführungspfade auf Basis der Graph-Transformationsregeln. Dabei entspricht jeder Zustandsübergang im Transitionssystem der Anwendung einer Graph-Transformationsregel.

Wir wollen das Beispiel einer DMM-Spezifikation für UML-Aktivitätendiagramme aus [Hornkamp2009] heranziehen, um den DMM-Ansatz zu veranschaulichen. Im linken Teil der Abbildung 5.30 zeigen wir den relevanten

Ausschnitt aus dem Meta-Modell für UML-Aktivitätendiagramme. Dieses gegebene Meta-Modell wird erweitert, um das benötigte Laufzeit-Meta-Modell zu erhalten. Die benötigten Laufzeitelemente modellieren wir hierbei in einem eigenen Meta-Modell dargestellt im rechten Teil der Abbildung 5.30. Mit den Laufzeitelementen erweitern wir so das gegebene Meta-Modell um ein Token-Konzept. Mit Hilfe des Token-Konzeptes kann innerhalb des Laufzeit-Meta-Modell ausgedrückt werden, in welchem Zustand sich eine Laufzeitinstanz befindet.

Hierzu modellieren wir das Laufzeitelement Token, das einer ActivitiyNode zugeordnet wird. Die Klasse ActivityNode repräsentiert dabei eine Aktivität im Aktivitätendiagramm. Ist ein Token einem ActivityNode zugeordnet, bedeutet dies, dass diese Aktivität aktuell ausgeführt wird. Zusätzlich kann ein Token mit einer beliebigen Anzahl an Offer Objekten verlinkt sein. Eine Offer stellt ein Angebot für einen möglichen Tokenfluss über eine Kante im Aktivitätendiagramm dar, um die nächste Aktivität auszuführen. Die verbindenden Kanten zwischen Aktivitäten werden durch die Klasse ActivityEdge modelliert. Eine Offer für einen Tokenfluss kann entweder mit einer Aktivität (ActivityNode) oder mit einer Kante (ActivityEdge) verlinkt sein.

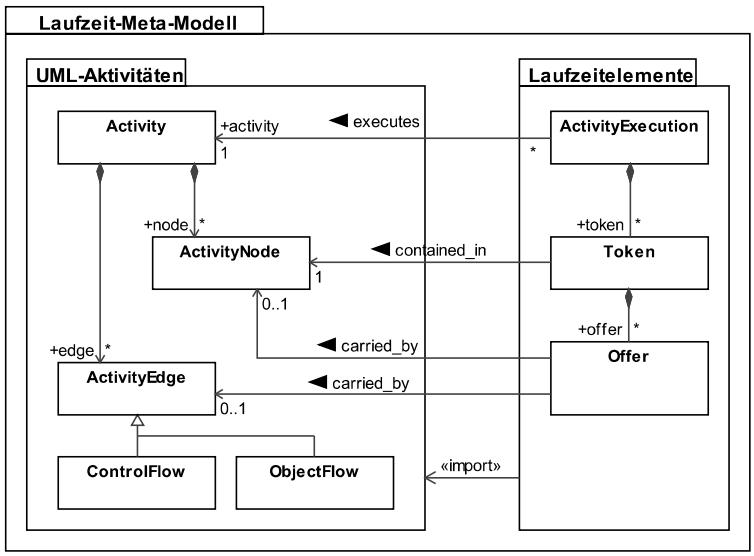


Abbildung 5.30: Laufzeit-Meta-Modell für UML-Aktivitätendiagramme

Das Laufzeit-Meta-Modell ermöglicht es den aktuellen Ausführungszustand eines Modells zu definieren. Zusätzlich benötigt man ein Regelwerk, das formal definiert, welche Zustandsübergänge gemäß der Ausführungssemantik der Modellierungssprache zulässig sind. Dieses Regelwerk wird im DMM-Ansatz mit Hilfe von Graph-Transformationsregeln über dem Laufzeit-Meta-Modell definiert. Die Graph-Transformationsregeln legen fest, wie sich eine Instanz des Laufzeit-Meta-Modells gemäß der Ausführungssemantik bei einem Zustandsübergang verändert. Im Beispiel wird mittels der Graph-Transformationsregeln der Tokenfluss als Veränderung der Instanz des Laufzeit-Meta-Modells simuliert. Durch die formale Definition der Graph-Transformationsregeln

wird so die Ausführungssemantik basierend auf einem gültigen Tokenfluss eindeutig festgeschrieben.

Wir wollen das Beispiel einer DMM-Graph-Transformationsregel, kurz DMM-Regel, für den Tokenfluss über eine Kante im Aktivitätendiagramm zu einem Knoten genauer betrachten. Ein Tokenfluss kommt zustande, wenn ein Token einem ActivityNode zugeordnet wird und eine ActivityEdge ausgehend von dem ActivityNode über eine Offer in Relation zum Token steht. Die Offer wird angenommen, der Link zur ActivityEdge gelöst und die Offer mit dem verknüpften neuen ActivityNode verlinkt. Hat der neue ActivityNode die Offer für den Token, wird der Token dem neuen ActivityNode zugeordnet. Logisch ist der Token über die ActivityEdge von einem ActivityNode zum anderen geflossen, so dass nun die Aktivität des neuen ActivityNode ausgeführt wird.

Die DMM-Regel zugehörige moveOffer zeigen wir in Abbildung 5.31. Diese Regel drückt aus, wie eine angebotene Offer einer ActivityEdge auf einen ActivityNode als Ziel des Tokenflusses übergeben wird.

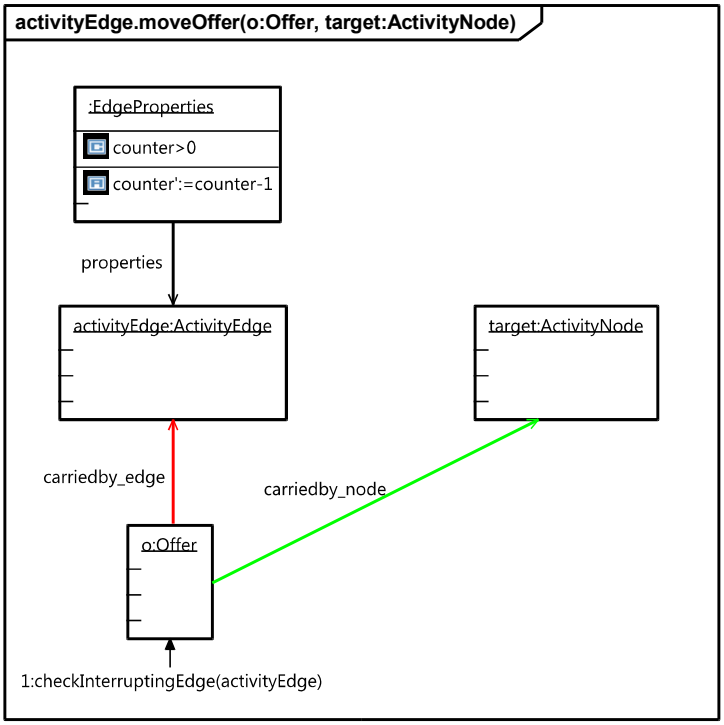


Abbildung 5.31: Beispiel einer DMM Graph-Transformationsregel

Jede DMM-Regel definiert einen Kontextknoten (auch Kontext-Objekt genannt), der festlegt in welchem Kontext eine Regel ausgeführt wird. Er definiert den Ankerpunkt zu dem alle weiteren Objekte der Regel in Beziehung stehen. Der Kontextknoten der Regel in Abbildung 5.31 ist als Präambel für den Namen der Regel oben links festgelegt auf activityEdge:ActivityEdge.

Der Name der Regel ist moveOffer. Zusätzlich verfügt diese Regel über zwei Parameter. Der erste Parameter o:Offer definiert ein Objekt vom Type Offer, um festzulegen für welches Offer-Objekt diese Regel angewendet wird. Der zweite Parameter definiert das Ziel für den Tokenfluss durch Angabe eines

ActivityNodes-Objektes, hier mit dem Namen target. Die DMM-Regel soll nun formal spezifizieren, dass das Objekt o:Offer mit der Kante activityEdge:ActivityEdge verlinkt ist. Um den Tokenfluss zu etablieren, muss das Objekt o:Offer nun mit der target:ActivityNode verlinkt und der Link zur activitiyEdge:ActivityEdge gelöst werden.

DMM-Regeln bieten eine Notation für Links, die bei Ausführung der Regel erwartet und dann gelöscht werden (roter Link carriedby_edge) und eine Notation für Links, die durch bei Ausführung der Regel neu erzeugt werden (grüner Link carriedby_node). Mit Hilfe dieser beiden Notationselemente können wir in der DMM-Regel in Abbildung 5.31 festlegen, dass der Link carriedby_edge erwartet und gelöscht wird und der Link carriedby_node zum Objekt target:ActivityNode neu erzeugt wird. Auf diese Weise wandert die o:Offer von der activityEdge:ActivityEdge zum target:ActivityNode.

Um das Beispiel abzuschließen müsste anschließend eine weitere Regel aufbauend auf der gezeigten moveOffer Regel angewendet werden. Diese Regel setzt den Token auf den activityNode:ActivityNode. Im Ergebnis ist der Tokenfluss über die Kante activityEdge:ActivityEdge formal spezifiziert.

Für eine vollständige Darstellung der Modellierungsmöglichkeiten für DMM-Regeln sei auf [Hausmann2005] verwiesen. Für diese Arbeit ist das vermittelte Grundverständnis ausreichend. Daher gehen wir im Folgenden nicht auf weitere Details ein, sondern widmen uns einem testgetriebenen Verfahren zur Erstellung von Semantikspezifikationen, insbesondere von DMM-Spezifikationen.

Testgetriebene Semantikspezifikation

Mit DMM haben wir einen Ansatz zur Spezifikation der Ausführungssemantik von visuellen Modellierungssprachen vorgestellt. In diesem Abschnitt wollen wir uns mit einem Verfahren zur Erstellung solcher Spezifikationen widmen. In [Soltenborn2009] stellen die Autoren Soltenborn und Engels einen testgetriebenen Ansatz für die Erstellung von Semantikspezifikationen vor. Dabei gehen sie insbesondere auf DMM-Spezifikationen ein. Wir führen den Ansatz als Grundlage für die Umsetzung des Vorgehens einer anforderungsbasierten Wiederverwendung von Sprachen ein.

Die Grundidee eines testgetriebenen Ansatzes ist, dass für ein zu erstellendes Artefakt mit einem Test zunächst dessen Eigenschaften überprüfbar festgelegt werden. Während der eigentlichen Entwicklung des Artefaktes, wird das Artefakt mit Hilfe des Tests immer wieder auf seine Eigenschaften geprüft. Besteht das erstellte Artefakt den Test, wird auf analoge Weise das nächste Artefakt entwickelt.

Im hier betrachteten testgetriebenen Ansatz zur Erstellung von Semantikspezifikationen für visuelle Modellierungssprachen aus [Soltenborn2009] ist die Semantikspezifikation das zu erstellende Artefakt. Mit Hilfe einer Reihe von Tests kann während der Erstellung der Semantikspezifikation überprüft werden, ob diese die geforderten Eigenschaften erfüllt. Ein Test definiert für ein Beispielmodell in der Modellierungssprache das erwartete Verhalten bei Aus-

führung des Beispielmodells auf Basis der zu erstellenden Semantikspezifikation.

Das erwartete Verhalten wird dabei in Form von erwarteten Ausführungspfaden bestehend aus einzelnen Ausführungsereignissen (engl. traces of execution events) spezifiziert. Ein Ausführungsereignis (engl. execution event) repräsentiert dabei die Ausführung eines Sprachelements des Meta-Modells der Modellierungssprache. Für die Erstellung der Tests definiert man Ausführungsereignisse nur für die zentralen Sprachelemente, da nur deren Ausführung überhaupt von Interesse ist. Typischerweise enthält das Meta-Modell neben den zentralen Sprachelementen eine ganze Reihe weiterer Elemente als Hilfskonstrukte, die für die Betrachtung eines Ausführungspfades jedoch nicht von Interesse sind.

Den zugrunde liegenden Prozess zur Erstellung von Testmodellen und des erwarteten Verhaltens aus [Soltenborn2009] zeigen wir in Abbildung 5.32. Zu Beginn wird ein Testmodell erstellt und im Anschluss dessen Semantik diskutiert. Man überlegt sich, was bei Ausführung des Testmodells geschehen müsste. Anschließend identifiziert man die Ausführungsereignisse, die für die Ausführung von Interesse sind. Auf Grundlage der betrachteten Ausführungsereignisse werden im letzten Schritt alle Ausführungspfade als Sequenzen von Ausführungsereignissen beschrieben, die man bei Ausführung des Testmodells erwarten würde.

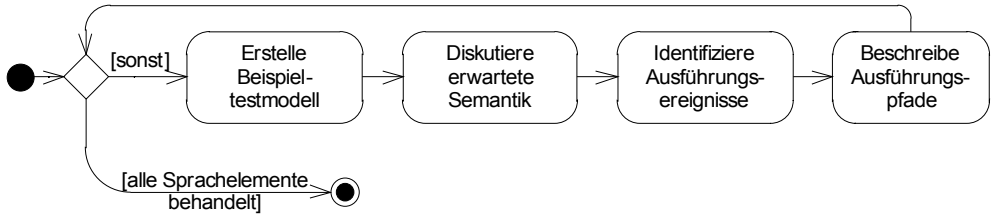


Abbildung 5.32: Prozess zur Erstellung von Testmodellen nach [Soltenborn2009]

Hat man eine Menge geeigneter Testmodelle und deren Ausführungspfade spezifiziert, kann man zur Erstellung der eigentlichen Semantikspezifikation schreiten. Geeignete Testmodelle sollten gemäß [Soltenborn2009]

- sich auf wenige Sprachelemente und ihre Semantik konzentrieren,
- gemeinsam alle betrachteten Sprachelemente der Sprache abdecken und
- jeweils zu einem endlichen Transitionssystem führen.

Betrachten wir ein Beispiel für ein Testmodell für UML-Aktivitätendiagramme in Abbildung 5.33. Das Testmodell enthält die drei Aktivitäten (Activity) A, B und C sowie einen Entscheidungs- (DecisionNode) und Zusammenführungsknoten (MergeNode). Die Ausführung dieses Aktivitätendiagramms würde entweder zur Ausführung der Aktivitäten »erst A dann B« oder »erst A dann C« führen. Das Ausführungsereignis von Interesse ist in diesem Fall die Ausführung einer Aktivität. Nicht von Interesse ist in diesem Beispiel etwa die Ausführung des Entscheidungsknotens (DecisionNode). Daher definieren wir

lediglich das Ausführungsereignis ActionExecutes, dem wir den Namen der ausgeführten Aktivität als Parameter mitgeben. Das Ereignis ActionExecutes("A") besagt, dass die Aktivität mit Namen »A« ausgeführt wurde.

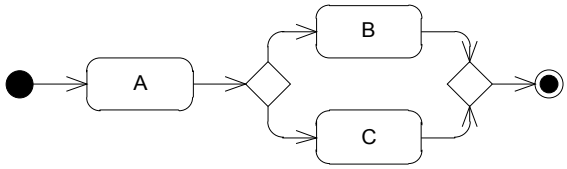


Abbildung 5.33: Beispiel eines Testmodells für Aktivitätendiagramme nach [Soltenborn2009]

Für das Testmodell in Abbildung 5.33 können wir die erwarteten Ausführungspfade wie folgt auflisten:

- ActionExecutes("A") ActionExecutes("B")
- ActionExecutes("A") ActionExecutes("C")

Bei Ausführung des Testmodells erwarten wir, dass beide Ausführungspfade im Transitionssystems zu finden sind, da dieses alle möglichen Ausführungspfade enthält. Nachdem Testmodelle und erwartete Ausführungspfade definiert wurden, kann mit der Erstellung der eigentlichen Semantikspezifikation der Sprache begonnen werden. Die Reihenfolge ist nicht als strikt anzusehen, denn üblicherweise entwickelt man neue Tests und die Semantikspezifikation in Zyklen, um so sukzessiv die gesamte Semantikspezifikation zu erstellen. Den hierfür vorgeschlagenen Prozess aus [Soltenborn2009] zeigen wir in Abbildung 5.34.

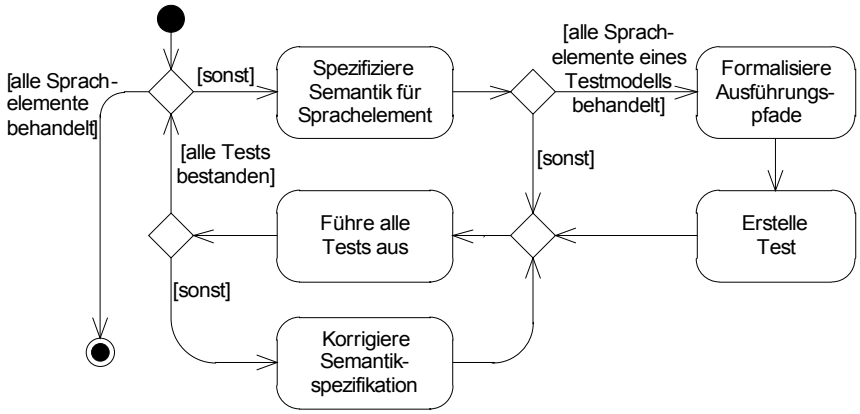


Abbildung 5.34: Testgetriebene Erstellung der Semantikspezifikation nach [Soltenborn2009]

Zur Überprüfung der erwarteten Ausführungspfade bei Ausführung der Testmodelle müssen diese formalisiert werden. Diese Formalisierung erfolgt durch Übersetzung der Ausführungspfade in temporal-logische Ausdrücke, die dann per Model-Checking direkt auf dem berechneten Transitionssystem als Test

überprüft werden können. Für das in in Abbildung 5.33 gezeigt Beispiel ergibt sich etwa das in Abbildung 5.35 gezeigte Transitionssystem.

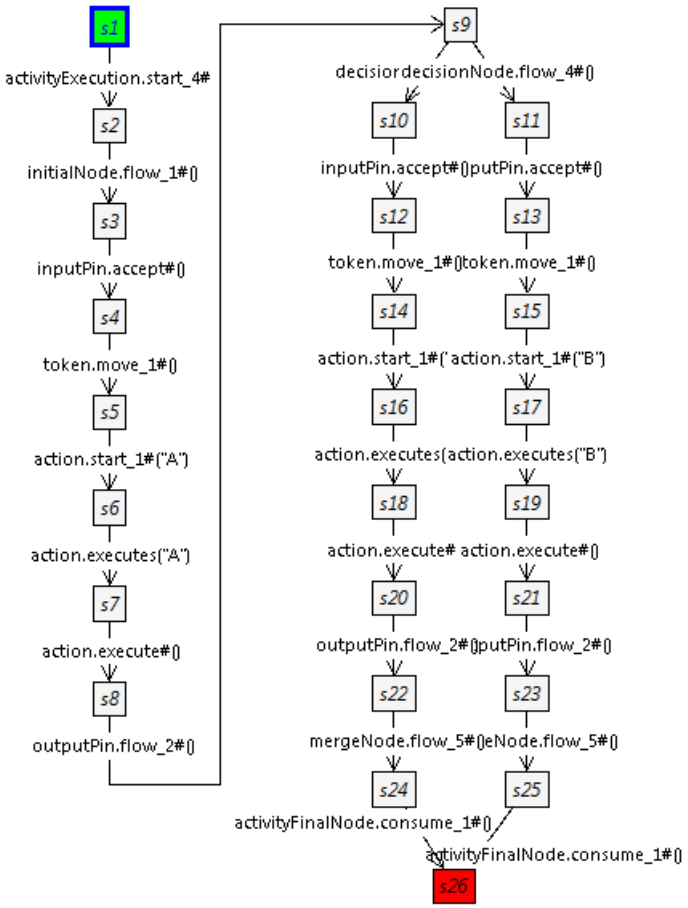


Abbildung 5.35: Transitionssystem

Die temporal-logischen Ausdrücke, werden mit Hilfe der »Computation Tree Logic« (CTL) [Clarke1986] spezifiziert. Betrachten wir als Beispiel den folgenden Ausführungspfad:

ActionExecutes("A") ActionExecutes("B")

Dieser Ausführungspfad besagt, dass das Transitionssystem einen Pfad enthalten muss, bei dem an irgendeinem Zeitpunkt die Aktivität "A" und später Aktivität "B" ausgeführt wird. In CTL können wir dies mit Hilfe des $\boldsymbol{EF}$ Operators spezifizieren. Der Ausdruck

$$s_0 \models \boldsymbol{EF}(f)$$

bedeutet, dass in einer Struktur M (dem Transitionssystem) ein Pfad existieren soll, so dass ab Zustand s_0 die CTL-Formel $\boldsymbol{EF}(f)$ erfüllt ist [Clarke1986]. $\boldsymbol{EF}$ ist dabei die abkürzende Notation für $\boldsymbol{E}[true\ \boldsymbol{U}\ f]$, die besagt, dass ein Pfad existiert ($\boldsymbol{E}$ ist der Existenz-Operator), auf dem so lange wahr (

true) gilt, bis die CTL-Formel f erfüllt ist ($\boldsymbol{U}$ ist der Solange-Bis-Operator) [Clarke1986]. Mit anderen Worten, es soll ein Pfad existieren auf dem irgendwann f erfüllt ist.

In einer DMM-Spezifikation sind die Transitionen im Transitionssystem Ausführungen von DMM-Regeln. Einer zu prüfenden Eigenschaft entspricht somit die Ausführung einer DMM-Regel und wir können mit Hilfe von CTL-Formeln überprüfen, ob bestimmte DMM-Regeln ausgeführt wurden.

Die Ausführungspfade beschreiben Folgen von Ausführungsereignissen. Wir können jedes Ausführungsereignis in Beziehung zur seiner korrespondierenden DMM-Regel setzen. Jedes Ausführungsereignis entspricht dabei genau einer DMM-Regel. Das Ausführungsereignis ActionExecutes("A") kann so direkt in die korrespondierende DMM-Regel action.executes("A") übersetzt werden. Den oben angegebenen Ausführungspfad können wir so in folgende CTL-Formel übersetzen.

$$p_1 := \boldsymbol{EF}\left(action.executes('A') \wedge \boldsymbol{EF}\left(action.executes('B')\right)\right)$$

Mit p_1 können wir nun theoretisch die gewünschte Eigenschaft auf dem Transitionssystem per Model-Checking überprüfen. Es stellt sich jedoch ein weiteres technisches Problem, da die bisherige CTL-Formel nicht spezifiziert, was vor der Aktivität "A", zwischen "A" und "B" und nach "B" geschieht. Wir müssen zusätzlich ausdrücken, dass die zusätzlichen Zustände, die das Transitionssystem vor, nach und zwischen den Zuständen von Interesse enthält, nicht zu beachten sind. Hierzu benötigen wir zwei weitere CTL-Operatoren: den Nachfolge-Operator $\boldsymbol{X}$ und den All-Quantifizierer $\boldsymbol{A}$.

Zur vereinfachten Darstellung wird in [Soltenborn2009] nun folgende abkürzende Schreibweise eingeführt. Die Autoren bezeichnen $action.executes('A')$ als a_A (a_B und a_C entsprechend). Dann definieren sie die Menge $R=\{a_A, a_B, a_C\}$ als die Menge aller DMM-Regeln für die Ausführungsereignisse, die betrachten werden. Schließlich wird noch das Prädikat $\hat{R}$ als $\wedge_{r\in R}(\neg r)$ definiert. Mit Hilfe dieser Hilfskonstrukte kann nun die eigentliche CTL-Formel kanonisch aufgebaut werden:

$$P_1 := \boldsymbol{E}\left(\hat{R}\,\boldsymbol{U}\,X_A\right)$$

Die intuitive Aussage von P_1 ist, dass man das erste Vorkommen von a_A bestimmen möchte, wobei verhindert wird, dass a_B oder a_C zuvor auftreten. So definieren wir im nächsten Schritt X_A wie folgt:

$$X_A := a_A \wedge \boldsymbol{E}\,\boldsymbol{X}\,\boldsymbol{E}\left(\hat{R}\,\boldsymbol{U}\,X_B\right)$$

X_A besagt, dass ab dem Auftreten von a_A nachfolgend nach einem Auftritt von a_B gesucht wird, wobei zwischen den Zuständen, die a_A bzw. a_B erfüllen, kein weiteres Vorkommen von a_A erlaubt ist. Würde der gesuchte Pfad aus mehr DMM-Regeln als aus a_A und a_B bestehen, könnte man die Formel an dieser Stelle für weitere DMM-Regeln in einer Abfolge beliebig erweitern

und so automatisch aufbauen. Abschließend müssen wir nun noch X_B definieren, wodurch wir verhindern, dass nach einem Auftreten von a_B die DMM-Regeln a_A oder a_B nochmals auftreten können. X_B wird somit wie folgt definiert:

$$X_B := a_B \wedge \boldsymbol{A}\boldsymbol{X}\,\boldsymbol{AG}(\hat{R})$$

Der $\boldsymbol{AG}$ Operator ist dabei der globale Operator, der als $s_0 \models \boldsymbol{AG}(f) \equiv \neg\,\boldsymbol{EF}(\neg f)$ definiert ist. Er bedeutet, dass f in jedem Zustand auf jedem Pfad von s_0 erfüllt ist [Clarke1986].

Mit X_B ist die Definition von P_1 als Formalisierung des Pfades ActionExecutes("A") ActionExecutes("B") abgeschlossen. Für den Pfad »ActionExecutes("A") ActionExecutes("C")« können wir analog verfahren und diesen Pfad als P_2 formalisieren.

Nach der Formalisierung aller Ausführungspfade können wir den eigentlichen Test erstellen. Der Test formalisiert, dass alle geforderten Ausführungspfade im Transitionssystem vorhanden sein müssen. Hierzu kann mit einem Model-Checker jeder formalisierte Pfad P_x geprüft werden.

Zusätzlich müssen wir noch prüfen, dass es keinen Pfad im Transitionssystem gibt, auf dem entweder P_1 oder P_2 gilt. Würde ein Pfad existieren, auf dem weder P_1 noch P_2 eingehalten wird, hätte man ein unerwartetes Verhalten im Transitionssystem entdeckt. Dies kann mit folgender CTL-Formel aufgedeckt werden [Soltenborn2009]:

$$\boldsymbol{AF}(P_1 \vee P_2)$$

$\boldsymbol{AF}(f)$ ist eine abkürzende Schreibweise für $s_0 \models \boldsymbol{AF}(f) \equiv \boldsymbol{A}[\mathit{true}\,\boldsymbol{U}\,f]$ und besagt, dass auf allen Pfaden von s_0 die CTL-Formel f erfüllt sein muss. Würde ein Model-Checker einen Pfad finden, auf dem weder P_1 noch P_2 gilt, würde er diesen Pfad als Gegenbeispiel ausgeben.

Mit der testgetriebenen Semantikspezifikation haben wir nach [Soltenborn2009] ein Verfahren vorgestellt, um das Ausführungsverhalten visueller Modellierungssprachen zu entwickeln. Insbesondere kann es eingesetzt werden, um eine DMM-Spezifikation zu erstellen.

Mit dem testgetriebenen DMM-Ansatz haben wir alle benötigten Grundlagen eingeführt, die wir für das Verfahren des semantischen Bindungstest benötigen. Wir stellen die Details dieses Verfahrens im nachfolgenden Abschnitt 5.4.2 vor.

5.4.2 Verfahren zum semantischen Bindungstest

Im semantischen Bindungstest spezifizieren wir ausgehend von den geforderten Verhaltensmustern die auszuführenden Tests. Das Verfahren hierzu besteht aus einer Reihe von Übersetzungsschritten, die es uns erlauben, die Verhaltens-

muster auf konkrete Beispiele zu übertragen und dann die konkreten Beispiele gegenüber der DMM-Spezifikation auf ihr erwartetes Verhalten hin zu testen.

Wir stellen das Verfahren in einer schematischen Übersicht in Abbildung 5.36 dar. Zur besseren Einordnung haben wir Zahlen als Verweise auf die korrespondierenden Schritte des Vorgehens zur anforderungsbasierten Wiederverwendung von Sprachen aus Abbildung 5.8 eingefügt.

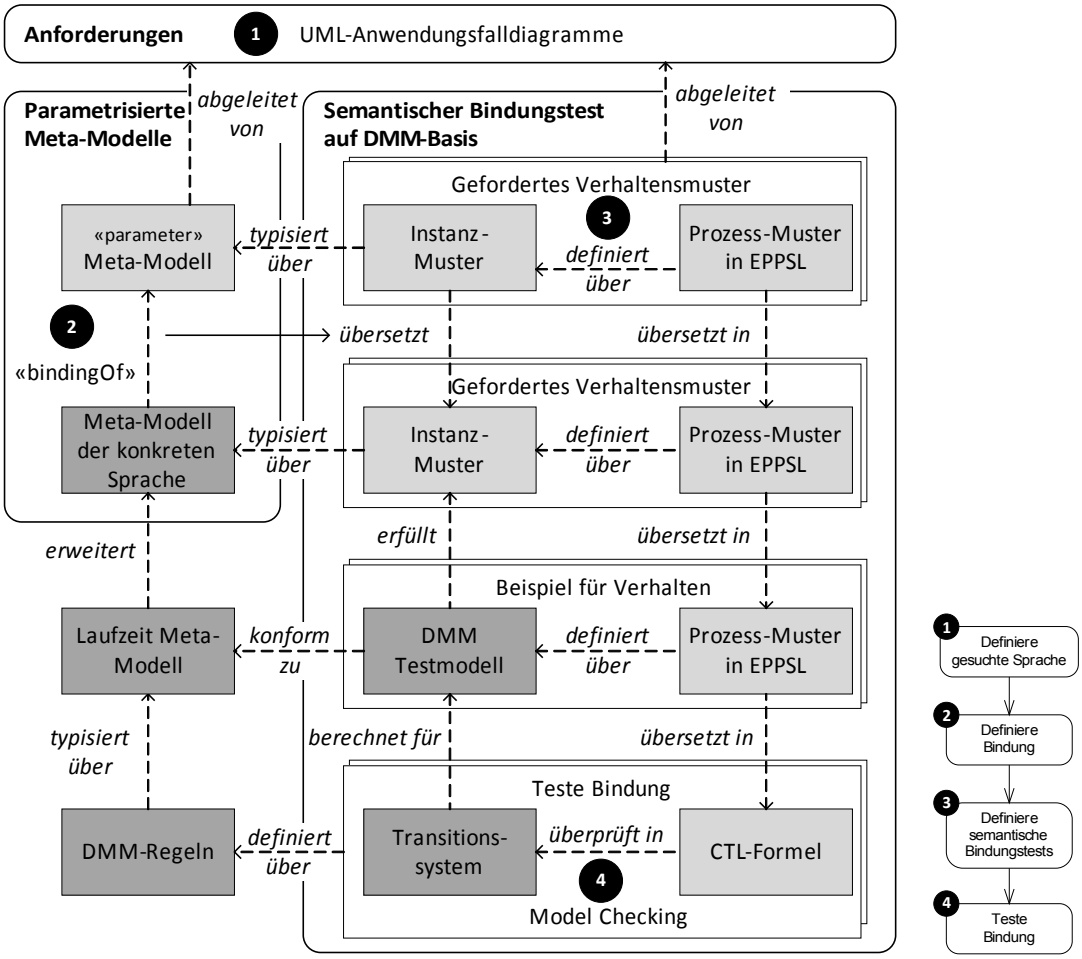


Abbildung 5.36: Umsetzung des Vorgehens der anforderungsbasierten Wiederverwendung von Sprachen

Ausgangspunkt des Verfahrens ist eine parametrisierte Sprache, bei der wir für einzelne Aspekte der Sprache existierende Sprachen wiederverwenden möchten. Die zu erfüllenden Anforderungen wurden mit Hilfe von Anwendungsfällen für die Sprache des Aspektes erfasst. Auf Basis der Anwendungsfälle wurde die geforderte Struktur und die Konzepte als Meta-Modell abgeleitet und eine Bindung mittels der Modellierungstechnik der parametrisierten Meta-Modelle definiert. Das geforderte Verhalten einzelner Sprachelemente ist durch Verhaltensmuster beschrieben.

Der semantische Bindungstest besteht aus mehreren Schritten. Für die Umsetzung der einzelnen Schritte müssen wir einige Voraussetzung einführen, ohne die weitere Analysen nicht möglich wären. Wir fordern, dass das Verhal-

ten der konkreten Sprache mit Hilfe des testgetriebenen DMM-Ansatzes spezifiziert ist. Dies bedeutet, dass eine gegebene Sprache durch sein Meta-Modell, dem Laufzeit-Meta-Modell, DMM-Regeln und durch eine Reihe von Beispielmodellen spezifiziert ist. Diese Voraussetzungen sind valide, da nur dann eine konkrete Sprache überprüft werden kann, wenn deren Ausführungsverhalten formal spezifiziert wurde. Die Forderung nach einem testgetriebenen Verfahren auf Basis von Beispielmodellen ist ebenfalls valide, da Beispielmodelle zum Zeitpunkt der Spezifikation des Verhaltens in jedem Fall vorhanden sein sollten.

Im ersten Schritt werden auf Basis der Bindung der konkreten Sprache an das Meta-Modell des formalen Parameters die Instanz- und Prozessmuster, die das geforderte Verhalten der Sprachelemente definieren, in Muster über dem Meta-Modell der konkreten Sprache übersetzt. Dieser Übersetzungsschritt ist entscheidend für die Prüfung der Bindung. Falls in der Bindung zwischen formalem Parameter und konkreter Sprache ein Fehler gemacht wurde, würde sich dieser Fehler auch auf die Übersetzung der Muster auswirken. An dieser Stelle sei betont, dass die Bindung der Meta-Modelle direkten Einfluss auf die Übersetzung des Verhaltens hat. Der Test deckt etwaige Fehler auf, so dass auf semantische Fehler in der Bindung geschlossen werden kann.

Im nachfolgenden Schritt machen wir uns zu nutze, dass das Verhalten der konkreten Sprache testgetrieben durch DMM spezifiziert wurde. Somit verfügen wir über Beispielmodelle in der konkreten Sprache, die ursprünglich zur testgetriebenen Spezifikation der konkrete Sprache benötigt wurden. Auf Basis der Instanzmuster wählen wir aus diesen Beispielmodellen diejenigen aus, die die Instanzmuster erfüllen. Für jedes Beispielmodell können wir außerdem das dazugehörige Prozessmuster in ein Prozessmuster über dem konkreten Beispielmodell übersetzen. So haben wir die allgemeinen Muster auf konkrete Beispiele übertragen.

Im letzten Schritt berechnen wir mit Hilfe der DMM-Regeln für jedes Beispielmodell dessen Gesamtverhalten in Form eines Transitionssystems. Gleichzeitig übersetzen wir das konkrete Prozessmuster des Beispiels in eine CTL-Formel, die einen erwarteten Ausführungspfad im Transitionssystem beschreibt. Sowohl das Transitionssystem als auch die CTL-Formel sind über DMM-Regeln definiert. Mit Hilfe eines Model-Checkers können wir die CTL-Formeln auf dem Transitionssystem überprüfen und erhalten entweder eine Bestätigung, dass der Ausführungspfad korrekt gefunden werden konnte oder eine Ablehnung, dass der geforderte Ausführungspfad nicht im Gesamtverhalten vorhanden ist. Im Falle einer Ablehnung haben wir einen Fall identifiziert, in dem entweder die Bindung fehlerhaft war oder das Verhalten der konkreten Sprache nicht zum geforderten Verhalten passt. Werden alle Tests bestanden, können wir die Bindung als semantisch korrekt interpretieren.

Wir wollen im Folgenden die Details des Verfahrens gemäß Abbildung 5.36 erläutern. Hierzu unterteilen wir das Vorgehen in vier Arbeitsschritte zu sehen in Abbildung 5.37 und auf die wir nachfolgenden eingehen werden.

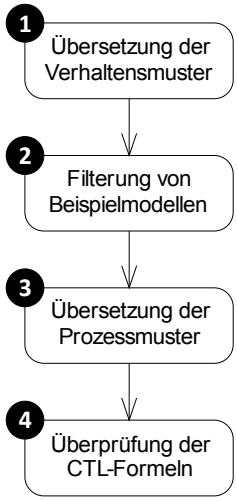


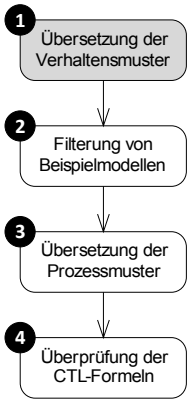
Abbildung 5.37: Vorgehen beim semantischen Bindungstest

Die vier Arbeitsschritte bestehen aus:

- *Übersetzung der Verhaltensmuster* über dem geforderten Meta-Modell in Ausführungsmuster über dem Meta-Modell der konkreten Sprache.
- *Filterung von Beispielmodellen* anhand der Instanzmuster und Übersetzung der entsprechenden Prozessmuster in Prozessmuster über den Beispielmodellen.
- *Übersetzung der Prozessmuster* in CTL-Formeln über DMM-Regeln.
- *Überprüfung der CTL-Formeln* auf dem Transitionssystem, das das Gesamtverhalten eines Beispielmodells definiert, per Model-Checking.

In den nachfolgenden Abschnitten werde wir die vier Schritte im Einzelnen erläutern.

Übersetzung der Verhaltensmuster



In diesem Schritt werden die Verhaltensmuster, die über dem geforderten Meta-Modell definiert sind, in Verhaltensmuster über dem Meta-Modell der konkreten Sprache übersetzt. Gegeben sind die geforderten Verhaltensmuster, wobei die Instanzmuster über dem Meta-Modell des Parameters typisiert sind. Des weiteren sind die Prozessmuster über dem Instanzmuster definiert. Diese Verhaltensmuster müssen in Verhaltensmuster spezifiziert über dem Meta-Modell der konkreten Sprache übersetzt werden. Diese Übersetzung wird auf Basis der Bindung zwischen dem Meta-Modell des Parameters und dem Meta-Modell der konkreten Sprache durchgeführt.

In Abschnitt 5.3.2 haben wir für parametrisierte Meta-Modelle die Eigenschaften und eine Notation für die Bindung definiert. Wir wollen am Beispiel einer Bindung zwischen der BehaviorDL und UML::Activity, die

wir im Beispiel ab Seite 146 im Abschnitt 5.3.2 gezeigt haben, veranschaulichen, wie eine solche Übersetzung durchgeführt wird. Hierzu betrachten wir erneut das Verhaltensmuster aus Abbildung 5.16, das wir unter Berücksichtigung der Bindung aus Listing 5.5 in ein Verhaltensmuster über dem UML::Activity Meta-Modell aus Abbildung 5.27 übersetzen. In Abbildung 5.38 zeigen wir im oberen Teil das ursprüngliche Verhaltensmuster und im unteren Teil das übersetzte Muster über dem UML::Activity Meta-Modell.

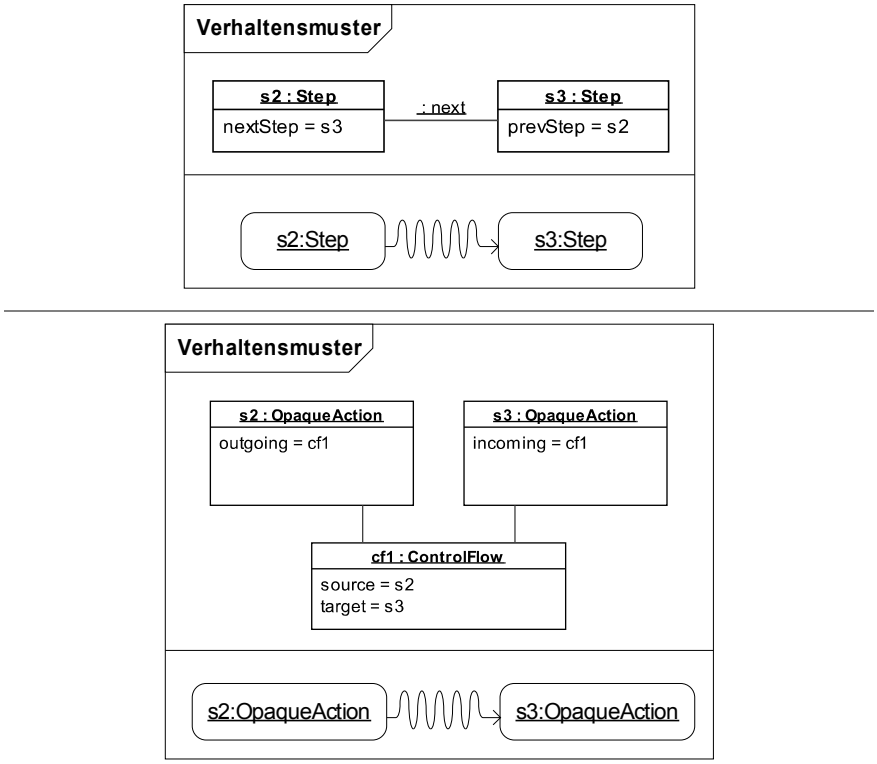
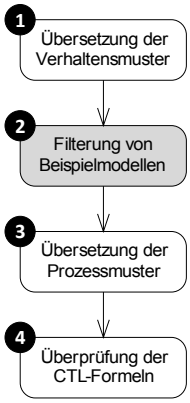


Abbildung 5.38: Übersetzung eines Verhaltensmusters

Laut Bindung in Listing 5.5 wird ein Step auf eine OpaqueAction und die next Beziehung zwischen zwei Steps auf einen ControlFlow abgebildet, dessen source und target auf die entsprechenden OpaqueActions verweisen. Damit können wir die Objektstruktur mit Hilfe der Bindung übersetzen. Das Prozessmuster können wir übersetzen, indem wir die Zustände nun gemäß der neuen Struktur überführen. Im Ergebnis erhalten wir auf der rechten Seite in Abbildung 5.38 ein Verhaltensmuster, das über über dem UML::Activity Meta-Modell definiert ist. Analog verfahren wir mit allen spezifizierten Verhaltensmustern.

Filterung von Beispielmodellen



Auf Basis der Verhaltensmuster, die nun über dem Meta-Modell der konkreten Sprache definiert sind, können wir nun Beispielmodelle aus der Menge der verfügbaren Beispielmodelle herausfiltern, die bestimmten Instanzmustern entsprechen. Zur Erinnerung: Wir haben vorausgesetzt, dass die konkrete Sprache testgetrieben entwickelt wurde, so dass wir von der Existenz einer Menge von Beispielmodellen ausgehen können.

Wir wollen diesen Schritt wieder an einem Beispiel verdeutlichen. Angenommen es läge das Beispielmodell aus Abbildung 5.39 vor. Im linken Teil von Abbildung 5.39 zeigen wir ein Aktivitätendiagramm in konkreter Syntax und im rechten Teil einen Ausschnitt aus der dazugehörigen abstrakten Syntax gemäß des UML::Activity Meta-Modells aus Abbildung 5.27. Auf Basis der abstrakten Syntax überprüfen wir nun welche Instanzmuster aus den gegebenen Verhaltensmustern auf dieses Beispielmodell passt.

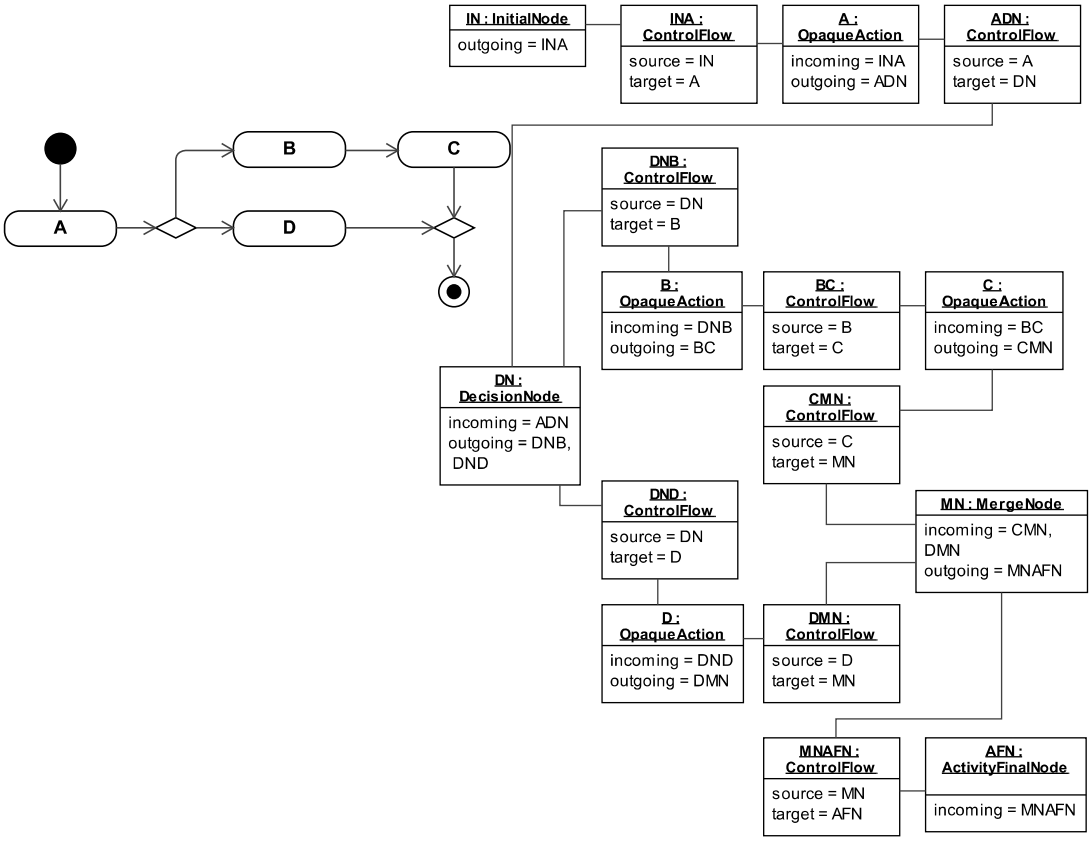


Abbildung 5.39: Beispiel eines UML-Aktivitätendiagramms in konkreter und abstrakter Syntax

Ein konkretes Beispielmodell erfüllt ein Instanzmuster, wenn dieselbe Konfiguration aus Objekten und Links im Beispielmodell gefunden werden kann. Hierbei erfolgt die Prüfung für sich entsprechende Objekte auf Basis der Objekttypen. Außerdem wird nach einer minimalen Überdeckung gesucht. Instanzmuster definieren, welche Konfigurationen aus Objekten und Links mindestens vorhanden sein müssen. Verfügen Objekte im konkreten Beispiel über weitere Links zu anderen Objekten, werden diese nicht beachtet und haben keine Auswirkung.

Im Beispielmodell in Abbildung 5.39 erkennen wir zum Beispiel die Objektkonfiguration aus B:OpaqueAction, BC:ControlFlow und C:OpaqueAction. Diese Objektkonfiguration passt auf das Instanzmuster in Abbildung 5.38 bestehend aus s2:OpaqueAction, cf1:ControlFlow und s3:OpaqueAction. Das Instanzmuster ist in diesem Fall erfüllt, obwohl beispielsweise DNB:ControlFlow verlinkt ist mit B:OpaqueAction, was nicht Teil des Instanzmusters ist. Dieser Zusammenhang ist für den semantischen Bindungstest nicht von Belang und kann somit ignoriert werden.

Wir haben mit dem Beispielmodell in Abbildung 5.39 ein Beispiel gefunden, dass das Instanzmuster aus Abbildung 5.38 erfüllt. Wir können somit das Verhaltensmuster auf dieses Beispiel anwenden. Hierzu überführen wir das Prozessmuster, das über dem Meta-Modell definiert ist, auf das konkrete Beispiel. Diese Überführung ist trivial, da wir lediglich die sich entsprechenden Objekte in das Prozessmuster einsetzen. Im Ergebnis dieses Beispiels erhalten wir das Prozessmuster im linken Teil von Abbildung 5.40. Dieses Prozessmuster beschreibt nun einen konkret geforderten Ablauf im Beispielmodell, das wir nochmal im rechten Teil von Abbildung 5.40 dargestellt haben.

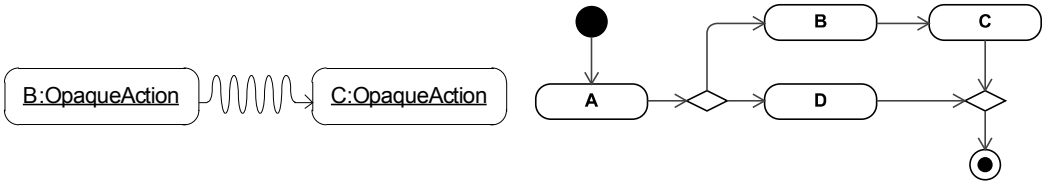
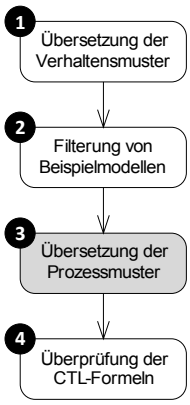


Abbildung 5.40: Prozessmuster über einem konkreten Beispielmodell eines UML-Aktivitätendiagramms

Anhand des Beispiels haben wir gezeigt, wie ein Verhaltensmuster auf konkrete Beispielmodelle angewendet werden kann. Im Ergebnis erhalten wir ausgewählte Beispielmodelle mit konkreten geforderten Prozessmustern. Die Prozessmuster entsprechen dabei den gestellten Anforderungen an das Verhalten der wiederverwendeten Sprache. Um dieses geforderte Verhalten nun überprüfen zu können, müssen wir die Prozessmuster im letzten Schritt des semantischen Bindungstest in eine Form übersetzen, die eine Überprüfung durch einen Model-Checker erlaubt.

Übersetzung der Prozessmuster



Das Verhalten ist nach dem DMM-Ansatz über DMM-Regeln spezifiziert. Diese Graph-Transformationsregeln legen formal die möglichen Ausführungspfade fest. Das Gesamtverhalten eines Beispielmodells lässt sich mit dem DMM-Ansatz durch Ausführung der DMM-Regeln mit einem Model-Checker berechnen. Dabei entsteht ein Transitionssystem, dessen Zustände unterschiedliche Objektkonfigurationen des Beispielmodells sind. Jede Objektkonfiguration repräsentiert einen Ausführungszustand. Die Übergänge zwischen den Zuständen sind durch die DMM-Regeln festgelegt. Wir halten fest, dass das Transitionssystem über DMM-Regeln definiert ist.

Für den semantischen Bindungstest wollen wir ein geforderte Verhalten in diesem Transitionssystem, das dem Gesamtverhalten entspricht, überprüfen. Da das Transitionssystem über DMM-Regeln definiert ist, muss auch das geforderte Verhalten über DMM-Regeln definiert sein. Dies bedeutet, dass wir die Prozessmuster aus dem vorigen Schritt nun in Prozessmuster über DMM-Regeln überführen müssen.

In einer DMM-Spezifikation gibt es eine implizite eins-zu-eins Beziehung zwischen Konzepten des Laufzeit-Meta-Modells einer Sprache und dessen Ausführung formalisiert durch eine DMM-Regel. So gibt es zum Beispiel in der DMM-Spezifikation für UML-Aktivitätendiagramme eine Regel für die Ausführung einer Action, von der unter anderem auch die OpaqueAction erbt. Diese DMM-Regel heißt action.executes(String name) und bekommt als Parameter den Namen der Action, die ausgeführt wird. Wir können somit zu jedem Konzept eine DMM-Regel identifizieren, die dessen Ausführung formalisiert. Auf dieser Grundlage können wir die Prozessmuster über den konkreten Beispielmodellen in Prozessmuster über entsprechende DMM-Regeln überführen.

Für das Beispiel aus Abbildung 5.40 können wir das Prozessmuster überführen, indem wir die Objekte in den Zuständen in entsprechende Aufrufe von DMM-Regeln übersetzen. Das Objekt B:OpaqueAction entspricht danach der Ausführung der DMM-Regel action.executes("B") und das Objekt C:OpaqueAction der DMM-Regel action.executes("C"). Das Ergebnis dieser Übersetzung zeigen wir in Abbildung 5.41.

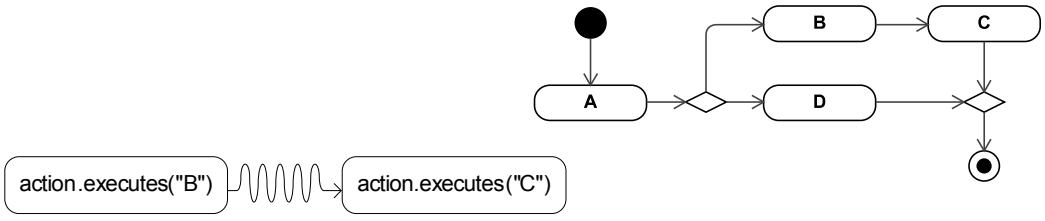


Abbildung 5.41: Prozessmuster über DMM-Regeln für Beispielmodell eines UML-Aktivitätendiagramms

Mit dem Beispiel in Abbildung 5.41 haben wir nun Prozessmuster, die über DMM-Regeln definiert sind. Um diese Prozessmuster nun auf dem Transitionssystem für das Beispielmodell von einem Model-Checker überprüfen zu lassen, müssen wir die Prozessmuster noch in eine Sprache überführen, die einem Model-Checker als Eingabe dienen kann.

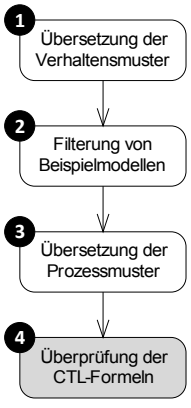
Bei Einführung der EPPSL in Abschnitt 5.2.2 haben wir erwähnt, dass Prozessmuster in EPPSL in temporal-logische Ausdrücke als CTL-Formeln [Clarke1986] übersetzt werden können. CTL ist ein Format, das von Model-Checkern, wie zum Beispiel dem Model-Checker Groove[1] [Kastenberg2006], als Eingabe verwendet werden kann.

In [Khaluf2010] wurde eine Übersetzung von EPPSL nach CTL bereits beschrieben. Für viele Konstrukte der EPPSL gibt es direkte Entsprechungen in CTL, so dass eine Übersetzung der Prozessmuster möglich ist. Für die Schlängelnotation, wie wir sie auch im Beispiel in Abbildung 5.41 verwenden, gibt es zum Beispiel die Entsprechung in CTL als $s_0 \models \boldsymbol{AF}(f)$, die aussagt, dass f auf allen Ausführungspfaden ausgehend von s_0 gültig sein wird. Damit wird das Eintreten von f als unausweichlich gefordert.

Angewandt auf das Beispiel in Abbildung 5.41 ergibt sich für die Schlängelnotation die CTL-Formel $\boldsymbol{AF}(action.executes('C'))$. Als Vorbedingung dafür, dass die Ausführung von action.executes("C") eintritt, muss zuvor action.executes("B") eingetreten sein. Diese Aussage können wir in CTL durch $s_0 \models \boldsymbol{EF}(f)$ ausdrücken. Diese Formel besagt, dass auf irgendeinem Ausführungspfad ausgehend von s_0 die Eigenschaft f eintreten wird. Damit können wir das Prozessmuster in EPPSL aus Abbildung 5.41 in folgende CTL-Formel übersetzen:

$$\boldsymbol{EF}(action.executes('B') \wedge \boldsymbol{AF}(action.executes('C')))$$

Überprüfung der CTL-Formeln



Die übersetzte CTL-Formel kann direkt auf dem Transitionssystem des Beispielmodells durch einen Model-Checker überprüft werden. In Abbildung 5.42 zeigen wir ein Bildschirmfoto des Model-Checkers Groove, mit dem wir die CTL-Formel auf dem im Hintergrund zu sehenden Transitionssystem prüfen können.

Analog können alle Prozessmuster in CTL-Formeln überführt werden, so dass für alle Beispielmodelle das geforderte Verhalten überprüft werden kann. Bestätigt der Model-Checker das geforderte Verhalten ausgedrückt durch die CTL-Formeln, haben wir eine Bestätigung, dass die Bindung korrekt ist. Diese Bestätigung durch den Model-Checker ist kein Beweis, sondern lediglich ein Testergebnis dessen Qualität von den Beispielmodellen und den gestellten Anforderungen an das Verhalten abhängig ist.

1 http://groove.sourceforge.net/

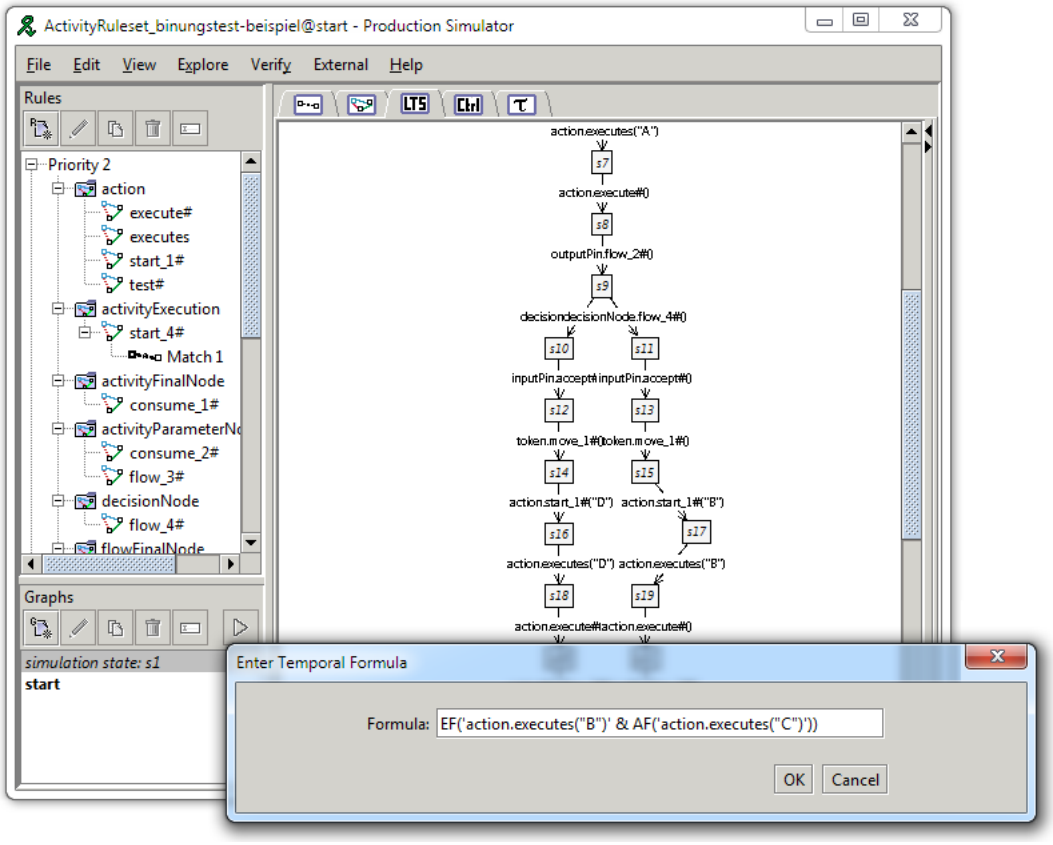


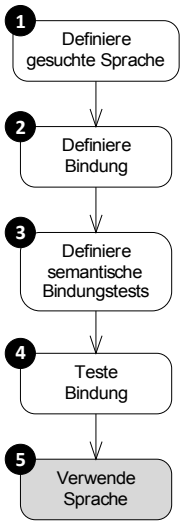
Abbildung 5.42: Prüfung der CTL-Formel im Model-Checker Groove

Der semantische Bindungstest gibt uns so eine verlässlichere Aussage über die Korrektheit der Bindung. Ohne diesen Test würden semantische Fehler nicht aufgedeckt werden können. Erst der semantische Bindungstest gibt ein Verfahren an die Hand, mit dem es möglich ist, die Bindung semantisch überprüfen zu können. Nachdem die Bindung durch den semantischen Bindungstest überprüft wurde, kann der formale Parameter eines parametrisierten Meta-Modells durch die konkrete Sprache ersetzt werden. Wir kommen damit zu Schritt 5 unseres Vorgehens in Abbildung 5.8. Die Wiederverwendung einer Sprache erläutern wir im nachfolgenden Abschnitt 5.5.

5.5 Wiederverwendung

Durch den semantischen Bindungstest haben wir die Korrektheit der Bindung überprüft. Ist die Bindung gemäß der Tests fehlerfrei, kann der formale Parameter durch die konkrete Sprache ersetzt und verwendet werden. Dies entspricht Schritt 5 unseres Vorgehens.

Auf Ebene der parametrisierten Meta-Modelle bedeutet dies, dass das Meta-Modell des formalen Parameters durch das Meta-Modell der konkreten Sprache ersetzt wird.



Die Parameterersetzung basiert auf einer korrekten Bindung. Die Bindung gibt vor, welche Klassen und welche Assoziationen wie zu ersetzen sind, indem Elemente durch ihre Abbildung ersetzt werden. Es entsteht ein neues Meta-Modell, in dem alle Klassen des Parameters durch Klassen des Meta-Modells der konkreten Sprache ersetzt sind. Wir legen die Ersetzung eines Parameters wie folgt fest:

> *Ein formaler Parameter wird durch einen konkreten Parameter ersetzt, indem gemäß der Bindung, die Klassen sowie Assoziationen des formalen Parameters durch gebundenen Klassen und Assoziationen des konkreten Parameters ersetzt werden. Etwaige vorhandene Einschränkungen werden übernommen.*

Wir werden die Parameterersetzung an einem Beispiel in Abbildung 5.43 verdeutlichen. Im oberen Teil von Abbildung 5.43 sehen wir erneut die Bindung aus Beispiel 1 in Abbildung 5.22. Bei Ersetzung des Parameters P mit der Sprache L werden die gebundenen Klassen und Assoziationen von P durch jene aus L ersetzt. Es entsteht ein Meta-Modell, das wir nun als MM+L bezeichnen.

In diesem Meta-Modell ist der Parameter durch die konkrete Sprache ersetzt worden. Zusätzlich bleibt jedoch das Wissen über die Bindung erhalten, so dass man stets nachvollziehen kann, welche Klassen und Assoziationen einst aufeinander abgebildet wurden. Dies ist insbesondere bei Verwendung von OCL-Einschränkungen wichtig, denn diese Einschränkungen werden in das resultierende Meta-Modell übernommen.

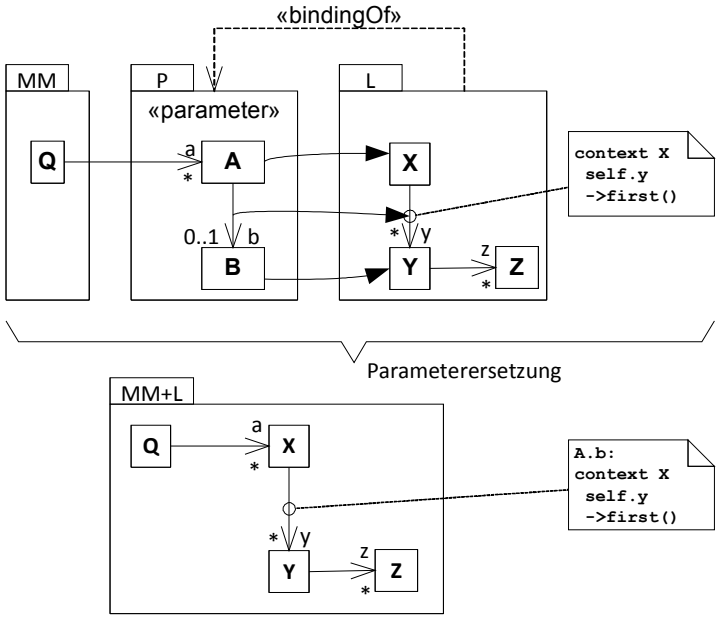


Abbildung 5.43: Wiederverwendung durch Parameterersetzung

Betrachten wir im Beispiel die Einschränkung in Abbildung 5.43, die festlegt, wie die Kardinalität der Assoziation zwischen X und Y einzuschränken ist, um der 0..1 Kardinalität zwischen A und B zu entsprechen. Diese Einschränkung muss erhalten bleiben, so dass das resultierende Meta-Modell konform zu den Anforderungen des Parameters ist.

Mit der Parameterersetzung wurde die konkrete Sprache erfolgreich wiederverwendet und integriert. Der Ansatz der anforderungsbasierten Wiederverwendung hat dabei sichergestellt, dass nur solche Sprachen wiederverwendet werden, die sowohl syntaktisch als auch semantisch zum formalen Parameter passen. Dieser Ansatz ist bestehenden Ansätzen in dieser Hinsicht überlegen, wie wir im folgenden Abschnitt 5.6 zu verwandten Arbeiten erläutern werden.

5.6 Verwandte Arbeiten

Das allgemein formulierte Problem der Wiederverwendung von Sprachen, wird häufig auch als Problem der Wiederverwendung domänenspezifischer Sprachen (engl. Domain Specific Language (DSL)) aufgefasst. Das Potential für die Verwendung von DSLs unter anderem im Bereich der modellgetriebenen Softwareentwicklung, der eng mit der Entwicklung von Meta-Modellen verbunden ist, wird beispielsweise in [Kurtev2006] hervorgehoben.

Im Sinne der Wiederverwendung von DSLs gibt es eine Vielzahl von Ansätzen. Die Wiederverwendung hat das Ziel, hohe Kosten für Neuentwicklungen von Sprachen einsparen zu können. In [Hudak1998] schlagen die Autoren zum Beispiel vor, DSLs direkt in höhere Programmiersprachen wie Haskell oder ML einzubetten. Andere betten DSLs während des Parsens ein, indem sie externe Nicht-Terminale einführen, die mittels der DSLs aufgelöst werden können [Krahn2008]. Diese Ansätze basieren jedoch nicht auf den graphentheoretischen Grundlagen der Meta-Modellierung, sondern auf Sprachmodellierung mittels Grammatiken. Daher betrachten wir diesen Zweig nicht tiefgreifend und widmen uns Ansätzen auf der Basis von Meta-Modellen.

Grundlage für die Bindung konkreter Sprachen an einen Parameter sind unter anderem Ansätze für die Komposition von Meta-Modellen. Diese Ansätze sehen eine Lösung darin, dass man das allgemeine Problem der Modell-Komposition und in diesem Zusammenhang das Problem des Mischens von Modellen (engl. model merging) löst. Hierzu muss jedoch zunächst das unterliegende Problem des Vergleichens zweier Modelle gelöst sein [Kolovos2006]. Ansätze wie in [Brunet2006] basieren daher auf dem Vergleich von Modellen, um korrespondierende Strukturen identifizieren zu können. In [Bezivin2006] präsentieren die Autoren ein kanonisches Schema zur Modell-Komposition, das als theoretische Grundlage dient, um diese Aufgabe einmal automatisch durchführen zu können. Dies ist insbesondere von Interesse, wenn man Modelle kombinieren möchte, die unterschiedliche Sichten auf ein System (z.B. ein Software-System) darstellen [Fleurey2008].

In [Ledeczi2001] werden für die Modell-Komposition drei neue Operatoren als Erweiterung der UML vorgestellt. Die vorgestellten Operatoren dienen der Konstruktion eines neuen kombinierten Meta-Modells aus zwei gegebenen

Meta-Modellen. Diese drei Operatoren sind equivalence, implementation inheritance und interface inheritance. Die Operatoren werden nicht zwischen Meta-Modellen angewendet, sondern zwischen einzelnen Klassen der zu komponierenden Meta-Modelle. Über eine equivalence Beziehung kann zum Beispiel eine Vereinigung der Eigenschaften zweier Klassen ausgedrückt werden. Die beiden Beziehungen implementation inheritance und interface inheritance drücken spezielle Vererbungsbeziehungen aus, bei denen nur bestimmte Teile der Elternklasse an die Kindklasse weitergegeben werden. Die einzelnen Operatoren sind in manchen Situationen sicherlich hilfreich, aber es fehlt insbesondere eine Betrachtung der Konzepte im Hinblick auf ihre Wiederverwendung.

Der Ansatz von [Blanc2005] bezieht sich auf die Wiederverwendung von Meta-Modellen. Auch dies ist ein konstruktiver Ansatz, der sich mit dem Problem beschäftigt, zwei Meta-Modelle im Sinne einer Wiederverwendung miteinander zu vereinigen. Hierzu definieren die Autoren die neue Beziehung «reuse and generalize», die zwischen zwei Meta-Modellen bestehen kann. Diese spezielle Beziehung beschreibt wie ein Meta-Modell Elemente eines anderen Meta-Modells wiederverwenden und dabei verallgemeinern kann. Auch dieser Ansatz geht nicht näher auf die Problematik der semantischen Bindung von Konzepten ein.

In [Muller2005] gehen die Autoren gezielt auf einige Eigenschaften der Anwendung parametrisierter Modelle ein. Sie betrachten jedoch nicht die Parametrisierung von Meta-Modellen. Die Autoren stellen einen konstruktiven Ansatz zur Erstellung von Modellen vor. Hierbei werden Modelle durch die Komposition parametrisierter Teilmodelle erstellt. Diese Teilmodelle werden im Ansatz als Template-Modelle bezeichnet. Da der Unterschied zwischen Modell und Meta-Modell in erster Linie darin besteht, wie man die Modelle interpretiert, sind Ansätze für parametrisierte Modelle prinzipiell für parametrisierte Meta-Modelle anwendbar. Hervorzuheben ist, dass die Autoren die UML und nicht etwa die MOF als Modellierungssprache verwenden. So verwenden die Autoren UML 2 Template Packages [UMLSUP23], um ein parametrisiertes Modell zu spezifizieren. In Abbildung 5.44 sehen wir ein Beispiel für ein Template-Modell, das die Struktur einer Sprache spezifiziert, mit der man Zähler (engl. counter) über Werte von Elementen beschreiben kann.

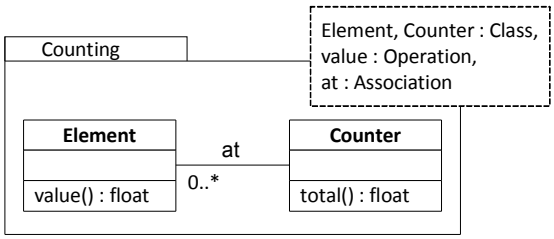


Abbildung 5.44: Counting Template aus [Muller2005]

Die Parameter des »Counting Templates« sind die beiden Klassen Element und Counter sowie die Operation value() von Element und die Assoziation at zwischen Element und Counter. Die UML erlaubt nur einzelne Modellelemente als Parameter durch Auflistung zu spezifizieren. Es ist nicht möglich ein Modell als Parameter zu verwenden. Die Auflistung der Parameter bestimmt welche

Elemente des Modells zur Bindung an Elemente eines zweiten Modells zugelassen sind.

Zur Bindung des Template an ein Modell wird der «apply» Operator verwendet. Mit diesem Operator wird angegeben welches Element des Template welchem Element des gebundenen Modells zugeordnet wird. Hierbei können nur Elemente gleichen Typs zueinander zugeordnet werden. Also Elemente vom Typ Class des Template auf Elemente vom Typ Class des Modells, Operation auf Operation usw. Aus der formalen Definition des «apply» Operators in [Muller2005] geht jedoch nicht hervor, wann Elemente gleichen Typs aufeinander abgebildet werden dürfen. Insbesondere die Abbildung von Assoziationen wird nicht genau definiert, so dass zum Beispiel unklar bleibt, wie mit Kardinalitäten zu verfahren ist. Des weiteren agiert der «apply» Operator nur auf der syntaktischen Ebene. Zur Bindung von Konzepten auf semantischer Ebene werden keine Aussagen getroffen.

In unserem Beispiel wenden wir mit Hilfe des «apply» Operators das Template aus Abbildung 5.44 auf ein Modell an, das Produkte in einem Einkaufswagen modelliert. Durch Anwendung des Template werden die Elemente des Einkaufswagen im Sinne des Templates abzählbar. Hierzu bilden wir, wie in Abbildung 5.45 gezeigt, Element auf Product, Counter auf Kart und at auf in ab. Durch Anwendung des Templates werden beide Modelle zu einem vereinigt. Hierdurch wird zum Beispiel die Klasse Product um die Operation value() aus dem Template erweitert. Das vereinigte Modell sehen wir im unteren Teil der Abbildung 5.45.

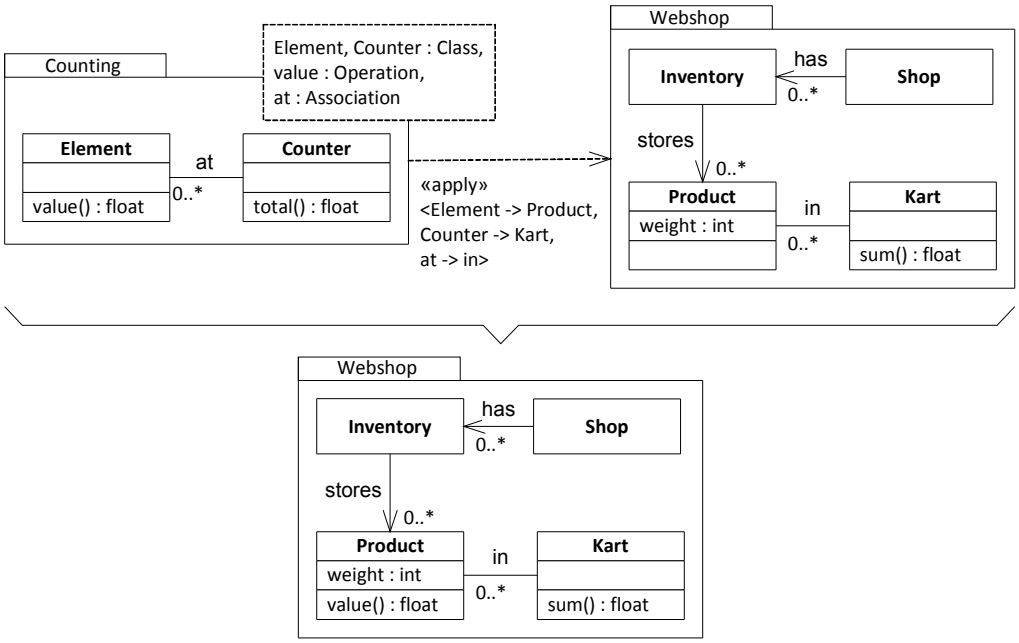


Abbildung 5.45: Anwendung des Counting Template auf das Webshop Modell

Durch Anwendung parametrisierter Modelle nach [Muller2005] können Modelle auf neue Weise komponiert werden. So ist es auch möglich Templates aufeinander anzuwenden und so eine Bibliothek aus Templates zu entwerfen.

Der Ansatz von [Muller2005] ist ein konstruktiver Ansatz zur Erstellung von Modellen. Hierbei werden gleiche Elemente aufeinander abgebildet und fehlende Elemente im Zielmodell ergänzt. Die verwendeten Templates entsprechen der Modellierung von Teilaspekten einer Sprache. Jedoch wird in diesem Ansatz keine existierende Sprache wiederverwendet, sondern aus den einzelnen Teilaspekten wird ein neues Modell konstruiert. Im Zentrum steht die Wiederverwendung der Templates zur Konstruktion – nicht die Wiederverwendung von Sprachen mittels Integration.

Ein weiterer Template-basierter Ansatz wird in [Emerson2006] vorgestellt. Hier werden Templates mit dem Ziel eingesetzt, existierende Meta-Modelle zur refaktorisieren. Wie in [Muller2005] ist dies ein konstruktiver Ansatz, um mit Hilfe eines Templates ein bestehendes Meta-Modell entsprechend den Vorgaben des Templates zu verändern. So kann der Aspekt, der durch das Template spezifiziert wird, nachträglich in ein Meta-Modell eingewoben werden. Der Ansatz fokussiert die technische Vereinigung zweier Meta-Modelle und betrachtet nicht die Frage welche Meta-Modelle für welche Aspekte eigentlich wiederverwendet werden sollen. Dementsprechend steht die Wiederverwendung ganzer Sprachen nicht im Mittelpunkt.

Neuere Ansätze, wie in [Weisemoeller2008], beschreiben Komponenten auf MOF Ebene, mit dem Konzept von angebotenen und benötigten Schnittstellen, um die Wiederverwendungsmöglichkeiten zu verbessern. In [Weisemoeller2008] zeigen die Autoren Weisemöller und Schürr einen Ansatz zur Definition und Komposition von MOF Meta-Modell Komponenten. Eine Meta-Modell Komponente verfügt über eine angebotene und eine benötigte Schnittstelle, wie man es aus der komponentenorientierten Softwareentwicklung kennt. In Abbildung 5.46 zeigen wir einen Ausschnitt aus einem Beispiel nach [Weisemoeller2008].

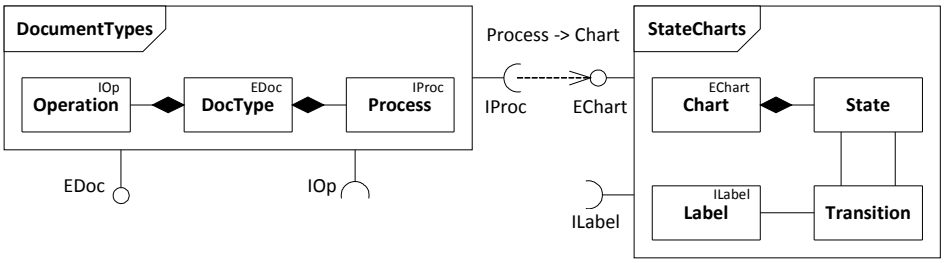


Abbildung 5.46: MOF Komponenten nach [Weisemoeller2008]

Die Meta-Modell-Komponente DocumentSpec bietet im Beispiel die Schnittstelle EDoc an und benötigt Komponenten, die die Schnittstellen IProc und IOp erfüllen. Im Beispiel haben wir die Komponente StateCharts, die eine passende Schnittstelle EChart anbietet, so dass die benötigte Schnittstelle IProc von DocumentTypes an die Schnittstelle EChart gebunden wird.

Schnittstellen sind als Teil-Meta-Modelle des gesamten Meta-Modells einer Komponente definiert. Im Beispiel in Abbildung 5.46 besteht jede Schnittstelle aus lediglich einer Klasse. Eine benötigte kann auf eine angebotene Schnittstelle abgebildet werden, wenn es einen totalen Homomorphismus zwischen den Meta-Modellen gibt, der Klassen auf Klassen und Assoziationen auf Asso-

ziationen abbildet. Im Beispiel wird so die Klasse Process auf die Klasse Chart abgebildet, wodurch die Bindung und damit die Komposition beider Komponenten festgelegt ist.

Durch die Angabe einer benötigten Schnittstelle wird festgelegt, welche anderen Komponenten gebunden werden können. Wird eine Komponente gebunden, so wird immer die gesamte Komponente gebunden. Es ist nicht möglich nur einen Teil und damit eine Teilsprache der Komponente zu binden. Außerdem wird vorausgesetzt, dass jede Komponente explizit angibt, was sie anzubieten hat. So ist es nicht möglich, frei zu bestimmen welches Teil-Meta-Modell einer Komponente gebunden werden soll. Die Schnittstellen beschreiben lediglich die angebotenen bzw. geforderten Strukturen, aber keine Semantik, so dass auch die Bindung nur auf struktureller Ebene durchgeführt wird.

Die Schnittstellen machen nach [Weisemoeller2008] keine Aussage über die Semantik der Komponenten und es ist nicht möglich nur Teile einer Komponente wiederzuverwenden, wenn man davon ausgeht, dass man eine existierende Sprache in eine Komponente überführen würde.

Einen weiteren Ansatz zeigen die Arbeiten von Juan de Lara und Esther Guerra als Teil ihres Meta-Modellierungsansatzes mit dem Namen »Meta-Depth«. Innerhalb dieses Meta-Modellierungsansatzes stellen die Autoren in [Lara2010] eine Erweiterung um das Konstrukt des »Concepts« vor. Ein »Concept« nach [Lara2010] ist gleichzusetzen mit einem Teilaspekt einer Sprache, den es umzusetzen gilt, denn ein »Concept« ist wie folgt definiert.

> *A concept in meta-modelling is a pattern specification that expresses requirements on models or meta-models.*
>
> **Concept nach [Lara2010]**

Nach dieser Definition entspricht ein »Concept« einem Parameter im Ansatz eines parametrisierten Meta-Modells. Die Autoren verwenden das Konstrukt des »Concept«, um ein bestimmtes Verhalten eines Meta-Modells unabhängig von einem konkreten Meta-Modell auf Basis eines »Concept«, zu sehen in Abbildung 5.47, beschreiben zu können.

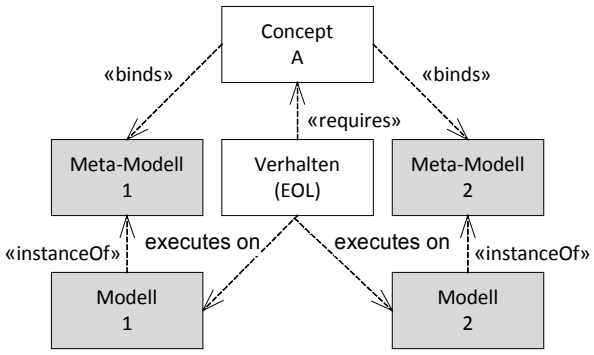


Abbildung 5.47: »Concept« Ansatz nach [Lara2010]

In MetaDepth kann das Verhalten eines Modells mit Hilfe der »Epsilon Object Language« (EOL) [Kolovos2006a] spezifiziert werden. Die EOL ist eine Erweiterung der »Object Constraint Language« (OCL) [OCL20] um imperative Konstrukte, um Modelle manipulieren zu können.

In Abbildung 5.47 erkennt man, dass ein bestimmtes Verhalten ein Concept benötigt («requires»). Das Concept wird beschrieben durch ein Meta-Modell, dass alle Elemente spezifiziert, die für das auszudrückende Verhalten notwendig sind. In diesem Sinne ist ein Concept ein minimales Meta-Modell für ein gegebenes Verhalten. Das Meta-Modell des Concept kann nun an unterschiedliche Meta-Modelle gebunden («binds») werden. Durch diese Bindung wird sichergestellt, dass das Concept auch Teil des gebundenen Meta-Modells ist. Erzeugt man nun Modelle, also Instanzen der gebundenen Meta-Modelle, so ist durch die Bindung an das Concept sichergestellt, dass das beschriebene Verhalten auf den Modell-Instanzen ausgeführt werden kann. Auf diese Weise kann das Verhalten auf abstrakte Weise über ein Concept spezifiziert werden und nach Bindung an konkrete Meta-Modelle auf diesen ausgeführt werden.

Die konkreten Meta-Modelle können hierbei existierende Sprachen sein, so dass durch das »Concept« vorgegeben wird, welche Sprachen gebunden werden können. Hierbei definiert das »Concept« die Struktur, also die geforderte abstrakte Syntax. Das Verhalten des »Concepts« ist über EOL definiert, so dass man theoretisch prüfen könnte, ob dieses Verhalten zum gegebenen Verhalten einer existierenden Sprache kompatibel ist. Diese Form der Prüfung wird von den Autoren in [Lara2010] jedoch nicht berücksichtigt, da auch nicht definiert wird, wie das Verhalten der gebundenen Meta-Modelle spezifiziert ist. Die Autoren treffen die Annahme, dass durch die Bindung auch das Verhalten an das gebundene Meta-Modell übergeben wird, unabhängig davon, welches Verhalten das Meta-Modell gegebenenfalls zuvor hatte. Einen semantischen Bindungstest sieht der Ansatz nicht vor, so dass unklar bleibt, wie dieses Problem zu lösen wäre.

Ein weiterer Ansatz besteht darin, die Techniken aus der Welt der Software-Produktlinien und deren Modellierung mittels Feature-Modellen, zu sehen in Abbildung 5.48, auf die Sprachentwicklung zu übertragen.

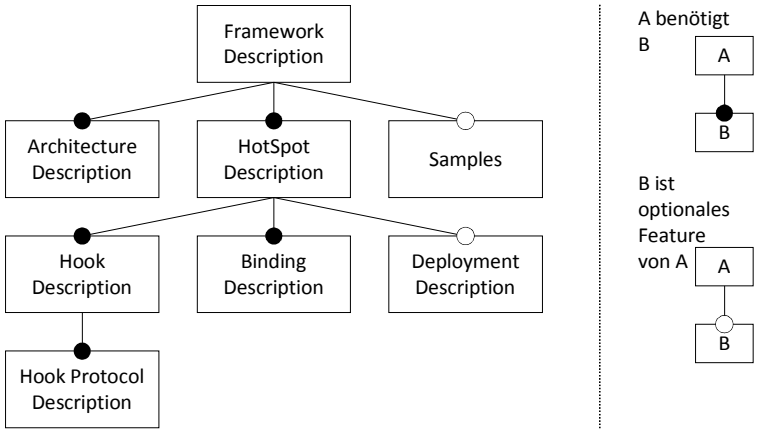


Abbildung 5.48: Sprachaspekte als Feature-Modell

Nach diesem Ansatz werden Varianten von Sprachen ebenfalls als Feature-Modelle beschrieben [White2009]. Variationspunkte wie in Feature-Modellen drücken parametrisierte Meta-Modelle durch die Deklaration und die Bindung von Parametern aus, so dass auch dieses Konzept integriert werden konnte.

Der Einsatz von Feature-Modellen bietet eine Möglichkeit, um die Komposition einer Sprache aus den wiederverwendeten Sprachen zu modellieren. In Abbildung 5.48 sehen wir jedoch, dass dies ist nur auf einem abstrakten Niveau vorgesehen ist. Die Details der Bindung zwischen den einzelnen Sprachfeatures werden nicht behandelt.

Die bisherigen Ansätze haben das Problem, eine passende Sprache zur Wiederverwendung zu finden, vornehmlich auf der syntaktischen Ebene behandelt. Durch ein Modell wird die abstrakte Syntax einer gesuchten Sprache definiert und ein Mensch muss durch scharfes Hinsehen beurteilen, ob eine gegebene Sprache zu diesen Anforderungen passt. Als einziger Ansatz wird in [Lara2010] beschrieben, dass ein dortiges »Concept« semantisch durch die Spezifikation eines Simulationsalgorithmus beschrieben wird. Leider beschreiben die Autoren nicht, wie man prüft, ob eine gegebene Sprache eine kompatible Semantik erfüllt.

Ein Weg hin zu mehr Semantik in dieser Problemstellung wird mit dem Einsatz von Ontologien erforscht. Allgemein lässt sich festhalten, dass bei Meta-Modellierung und modellgetriebener Entwicklung Ontologien immer vorhanden sind, auch wenn diese häufig nicht explizit gemacht werden. In [Terrasse2006] wird auf die enge Verknüpfung zwischen Meta-Modellierung und dem Einsatz von Ontologien hingewiesen. Dort wird die Frage aufgeworfen, ob man für eine Meta-Modellierung Meta-Modelle und Ontologien benötigt. Diese Fragestellung ist noch immer Teil des wissenschaftlichen Diskurses und daher eine allgemeine Antwort kaum möglich. Grundsätzlich ermöglichen es Ontologien mehr über semantische Zusammenhänge auszudrücken, aber die Prüfung, ob zwei Ontologien zueinander kompatibel sind, ist ein mindestens so schweres Problem wie zu bestimmen ob zwei Meta-Modelle kompatibel sind. Von daher erscheint der Ansatz über Ontologien das Problem lediglich zu verlagern, aber nicht endgültig zu lösen.

Eine Kombination von abstrakter Syntax einer Sprache und Ontologien wird z.B. in [Walter2009] aufgezeigt. Opdahl und Berio beschreiben in diesem Zusammenhang interessante Ideen, die sie für die Definition einer »Unified Enterprise Modelling Language« verwenden [Opdahl2006]. Sie verwenden eine Technik genannt »separation of reference«, die eine Interoperabilität zwischen Modellen ermöglicht, indem Teile eines Modells in Konzepte einer Modell-Ontologie überführt werden. Die Konzepte dieser Ontologie, werden dann auf eine weitere, allgemeine Ontologie abgebildet. Wenn man die Konzepte von zwei Modellen auf gleiche weise auf diese allgemeine Ontologie abbilden kann, so sind diese beiden Modelle interoperabel, da sie gleiche Konzepte bedienen. Es ist jedoch fragwürdig ob die Abbildung in höhere Ontologien wirklich zum Erfolg führt. Uns ist kein Ansatz bekannt der diese Techniken erfolgreich einsetzt, um etwa die Wiederverwendung einer Sprache automatisieren zu können.

Zusammenfassend ist uns kein Ansatz bekannt, der semantische Eigenschaften bei Festlegung einer Bindung überprüfbar macht. Der hier vorgestellte Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen erfüllt hingegen alle gestellten Anforderungen. Das Verfahren enthält mit dem semantischen Bindungstest eine Prüfmethode, um die Bindung auf ihre Korrektheit hin zu testen. Diese zusätzliche Möglichkeit unterscheidet das neue Verfahren von existierenden Ansätzen.

5.7 Zusammenfassung

In diesem Kapitel haben wir den Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen vorgestellt. Mit Hilfe der Modellierungstechnik der parametrisierten Meta-Modelle können wir stabile Sprachkerne definieren, die über Parameter andere existierende Sprachen für bestimmte Teilaspekte integrieren können. Der beschriebene Ansatz und die Modellierungstechnik erlauben es uns, ein parametrisiertes Meta-Modell für unsere gesuchte Framework-Beschreibungssprache aufzustellen, das die Anforderungen an die Wiederverwendung von Sprachen erfüllt. Mit Hilfe des semantischen Bindungstests können wir überprüfen, ob die Bindung zwischen Parametern und wiederverwendeten Sprachen auch semantisch valide sind. Eine Überprüfung, die bestehende Ansätze nicht leisten können.

In Kapitel 6 werden wir uns nun der Definition eines parametrisierten Meta-Modells für Framework-Beschreibungen widmen. Bestimmte Aspekte dieser Framework-Beschreibung werden als Parameter definiert sein, für die wir außerdem zeigen, wie durch die Anwendung des Ansatzes einer anforderungsbasierten Wiederverwendung von Sprachen, existierende Sprachen für diese Aspekte wiederverwendet werden können.

6

Framework Description Meta-Model

Der Geist einer Sprache
offenbart sich am deutlichsten
in ihren unübersetzbaren Worten.

– Marie von Ebner-Eschenbach

Wie im Abschnitt 4.2 »Existierende Ansätze zur Framework-Beschreibung« dargelegt, erfüllen die bestehenden Ansätze nicht die Anforderungen aus Abschnitt 4.1 an eine Framework-Beschreibungssprache. Unser Ziel ist daher, eine neue Framework-Beschreibungssprache zu definieren, die alle Anforderungen erfüllt. Diese Sprache definieren wir über ein Meta-Modell für Framework-Beschreibungssprachen mit dem Namen »Framework Description Meta-Model«, kurz FDMM. Das FDMM ist ein parametrisiertes Meta-Modell, das es erlaubt, existierende Sprachen für bestimmte Aspekte einer Framework-Beschreibung wiederzuverwenden. Hierzu verwenden wir den in Kapitel 5 vorgestellten Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen.

Eine Motivation für die Entwicklung eines Ansatzes zur anforderungsbasierten Wiederverwendung von Sprachen war es, eine flexible Möglichkeit zu schaffen, um Sprachen für das konkrete Problem einer ganzheitlichen

Framework-Beschreibungssprache wiederzuverwenden. Die Modellierung des FDMM als parametrisiertes Meta-Modell liefert uns die notwendige Flexibilität gemäß unserer Anforderungen ERW-B, ERW-E und ERW-W aus Abschnitt 4.1.3 an die Erweiterbarkeit von Framework-Beschreibungssprachen.

In Abschnitt 6.1 geben wir eine Übersicht über das FDMM und zeigen, wo im FDMM die gestellten Anforderungen an eine ganzheitliche Framework-Beschreibungssprache umgesetzt werden.

In Abschnitt 6.2 werden wir auf die Details des FDMM als parametrisiertes Meta-Modell eingehen und die einzelnen Aspekte einer vereinheitlichten Framework-Beschreibungssprache betrachten. In Abschnitt 6.3 zeigen wir außerdem die Bindung der Parameter des FDMM an konkrete Sprachen. Die beiden Abschnitte 6.2 und 6.3 bilden die Referenzdokumentation des FDMM für den geneigten Leser und gehen auf viele Details ein. Beide Abschnitte können jedoch nach belieben übersprungen werden, wenn diese Details nicht von unmittelbarem Interesse sind.

Die Anwendbarkeit des FDMM belegen wir durch einige Fallstudien, die wir in Abschnitt 6.4 erläutern. Wir werden hierzu keine vollständige Beschreibung eines Frameworks präsentieren, sondern darauf eingehen, welche Anforderungen an die Ausdrucksfähigkeit von Framework-Beschreibungssprachen in einzelnen Fallstudien erfolgreich umgesetzt werden konnten.

Abschließend zeigen wir in Abschnitt 6.5 wie sich das FDMM in bestehende Framework-basierte Entwicklungsprozesse integrieren lässt. Wir kommen zu dem Schluss, dass das FDMM die Framework-basierte Entwicklung ideal ergänzt, da mit dem FDMM die Frage nach dem »Wie« zur Erstellung einer Beschreibung beantwortet wird.

6.1 Übersicht

Wir stellen in der Übersicht in Abbildung 6.1 das FDMM als Paketdiagramm dar. Als modular aufgebautes Meta-Modell kapseln wir die Teilaspekte einer Framework-Beschreibung in unterschiedliche Pakete. Jedes Paket enthält dabei die Klassen des Meta-Modells, die für die Beschreibung des Aspektes notwendig sind. Durch die Verwendung von Import-Beziehungen werden die Pakete miteinander verknüpft, so dass das FDMM bestehend aus den Paketen für die Teilaspekte entsteht.

Das FDMM verfügt über drei Teilaspekte, für die wir existierende Sprachen wiederverwenden werden. Diese drei Teilaspekte sind dementsprechend als Parameter definiert und in Abbildung 6.1 zur Verdeutlichung grau hinterlegt worden. Diese drei Parameter sind:

- BehaviorDL: Zur Beschreibung von Abläufen in der Framework-Architektur wollen wir Sprachen wiederverwenden, die in der Lage sind, sequentielle Abläufe zu beschreiben.
- ProtocolDL: Zur Beschreibung eines Nachrichtenaustauschs zwischen einem Framework und einer Erweiterung wollen wir Sprachen wiederverwenden, die derartige Protokolle beschreiben können.

- APIDL: Zur Beschreibung der Programmierschnittstelle eines Frameworks werden häufig programmiersprachenspezifische Beschreibungsformen eingesetzt. Wir wollen die Sprachen zur Beschreibung der API wiederverwenden können, die sich für bestimmte Programmiersprachen etabliert haben.

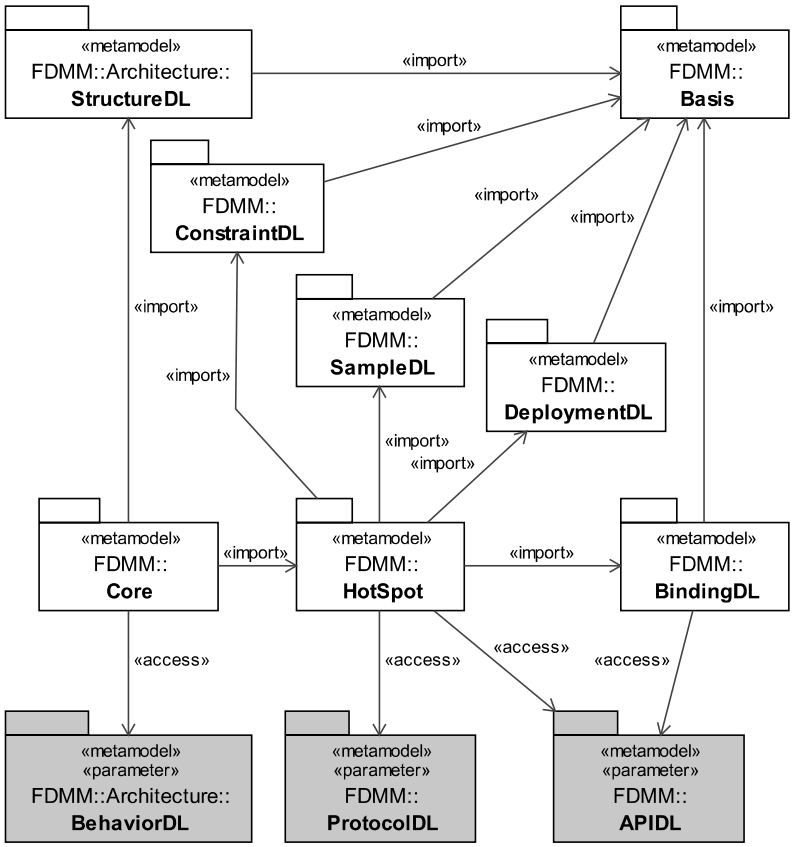


Abbildung 6.1: Pakete des FDMM

Durch den modularen Aufbau des FDMM setzen wir die Anforderungen an die Erweiterbarkeit einer Framework-Beschreibungssprache aus Abschnitt 4.1.3 um. In Tabelle 6.1 dokumentieren wir Details dieser Umsetzung.

Tabelle 6.1: Umsetzung der Anforderungen an die Erweiterbarkeit

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Umsetzung im FDMM
ERW-B	*Framework-Benutzer* ... existierende Sprachen, mit denen Framework-Benutzer bereits vertraut sind, zur Beschreibung einzelner Merkmale integrieren können.	Als parametrisiertes Meta-Modell können im FDMM existierende Sprachen wiederverwendet werden, die Framework-Benutzern bereits vertraut sind.

ID	Zielgruppe & Anforderung Eine Framework-Beschreibungssprache muss...	Umsetzung im FDMM
ERW-E	*Framework-Entwickler* ... es den Framework-Entwicklern ermöglichen, die Framework-Beschreibung um neue Beschreibungsformen für Erweiterungspunkte zu erweitern.	Ein modular aufgebautes Meta-Modell wie das FDMM kann durch neue Pakete erweitert werden. So kann auch die Beschreibung für Erweiterungspunkte durch neue Pakete erweitert werden.
ERW-W	*Werkzeug-Entwickler* ... müssen in der Lage sein, Werkzeuge auf Basis der Sprachdefinition der Framework-Beschreibungssprache zu erweitern.	Werkzeuge, die auf Basis eines Meta-Modells arbeiten können so gestaltet sein, dass sie Veränderungen am Meta-Modell leicht berücksichtigen können. Als Meta-Modell bringt das FDMM alle Voraussetzungen hierfür mit.

Neben den Anforderungen an die Erweiterbarkeit muss das FDMM die Anforderungen an die Ausdrucksfähigkeit aus Abschnitt 4.1.2 umsetzen. Wir wollen die Umsetzung der Ausdrucksfähigkeit im Folgenden näher betrachten.

Den Kern des FDMM, zu sehen in Abbildung 6.1, bildet das Paket Core, das über import und access Beziehungen alle weiteren Pakete des FDMM inkludiert. Das Core-Paket importiert alle Elemente des HotSpot-Paketes und transitiv ebenfalls alle Elemente des Basis-Paketes. So vereinigt Core alle Elemente, die im FDMM definiert werden.

Ein Teilaspekt des FDMM erfasst die Beschreibung der Architektur eines Frameworks, wie sie für einen Framework-Benutzer notwendig ist. Anforderung AUS-2 besagt, dass statische und dynamische Architekturübersichten beschrieben werden können müssen. Hierfür definieren wir Architekturbeschreibungssprachen zur Beschreibung des Aufbaus (StructureDL) und des Verhaltens (BehaviorDL) eines Frameworks. Durch Verweise aus den Architekturbeschreibungen auf verfügbare Erweiterungspunkte, bekommt der Framework-Benutzer, wie in AUS-2 gefordert, einen Überblick und erkennt, wo das Framework erweitert werden kann. Der strukturelle Aufbau wird im Paket StructureDL und das Verhalten im Sinne von Abläufen im Framework im Paket BehaviorDL erfasst. Zur Beschreibung der Abläufe wollen wir auf eine existierende Sprache zurückgreifen, die sich für diesen Anwendungszweck eignet. Daher deklarieren wir die BehaviorDL als Parameter des FDMM.

In Software-Stacks werden in der Regel mehrere Frameworks gleichzeitig verwendet. Daher muss neben der Erweiterung des Frameworks über Erweiterungspunkte auch beschrieben werden, wie man das Framework selbst konfiguriert und in andere Systeme integrieren kann (AUS-4). Die Beschreibung der Programmierschnittstelle des Frameworks wird hierbei durch das Paket APIDL definiert.

Die Erweiterungspunkte eines Frameworks bilden den zentralen Aspekt einer Framework-Beschreibung und müssen gemäß Anforderung AUS-3 beschrieben werden können. Erweiterungspunkte eines Frameworks werden über die Sprache im Paket HotSpot beschrieben. Mit Hilfe dieser Sprache kann beschrieben werden, was bei der Implementierung einer Erweiterung zu einem Erweiterungspunkt gemäß Anforderung AUS-5 zu beachten ist. Das FDMM berücksichtigt die folgenden Aspekte bei der Beschreibung von Erweiterungspunkten:

- Beschreibung von Beispielen, die die Benutzung eines Erweiterungspunktes in konkreten Szenarien gemäß Anforderung AUS-1 zeigen. Der Aufbau von Beispielen wird im Paket SampleDL definiert.
- Beschreibung der Kommunikation zwischen dem Framework und einer Erweiterung des Erweiterungspunktes über ein Kommunikationsprotokoll gemäß Anforderung AUS-6. Kommunikationsprotokolle werden im Paket ProtocolDL definiert.
- Beschreibung der Bindung zwischen Erweiterungen des Erweiterungspunktes und dem Framework gemäß Anforderung AUS-7. Die Bindung sorgt dafür, dass die Erweiterung dem Framework bekannt gemacht und genutzt wird. Die Bindung wird im Paket BindingDL definiert.
- Beschreibung der Installation einer Erweiterung im Framework gemäß Anforderung AUS-8, so dass die Erweiterung dem Framework zur Verfügung steht. Details der Installation werden im Paket DeploymentDL definiert.
- Beschreibung einzuhaltender Einschränkungen, die eine Implementierung einer Erweiterung zum Erweiterungspunkt gemäß Anforderung AUS-9 zu berücksichtigen sind. Einschränkungen werden im Paket ConstraintDL definiert.

Zusammenfassend stellen wir die Umsetzung der Anforderungen an die Ausdrucksfähigkeit in Tabelle 6.2 noch einmal übersichtlich dar.

Tabelle 6.2: Umsetzung der Anforderungen an die Ausdrucksfähigkeit

ID	Anforderung an die Ausdrucksfähigkeit Eine Framework-Beschreibungssprache muss...	Umsetzung im Paket
AUS-1	... Beispiele für die Benutzung des Frameworks beschreiben können.	SampleDL
AUS-2	... statische und dynamische Architekturübersichten beschreiben können.	StructureDL BehaviorDL
AUS-3	... die Erweiterungspunkte eines Frameworks beschreiben können.	Hotspot
AUS-4	... die Konfiguration und Integration des Frameworks beschreiben können.	APIDL

ID	Anforderung an die Ausdrucksfähigkeit Eine Framework-Beschreibungssprache muss...	Umsetzung im Paket
AUS-5	... die Implementierung einer Erweiterung beschreiben können.	HotSpot
AUS-6	... das Kommunikationsprotokoll zwischen Framework und Erweiterung beschreiben können.	ProtocolDL
AUS-7	... die Bindung von Erweiterungen an das Framework beschreiben können.	BindingDL
AUS-8	... das Format und die Installation einer Erweiterung in das Framework beschreiben können.	DeploymentDL
AUS-9	... einzuhaltende Bedingungen seitens einer Erweiterung beschreiben können.	ConstraintDL
AUS-10	... Varianten von Erweiterungspunkten beschreiben können.	HotSpot

Wie gezeigt setzt das FDMM die Anforderungen an die Erweiterbarkeit und die Ausdrucksfähigkeit für Framework-Beschreibungssprachen um. Die Anforderungen an die Anwendbarkeit haben wir noch nicht behandelt, da diese mit der Implementierung des FDMM als Werkzeug in Abschnitt 6.4 umgesetzt werden.

Wir definieren in dieser Arbeit keine konkrete Syntax für Instanzen des FDMM. Technisch lässt sich jede FDMM-Instanz in einem XML-Format speichern, wobei das FDMM direkt das Schema des XML-Formats vorgibt. In diesem Sinne wäre das XML-Format eine konkrete Syntax zur Erfassung von FDMM-Instanzen. Anstelle einer konkreten Syntax haben wir das FDMM in einem Werkzeug zur Erstellung von FDMM-basierten Framework-Beschreibungen implementiert. Auf das Werkzeug und die Implementierung gehen wir in Kapitel 7 ein.

In den folgenden Abschnitten 6.2.1 bis 6.2.11 werden wir das FDMM in einer Referenz anhand seiner Pakete detailliert beschreiben. In Abschnitt 6.3 gehen wir außerdem auf die Bindung der Parameter des FDMM genauer ein. Die technische Dokumentation des FDMM an dieser Stelle der Arbeit dient der Vollständigkeit und als Referenz. Derjenige Leser, der diese Details nicht benötigt, kann diese Abschnitte überspringen und in Abschnitt 6.5 mit der »Integration in Framework-basierte Entwicklungsprozesse« fortfahren.

6.2 Referenz

In den folgenden Abschnitten 6.2.1 bis 6.2.11 dokumentieren wir die Details des FDMM. Diese Referenz beschreibt alle Pakete und Klassen des FDMM und dient der technischen Dokumentation.

6.2.1 FDMM::Basis

Im Basis-Paket, zu sehen in Abbildung 6.2, befinden sich grundlegende Elemente, die in den anderen Paketen des FDMM benutzt werden. Zur verwendeten UML-Notation ist anzumerken, dass Schnittstellen als Klassen mit einem Kreis in der oberen rechten Ecke dargestellt werden. Als Wurzelelement wird im Basis-Paket die Schnittstelle FDObject definiert, die alle Elemente des FDMM implementiert. Dies stellt eine eindeutige Typisierung aller Elemente als FDObject sicher. FDObject steht für »Framework Description Object«. Elemente des FDMM mit einem Namen implementieren die Schnittstelle NamedElement und Elemente mit einem beschreibenden Text die Schnittstelle DescriptionElement. Zur Vereinfachung gibt es bereits die abstrakte Klasse Description im Basis-Paket, die ein benanntes Element mit einer Beschreibung repräsentiert.

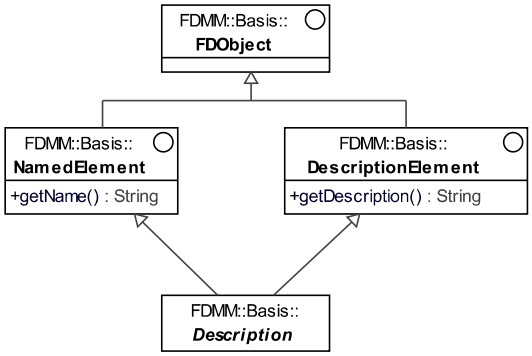


Abbildung 6.2: FDMM::Basis

6.2.2 FDMM::Core

Das Wurzelelement von FDMM::Core bildet die Framework-Beschreibung, ausgedrückt durch FrameworkDescription. Ausgehend von der FrameworkDescription kann über die assoziierten Klassen auf alle zugehörigen Informationen zugegriffen werden. Abbildung 6.3 zeigt den Inhalt des Paketes Core.

Das Paket Core enthält Klassen, die den Aufbau einer Framework-Beschreibung definieren. Nach [Meusel1997] sollte man Dokumentationen nach dem Pyramidenprinzip aufbauen, das heißt von allgemeinen Konzepten hin zu konkreteren Details. Wir unterstützen dieses Prinzip im FDMM, indem wir mit den Mitteln der Komposition von Übersichten auf weitere Details verweisen.

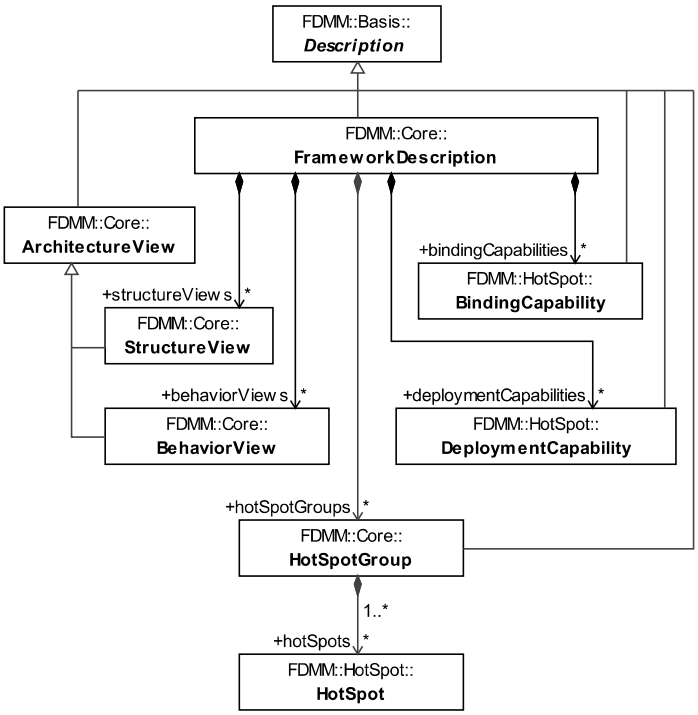


Abbildung 6.3: FDMM::Core

Beschreibungen der Architektur des Frameworks repräsentiert durch das Element ArchitectureView ist ein Beispiel für eine Übersicht, die auf weitere Details verweist. Das Element ArchitectureView erfüllt die Anforderung an Framework-Beschreibungen, statische und dynamische Architekturübersichten (siehe Abschnitt 4.1.2, Seite 79) beschreiben zu können. Hierzu werden die StructureView und die BehaviorView als spezielle Unterklassen von ArchitectureView deklariert. Mit einer StructureView lässt sich die architektonische Struktur des Frameworks beschreiben und mit einer BehaviorView dynamische Abläufe innerhalb des Frameworks. Durch Verweise auf Gruppen von Erweiterungspunkten aus diesen Architektursichten gelangt ein Framework-Benutzer von der Übersicht zu den Details. Wir werden in den Paketen StructureDL in Abschnitt 6.2.3 und BehaviorDL in Abschnitt 6.2.4 auf diese Aspekte eingehen.

Als weitere Elemente des Core-Paketes haben wir die Klassen BindingCapability zur Beschreibung von Bindungsmöglichkeiten und DeploymentCapability zur Beschreibung von Installationsmechanismen. Details zu diesen beiden Aspekten werden über die Pakete BindingDL in Abschnitt 6.2.7 und DeploymentDL in Abschnitt 6.2.9 spezifiziert.

6.2.3 FDMM::Architecture::StructureDL

Der Ausgangspunkt einer strukturellen Architekturbeschreibung ausgedrückt durch das Meta-Modell der StructureDL in Abbildung 6.4 ist die Klasse ArchitectureDescription. Ausgehend von der ArchitectureDescription baut sich eine Architekturbeschreibung aus architektonischen Elementen (ArchitectureElement)

und Beziehungen zwischen diesen auf. Es wird von der StructureDL nicht festgelegt, was ein architektonisches Element ist und die Beziehung ist definiert als eine Abhängigkeitsbeziehung (depends on).

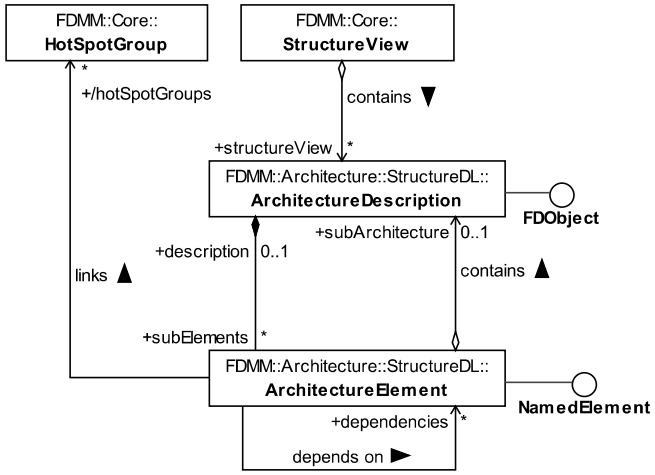


Abbildung 6.4: FDMM::Architecture::StructureDL

Beispiele für architektonische Elemente sind »Systeme«, »Ebenen« (engl. »Layer«), »Komponenten« oder »Module«. Beziehungen zwischen diesen Elementen sind geprägt durch Abhängigkeiten. Betrachtet man beispielsweise »Ebenen«, so kann man die Abhängigkeit zweier »Ebenen« dadurch ausdrücken, dass eine Ebene oberhalb und eine unterhalb liegt, indem die obere von der unten abhängt. Zwischen »Komponenten« und »Systemen« gibt es Abhängigkeitsbeziehungen, wie beispielsweise die Abhängigkeit eines »Clients« vom »Server«. Zur Verbindung architektonischer Elemente und Erweiterungspunkten kann zu jedem ArchitectureElement eine Liste von HotSpotGroups angegeben werden.

Für die Beschreibung der Architektur des Frameworks aus Sicht des Framework-Benutzers ist eine derart abstrakte Sicht ausreichend. Wichtig ist, einen Überblick zu erhalten und zu erkennen, an welchen Stellen Erweiterungspunkte angeboten werden.

Die StructureDL ist nicht als Parameter des FDMM deklariert worden, da hier keine direkte Wiederverwendung einer Sprache für die Struktur einer Architektur vorgesehen ist. Nichtsdestotrotz wäre dies sicherlich möglich. Wir legen noch fest, dass das Element ArchitectureDescription die Schnittstelle FDObject und ArchitectureElement die Schnittstelle NamedElement aus der Basis implementiert.

6.2.4 FDMM::Architecture::BehaviorDL

Die BehaviorDL haben wir bei Einführung des Ansatzes zur anforderungsbasierten Wiederverwendung von Sprachen bereits in Teilen vorgestellt. Da die BehaviorDL ein Parameter des FDMM ist, stellen wir gemäß unserem Ansatz zunächst eine Menge von Anwendungsfällen aus Sicht eines Framework-

Entwicklers auf. Die Anwendungsfälle haben wir in Abbildung 5.9 bereits vorgestellt, so dass wir diese hier nicht nochmals wiederholen wollen.

Auf Grundlage der Anwendungsfälle erstellen wir eine Sprachanforderungen aus Meta-Modell und Verhaltensmustern. In Abbildung 6.5 stellen wir zunächst das Meta-Modell der BehaviorDL vor.

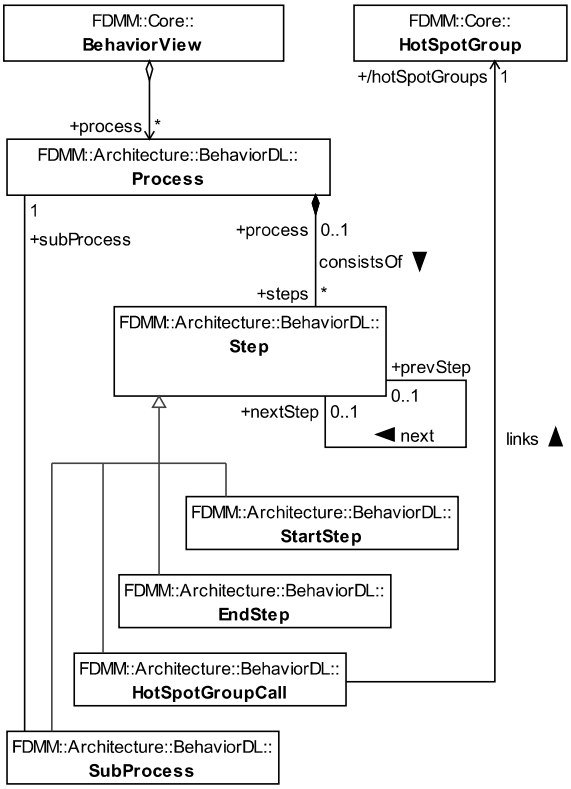


Abbildung 6.5: FDMM::Architecture::BehaviorDL

Im Meta-Modell der BehaviorDL wird für die Beschreibung von Abläufen die Klasse BehaviorView mit der Klasse Process assoziiert. Ein Process besteht aus einer Menge von Steps, wobei es verschiedene spezialisierte Unterarten von Steps gibt. So beginnt ein Process mit einem StartStep und endet mit einem EndStep. Ein Step kann selber wieder einen eigenen Process aufrufen, wenn der Step ein SubProcess ist. Um darzustellen, dass in bestimmten Steps bestimmte HotSpotGroups verwendet werden, kann ein Step auch ein HotSpotGroupCall sein. An diesen Steps erkennt ein Framework-Benutzer wo Erweiterungspunkte des Frameworks im Ablauf aufgerufen werden.

Das Meta-Modell der BehaviorDL stellt die geforderte Struktur und verwendete Konzepte dar. Nun muss noch das geforderte Verhalten in der Sprachanforderung spezifiziert werden. Einige Verhaltensmuster haben wir hierzu bereits in Abschnitt 5.2.2 gezeigt. Wir wollen an dieser Stelle die noch fehlenden Verhaltensmuster ergänzen. Zunächst betrachten wir den Aufruf eines SubProcess in Abbildung 6.6. Das Verhaltensmuster drückt aus, dass, wenn ein Step ein SubProcess ist. So wird zunächst der SubProcess ausgeführt, bevor der nachfolgende Step des Elternprozesses ausgeführt wird.

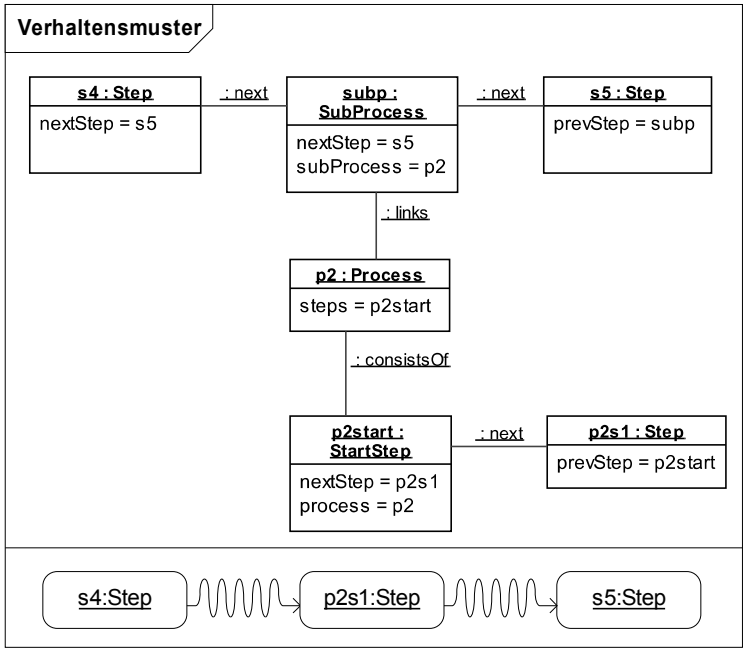


Abbildung 6.6: Verhaltensmuster für einen SubProcess der BehaviorDL

Bei Aufruf eines HotSpotGroupCalls muss noch festgelegt werden, dass sich ein solcher Aufruf im Verhalten nicht anders verhält, wie ein Step. Hierfür definieren wir das Verhaltensmuster in Abbildung 6.7.

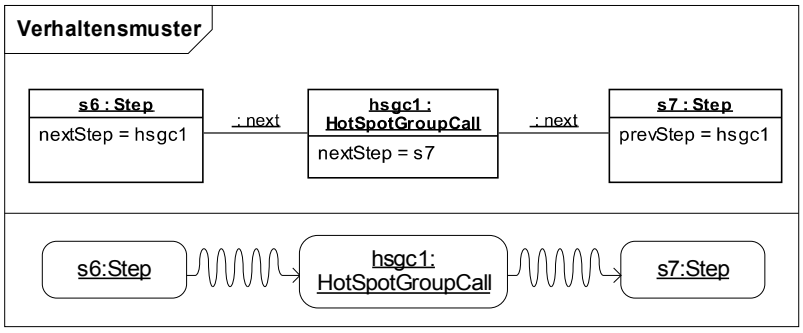


Abbildung 6.7: Verhaltensmuster für HotSpotGroupCalls der BehaviorDL

Das letzte Verhaltensmuster in Abbildung 6.8 definiert das Ende eines Process, wenn ein EndStep erreicht wird. Dabei gibt das Instanzmuster die Situation vor, dass ein Step als nextStep mit einem EndStep verlinkt ist. Für die Ausführung bedeutet dies, dass nach Ausführung des Step die Ausführung endet.

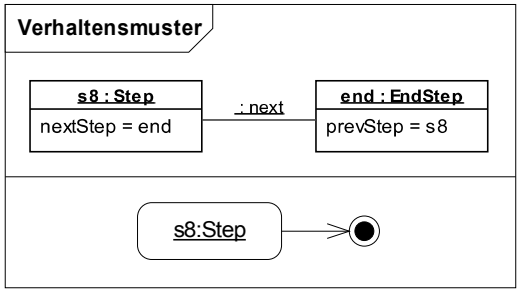


Abbildung 6.8: Verhaltensmuster für einen EndStep der BehaviorDL

Mit den Verhaltensmustern und dem Meta-Modell der BehaviorDL haben wir die Sprachanforderung des Parameters vollständig spezifiziert. In Abschnitt 6.3 werden wir zeigen wie die Sprache der UML-Aktivitätendiagramme als eine konkrete Sprache an die BehaviorDL gebunden werden kann.

6.2.5 FDMM::HotSpot

Das Paket HotSpot ist das Herzstück einer Framework-Beschreibung, da hier alle Elemente zusammengefasst werden, die sich mit der Beschreibung von Erweiterungspunkten, dem zentralen Framework-Konzept, befassen. Aufgrund der unterschiedlichen Aspekte eines HotSpots, importiert dieses Paket sechs weitere Pakete. Ein Übersicht zeigen wir in Abbildung 6.9.

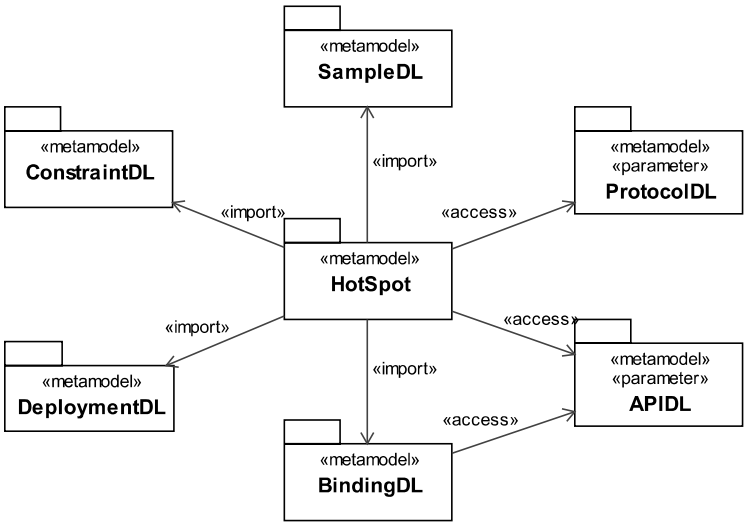


Abbildung 6.9: HotSpot-Paket mit Parametern

Die sechs Pakete sind die SampleDL zur Erfassung veranschaulichender Beispiele, die ProtocolDL zur Beschreibung von Kommunikationsprotokollen zwischen Framework und Erweiterung, die APIDL für Verweise auf eine API-Dokumentation, die ConstraintDL zur Beschreibung von zu beachtenden Einschränkungen, die BindingDL zur Beschreibung der Bindung von Erweiterungen

an das Framework und schließlich die DeploymentDL zur Beschreibung der korrekten Installation einer Erweiterung in die Laufzeitumgebung eines Frameworks.

Zwei dieser sechs Pakete sind als Meta-Modell Parameter deklariert. Diese sind ProtocolDL und APIDL, da für diese beiden Aspekte jeweils existierende Sprachen wiederverwendet werden sollen. Die Abbildung 6.9 zeigt Pakete und Parameter des HotSpot Paketes. Zu beachten ist, dass das Paket BindingDL die APIDL selbst wiederum als Parameter verwendet, was die Flexibilität der Modellierung mit parametrisierten Meta-Modellen verdeutlicht.

Das Paket HotSpot, zu sehen in Abbildung 6.10, verfügt über die Klassen zur Beschreibung von Erweiterungspunkten. Dreh und Angelpunkt ist hierbei die Klasse HotSpot. Ein HotSpot besteht aus einer Menge von HotSpotUnits, die beschreiben aus welchen Teilen (HotSpotUnit) eine Erweiterung zu einem bestimmten HotSpot besteht. Standardmäßig werden zwei Arten von HotSpotUnits unterschieden: CodingUnits stehen für programmiertechnische Einheiten, die eine Erweiterung bereitstellen muss und DeclarativeUnits stehen für deklarative Einheiten. Klassisches Beispiel für eine CodingUnit ist eine Klasse, die implementiert werden muss und die zusätzlich eine bestimmte Schnittstelle erfüllt. Eine DeclarativeUnit ist in vielen Frameworks zum Beispiel eine XML-Datei, in der notwendige Informationen einer Erweiterung deklarativ abgelegt werden. Es sind weitere Arten von HotSpotUnits denkbar, die in zukünftigen Versionen des FDMM integriert werden könnten. Denkbar ist zum Beispiel eine Einheit für Multimedia-Inhalte wie Bilder oder Musik.

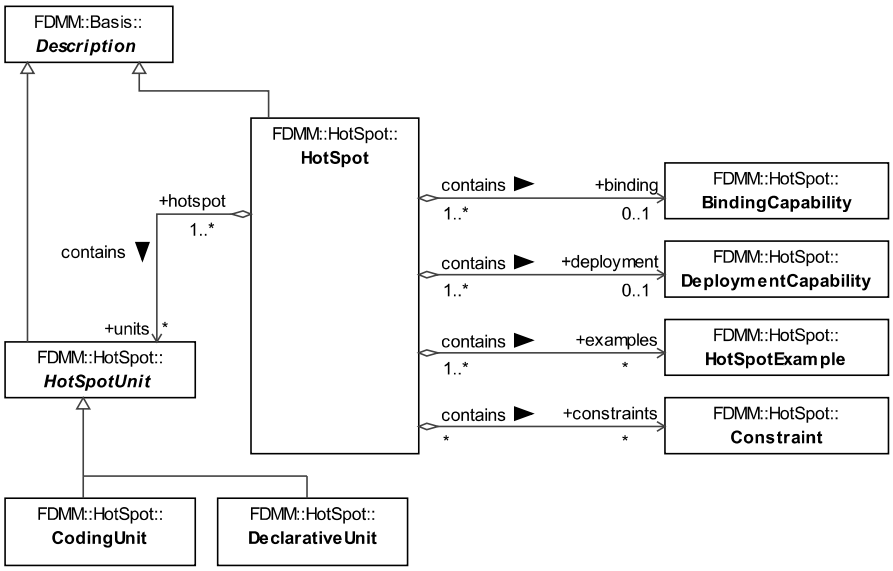


Abbildung 6.10: FDMM::HotSpot

Eine BindingCapability beschreibt, wie ein HotSpot an ein Framework gebunden wird und welche Schritte hierzu notwendig sind. Häufig handelt es sich dabei zum Beispiel um bestimmte Konfigurationsdateien, die entsprechende Einträge aufweisen müssen. Wir werden auf diesen Aspekt bei der Beschreibung der BindingDL in Abschnitt 6.2.7 näher eingehen.

Neben der BindingCapability beschreibt die DeploymentCapability wie ein HotSpot in der Laufzeitumgebung eines Frameworks zu installieren ist. Frameworks geben häufig bestimmte Verzeichnisse und Paketierungsformate, wie ZIP-Archive, vor. Diese Vorgaben müssen eingehalten werden, damit eine Erweiterung ins Framework eingebunden wird. Mehr Details werden in der Beschreibung der DeploymentDL in Abschnitt 6.2.9 gegeben.

Ein wichtiges Element in der Beschreibung von Frameworks sind anschauliche Beispiele, die zeigen, wie die beschriebenen Funktionen konkret eingesetzt werden. Hierzu dient das Element HotSpotExample und die dazugehörige SampleDL. Zu jedem HotSpot können Beispiele angegeben werden, die einem Benutzer verdeutlichen, wie ein HotSpot genutzt werden kann. Beispiele selbst können auf alle möglichen Elemente einer FDMM-basierten Framework-Beschreibung verweisen, so auch auf andere HotSpots. So ist es möglich ein ganzes Netz von Beispielen zu entwickeln, die alle Aspekte eines Frameworks abdecken. Näheres hierzu in der Beschreibung der SampleDL in Abschnitt 6.2.11.

Jeder HotSpot kann darüber hinaus bestimmten Einschränkungen unterliegen, die über die Klasse Constraint ausgedrückt werden. Eine Einschränkung lässt sich über die ConstraintDL definieren, die wir in Abschnitt 6.2.8 behandeln werden.

Eine CodingUnit, zu sehen in Abbildung 6.11, beschreibt einen Teil eines HotSpots, der auf programmatischem Wege realisiert wird, womit gemeint ist, dass hierfür Quellcode geschrieben werden muss. In einem objektorientierten Framework bedeutet dies, dass eine CodingUnit bestimmte Schnittstellen oder abstrakte Klassen definiert, die von einer Erweiterung implementiert werden müssen. Die zu implementierenden Methoden für eine CodingUnit werden als Hooks bezeichnet, die durch die Klasse HookCall repräsentiert werden. Ein HookCall ist eine vom Framework vorgegebene Methode, die eine Erweiterung implementieren muss. Da diese Methoden dem Framework bekannt sind, kann das Framework diese Methoden auf einer Erweiterung aufrufen, wodurch das „Inversion of control“-Prinzip eines Frameworks ermöglicht wird.

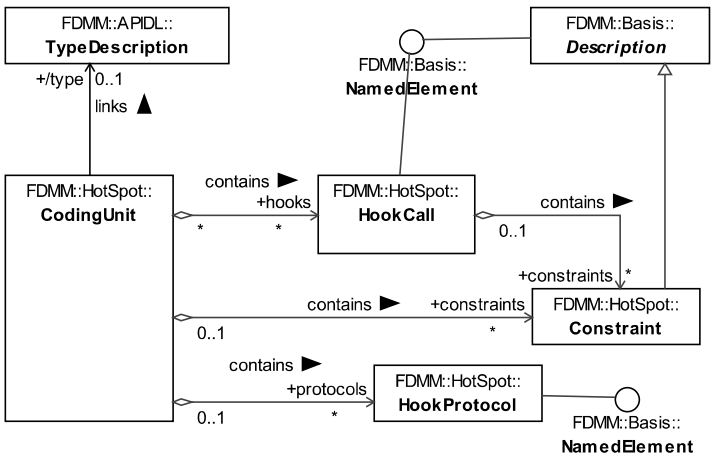


Abbildung 6.11: FDMM::HotSpot CodingUnit

Dieses Prinzip betont, dass eine Erweiterung nur dann aktiv werden kann, wenn diese vom Framework aufgerufen wird. Die Erweiterung wird in der Regel nicht von sich aus aktiv. Der Kontrollfluss wird vom Framework temporär an die Erweiterung übergeben und nach Abarbeitung durch die Erweiterung wieder an das Framework zurückgegeben. Die Parallele zum Ausspruch „Don´t call us, we call you“ hat diesem Verfahren auch den Beinamen „Hollywood-Prinzip“ eingebracht. Eine andere Bezeichnung hierfür sind „Callback-Funktionen“.

Wie für HotSpots können für einen HookCall zusätzliche Einschränkungen (Constraint) angegeben werden, die bei einer Implementierung zu beachten sind. Im Falle von HookCalls können sich diese Einschränkungen zum Beispiel auf die Parameter beziehen und so Wertebereiche einschränken. Weitere Möglichkeiten werden mit der Beschreibung der ConstraintDL in Abschnitt 6.2.8 diskutiert.

Weiteres Element einer CodingUnit sind die HookProtocols, die für Kommunikationsprotokolle zwischen einer Erweiterung und dem Framework stehen. Diese Protokolle werden mit Hilfe der bereits erwähnten HookCalls definiert, so dass ein Programmierer bei der Implementierung einer Erweiterung weiß, welche Methoden der Erweiterung wann und in welcher Reihenfolge aufgerufen werden. Die Protokolle werden mit Hilfe des Parameters ProtocolDL beschrieben, den wir in Abschnitt 6.2.10 näher betrachten. Da wir die ProtocolDL als Parameter definieren, ist es möglich, bereits existierende Protokollbeschreibungssprachen wiederzuverwenden.

6.2.6 FDMM::APIDL

Die APIDL stellt die Verbindung zwischen einer Framework-Beschreibung und einer häufig bereits vorhandenen API-Dokumentation her. Eine API-Dokumentation beschreibt in auflistender Weise die Klassen und Schnittstellen mit den dazugehörigen Methoden einer Software. Ziemlich jede etablierte Programmiersprache hat eine API-Dokumentationssprache. Dabei wird die Dokumentation häufig direkt aus den Kommentaren im Quellcode einer Software generiert. Für die Programmiersprache Java wird zum Beispiel das Werkzeug »JavaDoc« eingesetzt.

Die APIDL ist als Parameter definiert, da es naheliegend ist, eine bereits bestehende Sprache für die API-Dokumentation wiederzuverwenden. In Abbildung 6.12 spezifizieren wir die Anwendungsfälle für die Verwendung der APIDL.

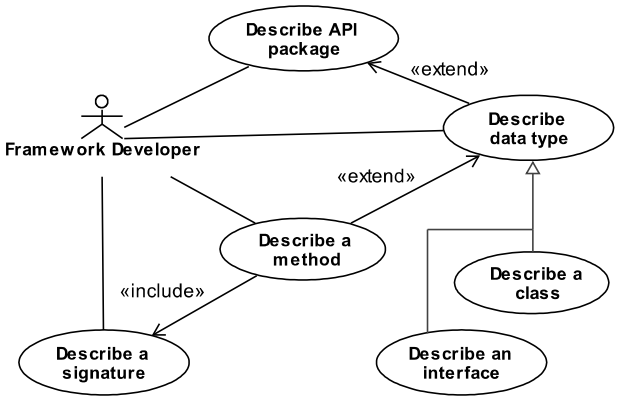


Abbildung 6.12: Anwendungsfälle der APIDL

Die Anwendungsfälle in Abbildung 6.12 beziehen sich auf die Beschreibung der API. Hierzu müssen Datentypen (data type) in Form von Klassen (class) und Schnittstellen (interface) beschrieben werden. Außerdem müssen Methoden (method) beschrieben und deren Signatur (signature) erfasst werden können.

Ausgehend von den Anwendungsfällen können wir das Meta-Modell der APIDL, zu sehen in Abbildung 6.13, ableiten.

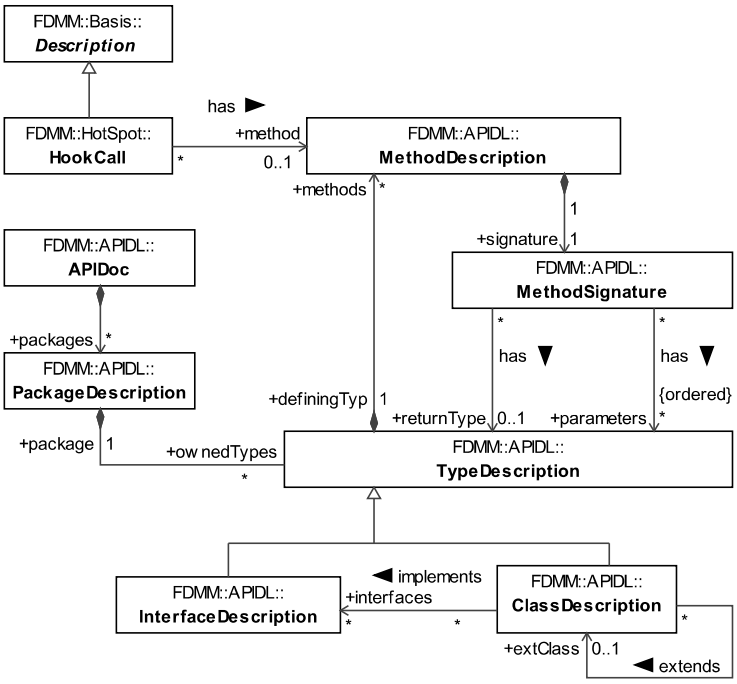


Abbildung 6.13: FDMM::APIDL

Eine API-Dokumentation wird demnach über die Klasse APIDoc aufgebaut, die auf eine Reihe von Paketbeschreibungen (PackageDescription) für Quellcodepakete verweist. Eine Paketbeschreibung enthält Beschreibungen für alle Ele-

mente des Paketes. Dies sind Beschreibungen von Typen (TypeDescription), also Klassen (ClassDescription) und Schnittstellen (InterfaceDescription).

Die Verbindung zwischen HookCalls aus dem HotSpot-Paket und einer APIDL wird über das Element MethodDescription hergestellt. Ein HookCall ist wie beschrieben eine Methode, die vom Framework für eine Erweiterung vorgegeben wird. Diese Methode ist somit auch in der API des Frameworks dokumentiert, so dass diese Informationen verbunden werden können. Methodenbeschreibungen (MethodDescription) gehören immer zu einer Typbeschreibung (TypeDescription) einer API-Dokumentation, wobei jede Methode eine Signatur (MethodSignature) besitzt.

Eine APIDL wird nicht nur in Verbindung mit HookCalls für HotSpots wiederverwendet, sondern auch durch die Meta-Modelle BindingDL (siehe Abschnitt 6.2.7) und SampleDL (siehe Abschnitt 6.2.11). Damit können die Informationen aus einer häufig bereits vorhandenen API-Dokumentation für eine vollständige Framework-Beschreibung mit den Konzepten der parametrisierten Meta-Modelle effizient wiederverwendet werden.

Das Meta-Modell der APIDL beschreibt die Struktur und Konzepte einer API-Dokumentation. Eine solche Dokumentation ist jedoch nicht ausführbar, so dass sie auch kein Verhalten aufweist. Aus diesem Grund verzichten wir für die APIDL auf die Definition von Verhaltensmustern. Daher kann für die APIDL kein semantischer Bindungstest durchgeführt werden. In diesem Fall beschränken wir uns auf die Bindung zwischen dem Meta-Modell der APIDL und dem Meta-Modell einer konkreten API-Dokumentationssprache. In Abschnitt 6.3.3 werden wir dies für die API-Dokumentationssprache »JavaDoc« im Detail betrachten.

6.2.7 FDMM::BindingDL

Eine Erweiterung muss an ein Framework gebunden werden, damit das Framework diese Erweiterung im Betrieb verwendet. Wie diese Bindung realisiert wird, kann mit der BindingDL beschrieben werden, deren Meta-Modell in Abbildung 6.14 zu sehen ist.

Die Beschreibung einer Bindung (BindingDescription) besteht aus einer Reihe von MetaEntries. Jedes MetaEntry steht für einen Teil einer BindingDescription, denn die Erstellung einer Bindung besteht häufig aus mehreren Konfigurationsschritten, die mit Hilfe der MetaEntries spezifiziert werden können. Hierzu enthält jedes MetaEntry zusätzlich eine Menge von Tasks. Jeder Task kann dabei auf einem Descriptor operieren. Ein Descriptor ist eine Konfigurationsdatei eines Frameworks. In einem MetaEntry können so Tasks für notwendige Änderungen an einem Descriptor beschrieben werden.

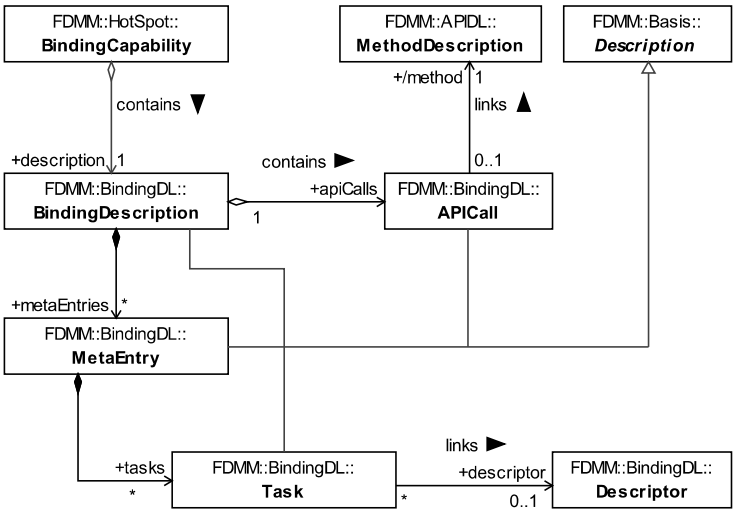


Abbildung 6.14: FDMM::BindingDL

Ein Descriptor wird häufig in Form einer XML-Konfigurationsdatei realisiert und hat in der Regel ein vorgegebenes Schema. Im Falle von XML-Dateien besitzen Descriptors zum Beispiel eine XML-Schema-Spezifikation oder eine DTD (engl. Document Type Definition). Für einen bestimmten HotSpot muss dieser Descriptor mit entsprechenden Informationen angereichert werden, was durch Tasks auf dem Descriptor geschieht. Durch diese Tasks wird beschrieben, welche Daten man zum Beispiel in den Descriptor an welcher Stelle eintragen muss oder welche zu verändern sind, damit die Bindung funktioniert.

Neben der Spezifikation der Bindung über MetaEntries in Konfigurationsdateien ist es auch möglich, auf die API des Frameworks zu verweisen. Es ist üblich, dass man Erweiterungen auch programmatisch im Framework bekannt machen kann. Ein klassisches Beispiel hierfür ist die Registrierung eines *Listeners*, der als Erweiterung des Frameworks auf bestimmte Ereignisse lauscht und dementsprechend in Aktion treten kann. Um solch einen Listener im Framework zu registrieren, muss der Framework-Benutzer meist eine bestimmte API-Methode (APICall) verwenden, die den neuen Listener im Framework registriert.

6.2.8 FDMM::ConstraintDL

Mit Hilfe der ConstraintDL können Einschränkungen für HotSpots oder auch HookCalls beschrieben werden. Einschränkungen können funktional und nichtfunktionaler Natur sein und helfen die geforderten Eigenschaften einer Erweiterung zu präzisieren. Die folgende Abbildung 6.15 zeigt das Meta-Modell der ConstraintDL.

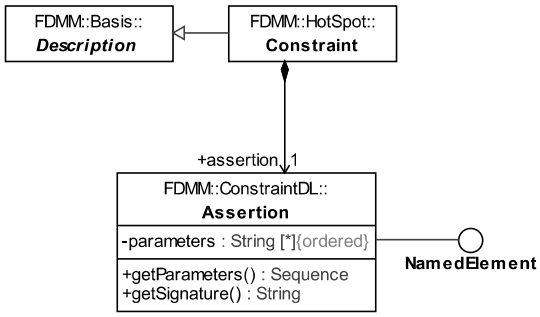


Abbildung 6.15: FDMM::ConstraintDL

Jedes Constraint besteht aus einer Assertion, die eine Zusicherung in Form einer Funktion mit Parametern beschreibt. Eine Assertion hat also einen Namen und eine Liste von Parametern. Die Semantik einer Assertion ergibt sich für den Framework-Benutzer aus dem Kontext und dem Namen der Assertion. Um zum Beispiel zu spezifizieren, dass ein Parameter `X` eines Hooks `calc(X)` nicht negativ ist, kann man folgende Assertion formulieren: `notNegative(X)`. Diese Assertion wird dann mit dem HookCall `calc` verlinkt, so dass die Verbindung von Hook und Einschränkung entsteht.

6.2.9 FDMM::DeploymentDL

Ein Deployment beschreibt im Falle des FDMM die Installation einer Erweiterung in der Laufzeitumgebung eines Frameworks. Wie in Abbildung 6.16 gezeigt, definiert es das Paketierungsformat und beschreibt die Installation.

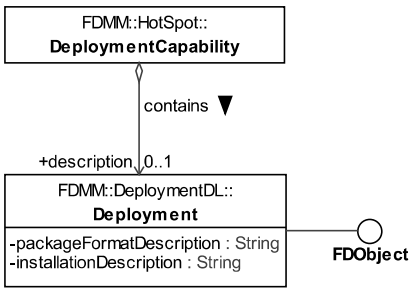


Abbildung 6.16: FDMM::DeploymentDL

Ein Deployment verfügt über eine Beschreibung des Paketierungsformates über das Attribut packageFormatDescription und eine Installationsanleitung über das Attribut installationDescription. Das FDMM beschränkt sich hier auf diese beiden Grundelemente, deren Inhalt nicht weiter festgelegt wird. Grund hierfür ist, dass es keine etablierten Standards für ein Deployment gibt, die hier sinnvoll vom FDMM unterstützt werden könnten.

6.2.10 FDMM::ProtocolDL

Ein Framework kommuniziert mit einer Erweiterung an einem Erweiterungspunkt durch den Aufruf der Hook-Methoden der Erweiterung. Je nach Erweiterungspunkt sind dabei eine Reihe von Hook-Methoden involviert. Der Framework-Entwickler muss beschreiben können wie genau das Kommunikationsprotokoll zwischen einem Framework und dessen Erweiterungen definiert ist. Das Protokoll definiert dabei die Aufruffolgen von Hook-Methoden einer Erweiterung durch das Framework.

Mit der ProtocolDL wollen wir diesen Sachverhalt beschreiben können. Da wir davon ausgehen, dass es für die Beschreibung von Protokollen bereits etablierte Sprachen gibt, die wiederverwendet werden können, definieren wir die ProtocolDL als Parameter des FDMM. Das Anwendungsfalldiagramm in Abbildung 6.17 zeigt die gestellten Anforderungen, aus denen wir die Sprachanforderung ableiten.

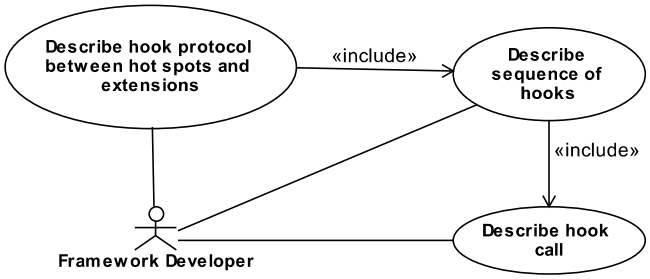


Abbildung 6.17: Anwendungsfälle der ProtocolDL

Die Anwendungsfälle in Abbildung 6.17 beschreiben den Fall, dass ein Kommunikationsprotokoll (hook protocol) zwischen Erweiterungspunkten (hot spots) und Erweiterungen (extensions) beschrieben werden können muss. Hierzu muss der Framework-Entwickler in der Lage sein Sequenzen von aufgerufenen Hook-Methoden (sequence of hooks) darzustellen und dabei die Hook-Aufrufe (hook call) auch einzeln beschreiben zu können.

Ein etabliertes Konzept zur Spezifikation von Protokollen ist ein Zustandsautomat. In jedem Kommunikationszustand werden dabei bestimmte Nachrichten versendet. Im Fall der ProtocolDL werden in unterschiedlichen Zuständen der Kommunikation die verschiedenen Hook-Methoden aufgerufen. Die Menge der Zustände und Hook-Methoden ergibt die möglichen Aufruffolgen von Hook-Methoden seitens des Frameworks.

Aus den Anwendungsfällen in Abbildung 6.17 leiten wir ein Meta-Modell zur Beschreibung der Struktur und der Konzepte ab. Bei Beschreibung der CodingUnit eines HotSpots in Abschnitt 6.2.5 haben wir mit Abbildung 6.11 bereits erörtert, dass mit einer CodingUnit auch ein Aufrufprotokoll (HookProtocol) assoziiert ist. Ausgehend von der Klasse HookProtocol integrieren wir das Meta-Modell der ProtocolDL und zeigen das Resultat in Abbildung 6.18.

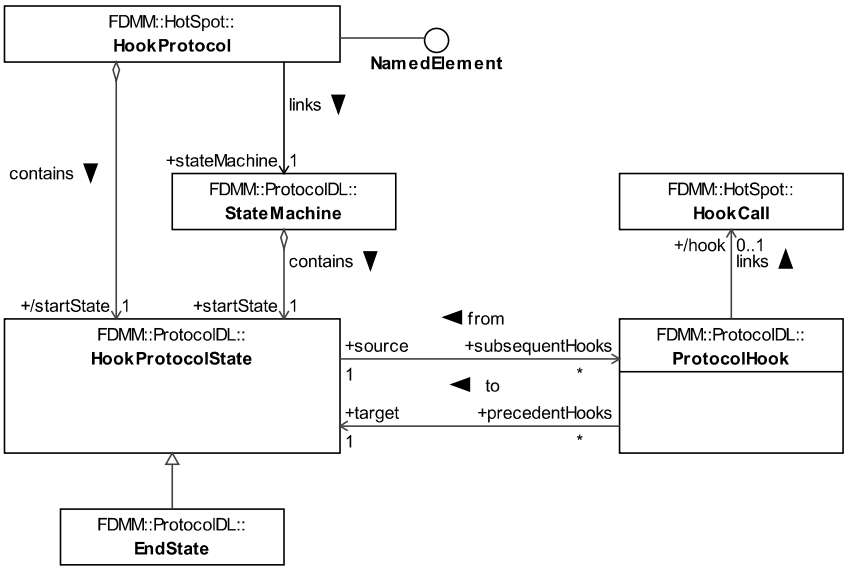


Abbildung 6.18: FDMM::ProtocolDL

Die Integration entsteht, indem die Klasse HookProtocol des HotSpot-Paketes auf eine StateMachine verweist. Eine StateMachine steht für das Konzept eines Zustandsautomaten. Ein solcher Zustandsautomat zur Definition der Aufrufreihenfolge von HookCalls verfügt über eine Reihe von Protokollzuständen (HookProtocolState). Innerhalb eines jeden Zustands können unterschiedliche HookCalls zu einem Zustandsübergang (ProtocolHook) führen, so dass sich die zugehörige CodingUnit anschließend in einem neuen Zustand befindet. Das Protokollende ist durch Erreichung des EndState definiert. Die Menge der Zustände und möglicher ProtocolHooks, die jeweils einem HookCall zugeordnet sind, ermöglicht die Beschreibung des Kommunikationsprotokolls zwischen Framework und Erweiterung.

Das Verhalten einer StateMachine ist geprägt durch die Abarbeitung von Zuständen und das Erreichen von Folgezuständen über ProtocolHooks. Die Ausführung endet bei Erreichen des EndState. Wir definieren hierfür einige Verhaltensmuster in unserer Sprachanforderung und beginnen in Abbildung 6.19 mit einem Muster, das den Startzustand festlegt.

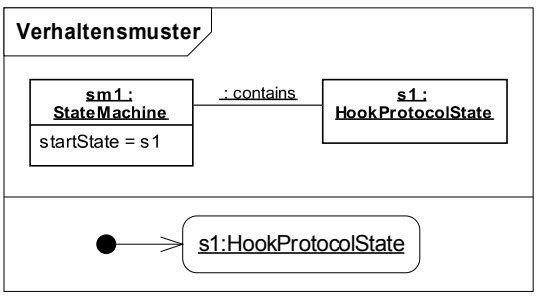


Abbildung 6.19: Verhaltensmuster für startState der ProtocolDL

Der Startzustand ist definiert durch den Zustand, der als startState mit einem StateMachine-Objekt verlinkt ist. Im nächsten Verhaltensmuster in Abbildung 6.20 definieren wir den Übergang von einem Zustand zum nächsten.

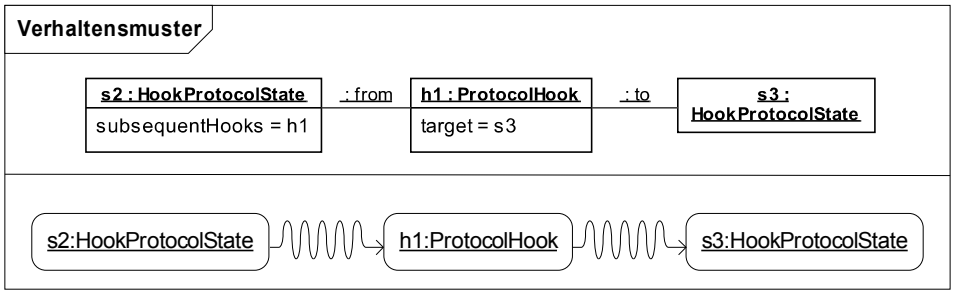


Abbildung 6.20: Verhaltensmuster für Zustandsübergänge der ProtocolDL

Ein Zustandsübergang erfolgt an einem ProtocolHook, der wiederum mit einem HookCall verlinkt ist. Der Übergang erfolgt von einem HookProtocolState definiert über den Link from zu einem HookProtocolState definiert über to. Von einem Zustand aus kann es auch mehrere mögliche Übergänge geben. Dieses Verhalten betrachten wir in Abbildung 6.21.

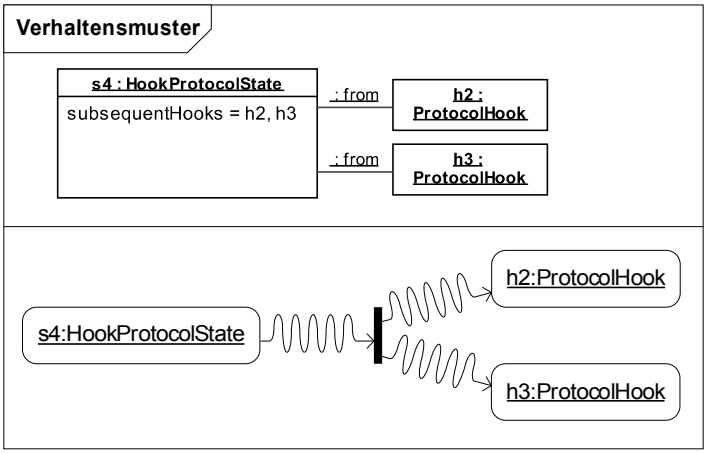


Abbildung 6.21: Verhaltensmuster für multiple Zustandsübergänge der ProtocolDL

Treten mehrere mögliche Zustandsübergänge an einem HookProtocolState auf, werden alle Übergänge verschränkt weiterverfolgt. Die Reihenfolge der Übergänge ist nicht festgelegt. Das Prozessmuster in Abbildung 6.21 legt somit fest, dass beide ausgehenden Zustandsübergänge erreicht werden müssen.

Das abschließende Verhaltensmuster in Abbildung 6.22 legt schließlich das Ende einer Protokollausführung fest. Das Ende ist erreicht, wenn der EndState über einen ProtocolHook erreicht wird.

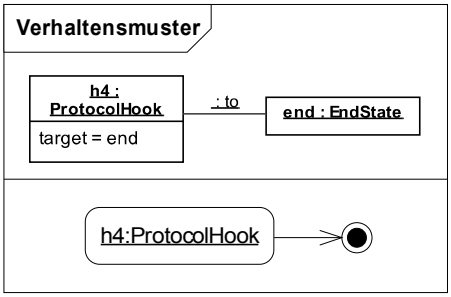


Abbildung 6.22: Verhaltensmuster für den Endzustand der ProtocolDL

Es ist naheliegend für die ProtocolDL eine Sprache wiederzuverwenden, die ebenfalls einen Zustandsautomaten beschreiben kann. Wir werden dies in Abschnitt 6.3.2 erörtern.

6.2.11 FDMM::SampleDL

Das Paket SampleDL enthält alle Elemente, um Beispiele für die Verwendung eines Frameworks zu dokumentieren. Dies kann zum Beispiel in Form von Tutorien geschehen. Hierzu verwendet die SampleDL Verweise auf die verschiedenen Elemente einer Framework-Beschreibung, wie sie durch das FDMM spezifiziert werden. Um diese Verweise spezifizieren zu können verwendet das Paket SampleDL selbst wiederum andere Pakete des FDMM als Parameter. Abbildung 6.23 zeigt die Paketstruktur in Bezug auf das SampleDL-Paket.

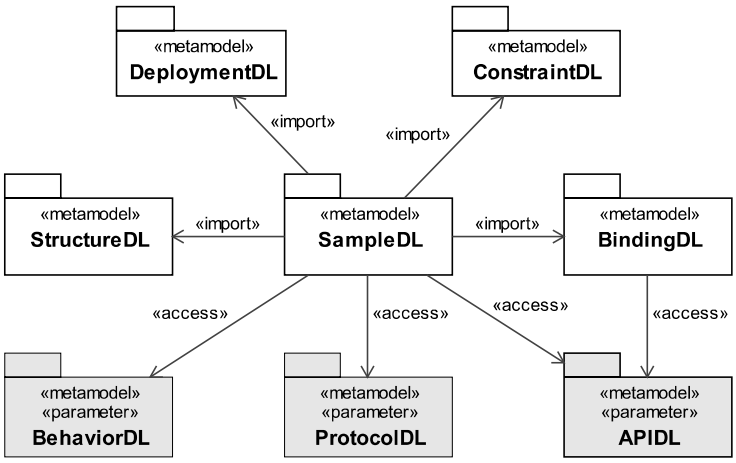


Abbildung 6.23: Paketstruktur der SampleDL

Das Paket SampleDL in Abbildung 6.23 verwendet Klassen aus allen anderen Paketen des FDMM. So können Beispiele mit den vorhandenen Informationen verknüpft werden. Ein Beispiel in der SampleDL kann so etwa direkt auf Elemente der Architektur aus der StructureDL verlinken.

Die Elemente des SampleDL-Paketes beschreiben ein Hyper-Dokument, das die Beschreibung eines Beispiels enthält und dabei alle verfügbaren Informa-

tionen der Framework-Beschreibung verlinken kann. Am einfachsten stellt man sich das Hyper-Dokument als eine Web-Seite vor, die beschreibende Texte und Verweise zu relevanten Bereichen in der Framework-Beschreibung enthält.

Im Meta-Modell der SampleDL, zu sehen in Abbildung 6.24, erkennen wir, dass die Beschreibung eines Beispiels als Hyper-Dokument (HyperDocument) aus einer Menge von DocElementen aufgebaut wird, die unterschiedliche Ausprägungen haben können. Elementar sind ChapterElemente, ParagraphElemente und LinkElement, die dazu dienen das HyperDocument zu strukturieren. Das allgemeine LinkElement kann dazu verwendet werden, um auf beliebige URLs des WWW zu verweisen.

Alle weiteren Elemente der SampleDL dienen zur Verbindung mit Informationen aus der Framework-Beschreibung. Diese Elemente sind das ArchitecturalViewElement mit Verbindung zur ArchitectureDL, das HookElement mit Verbindung zur APIDL und so weiter. Eine Übersicht über alle möglichen Verbindungen gibt Abbildung 6.24.

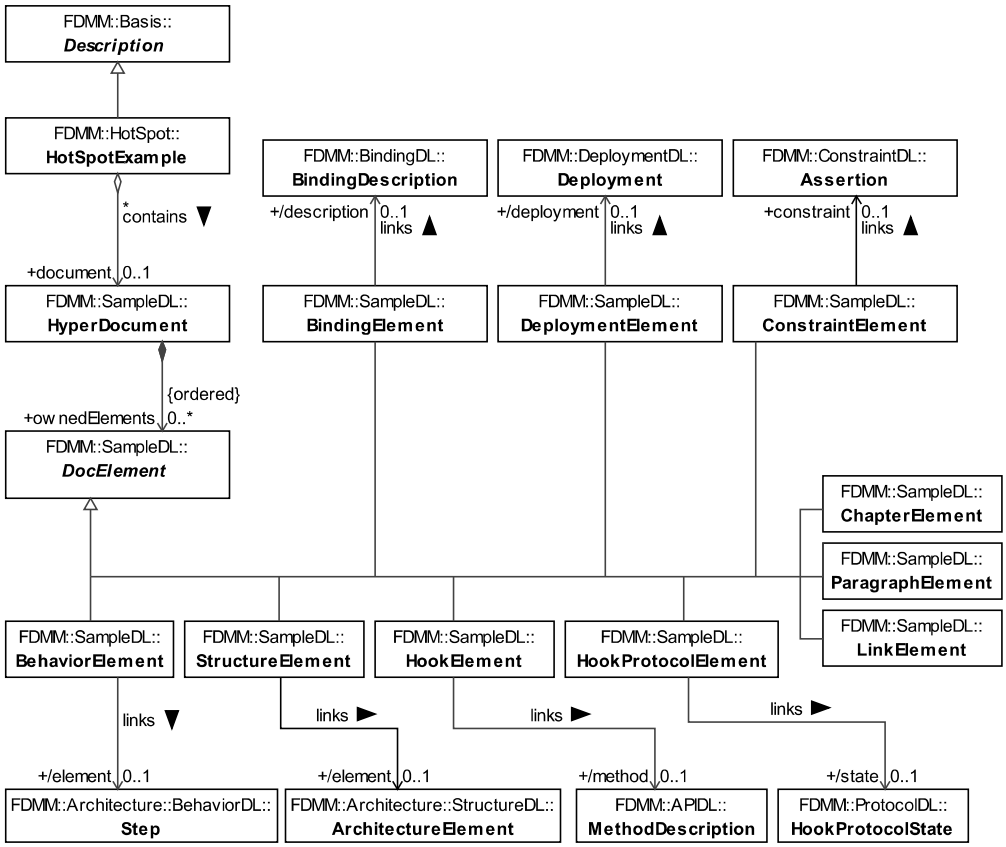


Abbildung 6.24: FDMM::SampleDL

Die Modellierung als Hyper-Dokument ermöglicht es Beispiele in Form von Web-Seiten bereitzustellen und alle bereits vorhandenen Informationen der Framework-Beschreibung wiederzuverwenden. Eine gute Sammlung von Beispielen zu einem Framework deckt so die gesamte Beschreibung eines Frameworks ab.

Mit der SampleDL haben wir das FDMM vollständig beschrieben. In nachfolgenden Abschnitt 6.3 gehen wir nun auf die Bindung der drei Parameter ein, dir wir im FDMM definiert haben.

6.3 Parameterbindung

Das FDMM verfügt über die drei Parameter BehaviorDL, ProtocolDL und APIDL. In diesem Abschnitt wollen wir diese drei Parameter an konkrete Sprachen binden und die semantischen Bindungstests durchführen, so dass wir ein FDMM erhalten, das wir für konkrete Framework-Beschreibungen einsetzen können.

In Abbildung 6.25 zeigen wir das Paketdiagramm mit der gewünschten Bindung. Wir sehen, dass UML::Activity, für UML-Aktivitätendiagramme, an den Parameter BehaviorDL gebunden werden soll. Des weiteren wollen wir UML:PSC, für »UML Protocol State Charts«, an die ProtocolDL binden und JEL::Doclet, worin sich das JavaDoc-Format zur Beschreibung einer API in der Programmiersprache Java verbirgt, an die APIDL.

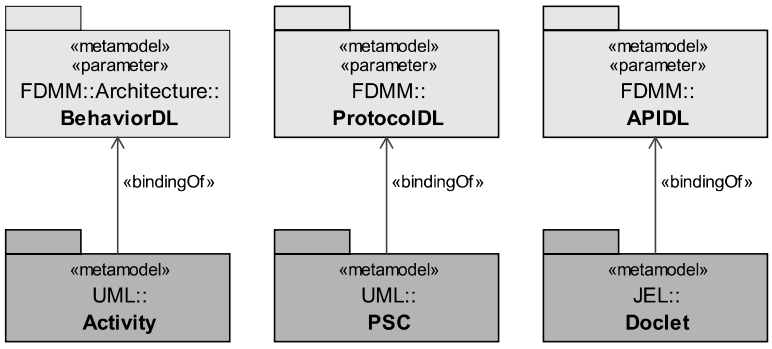


Abbildung 6.25: FDMM Pakete mit gebundenen Parametern

Zur Wiederverwendung der konkreten Sprachen spezifizieren wir für jeden Parameter eine Bindung. Hierzu betrachten wir die Meta-Modelle der Parameter und definieren die Bindung mit Hilfe der Bindungs-Syntax aus Kapitel 5.

Für die beiden Parameter BehaviorDL und ProtocolDL führen wir außerdem den semantischen Bindungstest durch. Wir haben das Konzept eines semantischen Bindungstest in Abschnitt 5.4 eingeführt. Da wir in dieser Arbeit nur das Konzept, aber keine konkrete Implementierung präsentieren, veranschaulichen wir den Bindungstest an einigen Beispielen. Der Parameter APIDL definiert kein Verhalten, so dass für diesen kein semantischer Bindungstest durchgeführt wird.

6.3.1 Bindung der BehaviorDL

Als Bindung an den Parameter der BehaviorDL wollen wir UML-Aktivitätendiagramme wiederverwenden. Wir haben die Verwendung von UML-Aktivitä-

tendiagrammen für die BehaviorDL bereits bei Einführung des Ansatzes einer anforderungsbasierten Wiederverwendung von Sprachen erläutert.

So haben wir den entsprechenden Auszug aus dem UML Meta-Modell [UMLSUP23] für UML-Aktivitätendiagramme, bezeichnet als UML::Activity, das die relevanten Klassen für die Modellierung von UML-Aktivitätendiagrammen enthält, bereits in Abbildung 5.27 des Abschnitts 5.3.2 auf Seite 147 gezeigt und erläutert. Wir verzichten daher an dieser Stelle auf eine wiederholende Darstellung.

Für die Bindung wird das Meta-Modell der BehaviorDL auf das UML::Activity Meta-Modell abgebildet. Auch diesen Schritt haben in Abschnitt 5.3.2 bereits aufgegriffen und eine Bindung in Listing 5.5 auf Seite 148 definiert.

Semantischer Bindungstest

Der semantische Bindungstest prüft die Korrektheit der Bindung. Den semantischen Bindungstest für die BehaviorDL haben wir in Abschnitt 5.4.2 bereits beispielhaft vorgestellt. Der Test ergab, dass die Bindung definiert in Listing 5.5 valide ist und somit die BehaviorDL durch UML::Activity ersetzt werden kann. Wir wollen daher auf weitere Testbeispiele für die BehaviorDL verzichten und widmen uns stattdessen im nächsten Abschnitt der Bindung der ProtocolDL.

6.3.2 Bindung der ProtocolDL

Wir wollen UML-Protokollzustandsdiagramme an den Parameter ProtocolDL binden. Hierzu betrachten wir das Meta-Modell für UML-Protokollzustandsdiagramme, kurz UML::PSC, in Abbildung 6.26.

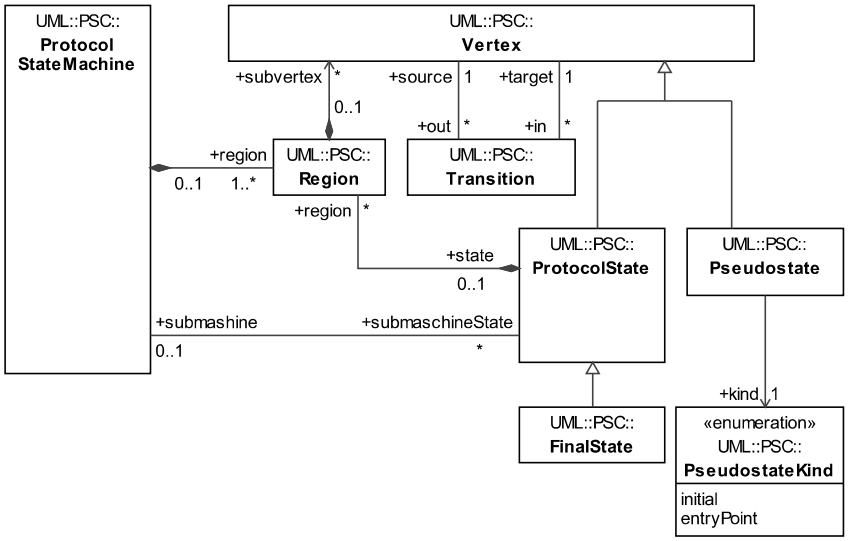


Abbildung 6.26: UML::PSC Meta-Modell

Das UML::PSC Meta-Modell in Abbildung 6.26 beschreibt einen Zustandsautomaten (ProtocolStateMachine) dessen Zustände (ProtocolState) spezielle Vertex

Objekte sind. Zwischen den Zuständen gibt es Übergänge (Transition), die zwei Zustände miteinander verbinden und den Übergang von dem einen in den anderen Zustand modellieren. Zur Modellierung spezieller Zustände wie einem initialen Anfangszustand (PseudostateKind.initial) oder dem Endzustand (FinalState) definiert das Meta-Modell entsprechende spezialisierte Klassen, die von Vertex bzw. ProtocolState erben.

Zur Definition der Bindung müssen alle Elemente der ProtocolDL, wie sie in Abbildung 6.18 definiert sind, auf das UML::PSC Meta-Modell abgebildet werden. Diese Bindung definieren wir in Listing 6.1.

Listing 6.1: Bindung zwischen ProtocolDL und UML::PSC

```
1  {
2     (StateMachine, ProtocolStateMachine),
3     (StateMachine.startState, [ProtocolStateMachine.region,
4                                Region.subvertex]
5          <context ProtocolStateMachine
6             self(region)
7             ->first()
8             ->collect(subvertex)
9             ->select(v:Vertex | v.oclIsTypeOf(Pseudostate))
10            ->collet(v:Vertex | v.oclAsType(Pseudostate))
11            ->select(p: Pseudostate |
12                        p.kind = PseudostateKind.initial )> ),
13    (HookProtocolState, Vertex),
14    (HookProtocolState.subsequentHooks, Vertex.out),
15    (ProtocolHook, Transition),
16    (ProtocolHook.target, Transition.target),
17    (EndState, FinalState)
18 }
```

In Zeile 2 der Bindung bilden wir die ProtocolDL StateMachine auf die ProtocolStateMachine ab. Der interessanteste Teil ist die Abbildung der startState Assoziation zwischen StateMachine und HookProtocolState. Diese Assoziation bilden wir auf eine Assoziationskette in UML::PSC ab. Diese Kette startet bei ProtocolStateMachine, wählt die erste verlinkte Region und dann das Vertex Objekt vom Typ Pseudostate von der Art (kind) initial. Diese Abbildung mit entsprechender OCL Einschränkung sehen wir in den Zeilen 3 bis 12. Die weitere Bindung in den nachfolgenden Zeilen ist wieder eine einfache Abbildung der Klassen und Assoziationen.

Semantischer Bindungstest

Wir wollen den semantischen Bindungstest für UML::PSC und die ProtocolDL analog zum semantischen Bindungstest der BehahiorDL durchführen. Hierzu verfolgen wir das Vorgehen in vier Schritten aus Abbildung 5.37.

Als Beispiel betrachten wir das Verhaltensmuster in Abbildung 6.20 auf Seite 198. Auf Basis der Bindung in Listing 6.1 können wir in Schritt 1 dieses Muster in ein Muster ausgedrückt über UML::PSC übersetzen. Das Ergebnis sehen wir in Abbildung 6.27.

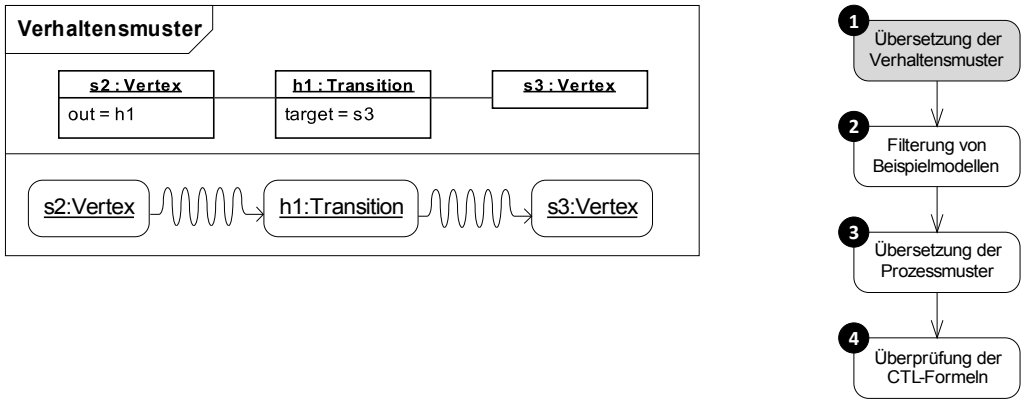


Abbildung 6.27: Verhaltensmuster für Zustandsübergänge in UML::PSC

Zur weiteren Durchführung des semantischen Bindungstest müssen wir vorhandene Beispielmodelle für UML::PSC herausfiltern, die dem Instanzmuster in Abbildung 6.27 entsprechen. Ein solches Beispiel sehen wir in Abbildung 6.28. Das Beispiel besteht aus einer einfachen Abfolge von zwei Zuständen ps1 und ps2.

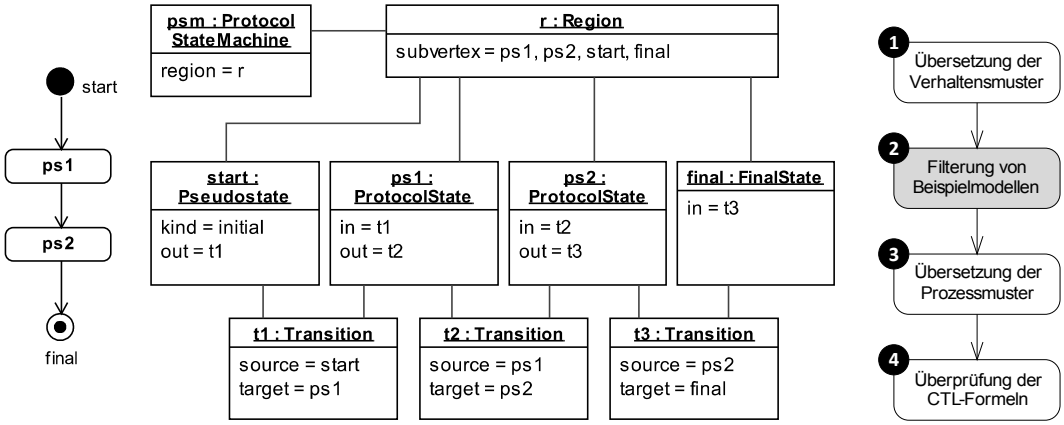


Abbildung 6.28: Beispiel eines UML-Protokollzustandsdiagramms in konkreter und abstrakter Syntax

Durch Anwendung des Instanzmusters aus dem Verhaltensmuster können wir das Prozessmuster nun auf das konkrete Beispiel anwenden und, wie in Abbildung 6.29 zu sehen, entsprechend überführen.

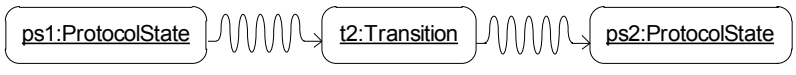
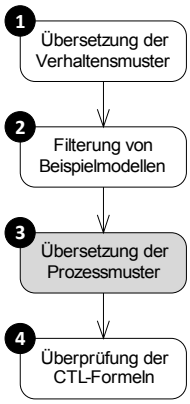


Abbildung 6.29: Prozessmuster über konkretem Beispiel eines UML-Protokollzustandsdiagramms



Das Prozessmuster in Abbildung 6.29 verwendet die Klasse ProtocolState anstelle von Vertex, wie noch im Prozessmuster in Abbildung 6.27. Dies ist möglich und gültig, da ProtocolState eine Unterklasse von Vertex ist. Das Prozessmuster über dem konkreten Beispiel müssen wir nun in ein Prozessmuster über DMM-Regeln überführen. Als Verhaltensspezifikation für die verwendeten UML-Protokollzustandsdiagramme verwenden wir hier die DMM-Spezifikation für UML-Zustandsdiagramme aus [Nesterow2010].

Die DMM-Spezifikation in [Nesterow2010] definiert die DMM-Regeln executeState() für die Ausführung eines Zustandes und fireTransition() für die Ausführung eines Zustandsübergangs. Mit diesen Angaben können wir in Abbildung 6.30 das Prozessmuster des Beispiels nun über DMM-Regeln formulieren.

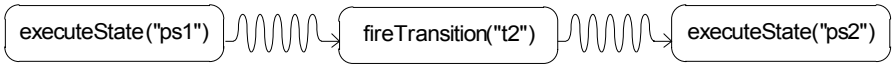
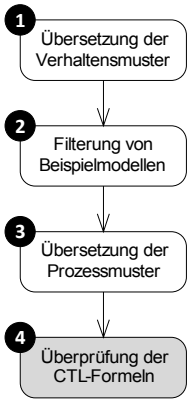


Abbildung 6.30: Prozessmuster über DMM-Regeln des konkreten Beispiels eines UML-Protokollzustandsdiagramms



Nach [Khaluf2010] können wir das Prozessmuster in Abbildung 6.30 abschließend in eine CTL-Formel übersetzen, die wir auf dem Gesamtverhalten des Beispielmodells per Model-Checking überprüfen können.

Die Auswertung dieser CTL-Formel im Gesamtverhalten ergibt, dass sich das erwartete Verhalten in diesem Fall mit dem tatsächlichen Verhalten deckt. Damit haben wir einen bestätigenden Test identifiziert, der belegt, dass die Bindung von UML::PSC an die ProtocolDL korrekt ist. Mittels weiterer Tests kann die Korrektheit der Bindung weiter erhärtet werden.

Die resultierende CTL-Formel lautet:

$$\boldsymbol{EF}\left(executeStep('ps1') \wedge \boldsymbol{AF}\left(fireTransition('t2') \wedge \boldsymbol{AF}\left(executeStep('ps2')\right)\right)\right)$$

Da wir das Mittel des semantischen Bindungstest in dieser Arbeit nur konzeptionell eingeführt haben, verzichten wir an dieser Stelle auf weitere Beispiele. Das Vorgehen ist für alle weiteren Tests analog.

6.3.3 Bindung der APIDL

Wir wollen die APIDL durch eine existierende Sprache ersetzen, die der Sprache entspricht, die das JavaDoc-Werkzeug zur Generierung der API-Dokumentation aus den Kommentaren innerhalb des Quellcodes von Java-Programmen verwendet. Standardmäßig generiert das JavaDoc-Werkzeug eine Dokumentation im HTML-Format. Dieses Format ist nicht direkt geeignet, um als Sprachdefinition zu dienen. Das JavaDoc-Werkzeug ermöglicht es jedoch

benutzerdefinierte »Doclets« zu definieren, um die Inhalte in anderen Formaten ausgeben zu lassen. So ist es möglich die Inhalte in der »Java Exporting Language« (JEL) in Form einer XML-Datei ausgeben zu lassen. Das verwendete JEL::Doclet Meta-Modell sehen wir in Abbildung 6.31.

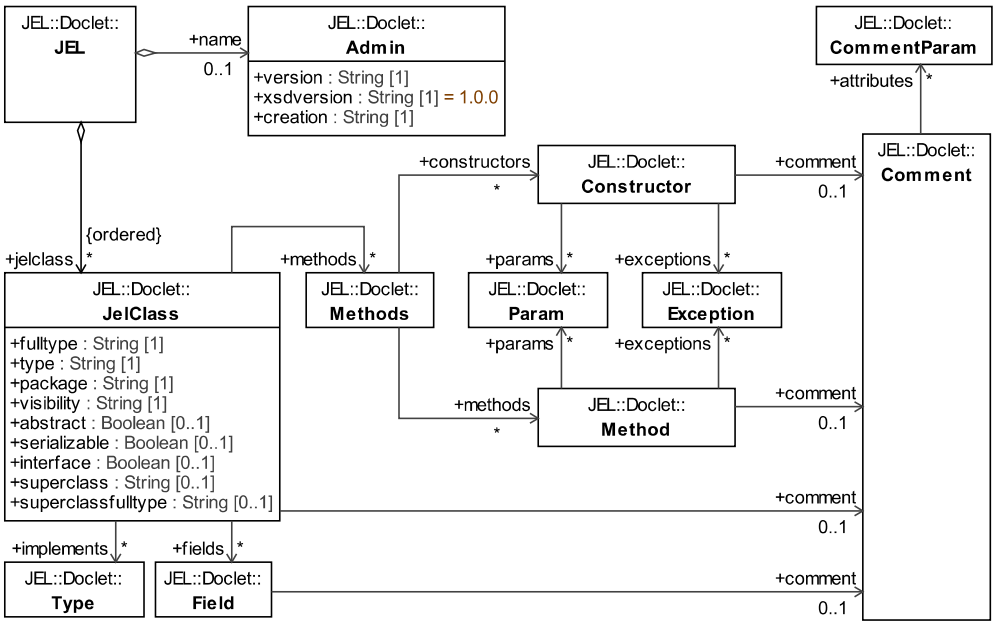


Abbildung 6.31: JEL::Doclet Meta-Modell

Mit Hilfe des JEL::Doclet Meta-Modells kann man Java-Klassen (JelClass) und ihre zugehörigen Methoden (Methods), deren Parameter (Param) und Ausnahmen (Exception) als Kommentare (Comment) dokumentieren.

Die Bindung zwischen der APIDL und JEL::Doclet definieren wir In Listing 6.2. Die Bindung ist relativ komplex, da an vielen Stellen die Bindung der Klassen und Assoziationen nur durch zusätzliche OCL-Einschränkungen möglich ist. Diese Notwendigkeit entsteht, da einige Elemente, die in der APIDL durch einzelne Klassen repräsentiert sind, in JEL::Doclet durch Klassen mit entsprechenden Attributen definiert sind. Es ist zwar möglich Attribute abzubilden, indem wir sie als Assoziationen auffassen, aber häufig ist die Bindung abhängig von einem konkreten Wert eines Attributes. In solchen Fällen können wir die Bindung nur durch den Einsatz von OCL definieren.

Ein Beispiel hierfür ist die Definition von Schnittstellen in der APIDL durch die Klasse InterfaceDescription. In JEL::Doclet ist eine Schnittstelle eine JelClass mit dem Attribut interface belegt mit true. Wir bilden hierzu, wie in Zeile 30 zu sehen, die InterfaceDescription auf JelClass ab und selektieren dann mit einer OCL-Einschränkung genau die JelClass, bei denen das Attribut interface entsprechend belegt ist. Dieses Schema setzen wir in Listing 6.2 wiederholt ein, um die Bindung zu definieren.

Wie bereits erörtert definiert die APIDL kein Verhalten, so dass kein semantischer Bindungstest durchgeführt werden kann. Da es keine Ausführung der

Sprache gibt, fehlt die semantische Ebene, auf der man die Bindung prüfen könnte.

Listing 6.2: Bindung zwischen APIDL und JEL::Doclet

```
1  {
2     (APIDoc, JEL),
3     (APIDoc.packages, JEL.jelclass
4       <context JEL
5          self.jelclass.package>),
6     (PackageDescription, JelClass)
7     (PackageDescription.ownedTypes, JelClass.package
8       <context JEL
9          self.jelclass
10         ->collect(c : JelClass | c.package = JelClass.package)
   >)
11    (TypeDescription, JelClass),
12    (TypeDescription.package, JelClass.package
13      <context JEL
14         self.jelclass
15         ->collect(c : JelClass |
16                      c.package = JelClass.package) >),
17    (TypeDescription.methods, [JelClass.methods,
   Methods.methods]
18      <context JelClass
19         self.methods.methods.first() >),
20    (MethodDescription, Method),
21    (MethodDescription.signature, Method.self),
22    (MethodSignature, Method),
23    (MethodSignature.returnType, Method.fulltype
24      <context JEL
25         self.jelclass
26         -> collect(c : JelClass |
27                      c.fulltype = Method.fulltype) >),
28    (MethodSignature.parameters, [Method.params, Param.fulltype]
29      <context JEL
30         self.jelclass
31         -> collect(c : JelClass |
32                      c.fulltype = Param.fulltype) >),
33    (InterfaceDescription, JelClass
34      <context JelClass
35         self.interface = true >),
36    (ClassDescription, JelClass
37      <context JelClass
38         self.interface = false >),
39    (ClassDescription.interfaces, JelClass.implements
40      <context JEL
41         self.jelclass
42         ->collect(c : JelClass |
43                      c.type = JelClass.implements.type),
44      (ClassDescription.extClass, JelClass.superclass
45      <context JEL
46         self.jelclass
47         -> collect(c : JelClass |
48                      c.type = JelClass.superclass) >),
49    (
50 }
```

Nachdem wir das FDMM vorgestellt und dessen Parameter an konkrete Sprachen gebunden haben, widmen wir uns nun einigen Fallstudien zur Framework-Beschreibung zu. In diesen Fallstudien wollen wir zeigen, dass wir

das FDMM zur Beschreibung real existierender Frameworks erfolgreich einsetzen können.

6.4 Fallstudien zur Framework-Beschreibung

Das FDMM wurde im Rahmen dieser Arbeit zur beispielhaften Dokumentation einiger Frameworks in Form von Fallstudien eingesetzt. Zur Erstellung der Beschreibungen wurde das in dieser Arbeit entwickelte Werkzeug »Companian« verwendet, das wir im nachfolgenden Kapitel 7 detailliert vorstellen werden. Companian bietet mit dem Companian-Editor eine Implementierung des FDMM und damit alle Möglichkeiten, die Framework-Beschreibung zu einem Framework als »Framework Description Models« (FDM) zu erfassen.

Wie bereits erwähnt, haben wir keine konkrete Syntax für das FDMM definiert. Stattdessen geben wir mit dem Companian-Editor eine Referenzimplementierung an. In Ermangelung einer konkreten Syntax zeigen wir im Rahmen dieser Arbeit keine vollständige Dokumentation eines konkreten Frameworks. Stattdessen beschränken wir uns auf Bildschirmfotos, die einen Eindruck über die Arbeit mit dem Companian-Editor vermitteln. In Abbildung 6.32 zeigen wir ein Bildschirmfoto bei Erstellung der Framework-Beschreibung des »Java Servlet Frameworks« (servlet-api) in Version 2.3.

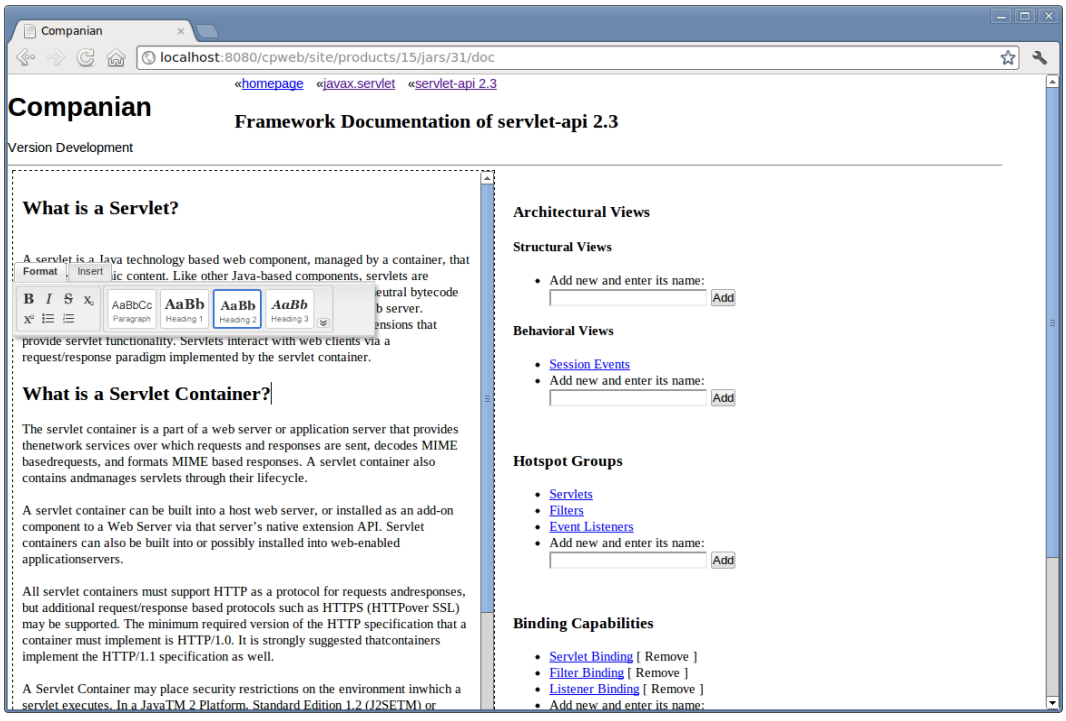


Abbildung 6.32: Companian-Editor zur Erstellung von Framework-Beschreibungen

Im linken Teil des Editors können beschreibende Texte, wie sie für viele Elemente des FDMM vorgesehen sind, erfasst werden. Im rechten Teil können

Informationen gemäß des FDMM hinzugefügt werden. Es können zum Beispiel verschiedene Sichten auf die Architektur über »Architectural Views« beschrieben oder Gruppen für Erweiterungspunkte als »Hotspot Groups« angelegt werden. Nach Klick auf den Link einer Gruppe gelangt man zu dessen Editierfunktionen und kann Erweiterungspunkte beschreiben. So wird das gesamte FDMM vom Editor abgebildet.

Das FDMM verwendet existierende Sprachen für einige Aspekte. So werden zum Beispiel die Kommunikationsprotokolle zwischen Framework und Erweiterungen, genannt Hook-Protokolle und definiert in der ProtocolDL durch UML-Protokollzustandsautomaten realisiert. Solche Automaten können mit einem UML-Modellierungswerkzeug wie »MagicDraw« erstellt, von dort exportiert und im Companian-Editor importiert werden. Durch die Bindung der ProtocolDL an UML::PSC stehen hierzu alle benötigten Informationen zur Verfügung.

In Abbildung 6.33 zeigen wir die Bildschirmfotos eines Hook-Protokolls im Companian-Editor und dessen Modellierung als UML-Protokollzustandsautomaten in MagicDraw. Im Bildschirmfoto stellt Companian die ProtocolDL in einer textuellen Repräsentation dar. Die textuelle Repräsentation in ProtocolDL entspricht dabei dem UML-Protokollzustandsautomaten wie er in MagicDraw im rechten Teil der Abbildung erstellt wurde.

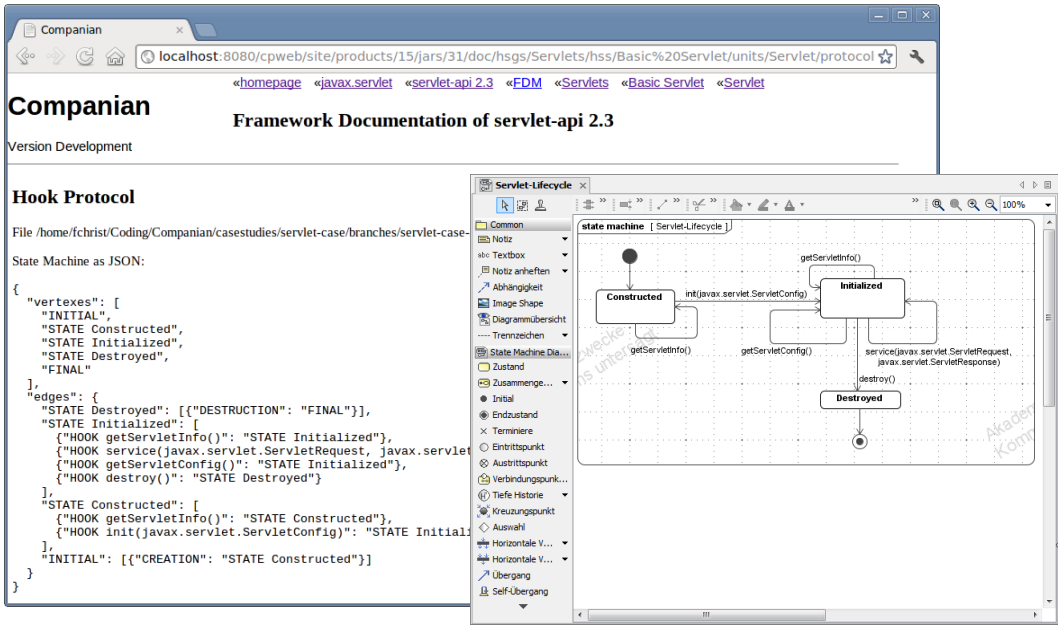


Abbildung 6.33: Hook-Protokoll in Companian (links) und MagicDraw (rechts)

Auf Grundlage des parametrisierten FDMM werden so Sprachen und Werkzeuge für den Einsatz in Companian wiederverwendet. Analog können im Companian-Editor UML-Aktivitätendiagramme für die BehaviorDL eingebunden werden, die zuvor in MagicDraw modelliert wurden.

Im Rahmen der Fallstudien wurde der Companian-Editor verwendet, um beispielhaft für die folgenden Frameworks eine Framework-Beschreibung als FDM anzulegen:

- *Java Servlet Framework*[2]
  Framework zur Behandlung von HTTP-Anfragen als Grundlage für die Entwicklung von Web-Anwendungen.
  Versionen 2.3, 2.4, und 3.0

- *Java Server Faces Framework*[3]
  Framework zur Erstellung der GUI einer Web-Anwendung.
  Versionen 1.2.0 und 2.0.0

- *Java Logging Framework*[4]
  Framework zur Erzeugung von Log-Meldungen in einer Anwendung.
  Versionen 1.2 und 2.0

- *Google Android Framework*[5]
  Framework
  Version 2.2

Jedes dieser Frameworks im Detail vorzustellen, wäre an dieser Stelle nicht zielführend. In Abschnitt 2.1.3 auf Seite 26 haben wir das Servlet Framework etwas ausführlicher eingeführt. Daher fokussieren wir uns im praktischen Teil dieser Arbeit auf dieses Framework.

Die ausgewählten Frameworks sind in der Programmiersprache Java implementiert. Die ersten drei Frameworks der Liste sind etablierte Frameworks zur Erstellung betrieblicher Informationssysteme als Web-Anwendungen. Diese Frameworks wurden ausgewählt, da sie in industriellen Forschungskooperationen eingesetzt wurden, so dass hier praktische Erfahrungen mit den Frameworks einfließen konnten. Nicht zuletzt entstammt die in dieser Arbeit betrachtete Problemstellung den Erfahrungen aus eben diesen industriellen Forschungskooperationen und den dort eingesetzten Frameworks.

Die ausführlichste Fallstudie im Bezug auf die Erstellung einer Framework-Beschreibung wurde für die Beschreibung der unterschiedlichen Versionen des »Java Servlet Frameworks« durchgeführt. In den Fallstudien zum »Java Server Faces Framework« und zum »Java Logging Framework« wurden jeweils nur Teile der Frameworks betrachtet und deren FDMM-basierte Beschreibung untersucht. Die Fallstudie zum »Google Android Framework« wurde im Rahmen einer Masterarbeit durchgeführt [Juergenfriedrich2011].

Die Fallstudien zur Framework-Beschreibung untersuchen unter anderem die Ausdrucksfähigkeit des FDMM. Darüber hinaus dienten sie als Praxistest für den Companian-Editor. In Tabelle 4.2 auf Seite 79 haben wir eine Reihe von Anforderungen an die Ausdrucksfähigkeit von Framework-Beschreibungssprachen aufgestellt. Wir wollen auf Basis der Fallstudien untersuchen, ob diese Anforderungen erfüllt wurden. Hierzu betrachten wir in Tabelle 6.3 jede Anforderung erneut und geben Referenzen zur Verwendung in den jeweiligen Fallstudien.

2 http://www.oracle.com/technetwork/java/index-jsp-135475.html
3 http://myfaces.apache.org/
4 http://logging.apache.org/log4j/
5 http://developer.android.com/about/versions/android-2.2.html

Tabelle 6.3: Ausdrucksfähigkeit im Kontext der Fallstudien zur Framework-Beschreibung

ID	Anforderung an die Ausdrucksfähigkeit Eine Framework-Beschreibungssprache muss...	Referenzen der Umsetzung
AUS-1	... Beispiele für die Benutzung des Frameworks beschreiben können.	• Im Servlet Framework wurde das Beispiel eines BasicServlets als Hypertext-Dokument beschrieben.
AUS-2	... statische und dynamische Architekturübersichten beschreiben können.	• Im Servlet Framework wurde der Ablauf eines Servlet-Requests modelliert. • Im Server Faces Framework konnte der Aufruf von PhaseListenern erfolgreich dokumentiert werden. • Im Android Framework wurde der Activity Lifecycle erfolgreich beschrieben [Juergen-friedrich2011].
AUS-3	... die Erweiterungspunkte eines Frameworks beschreiben können.	• Im Servlet Framework wurden alle Erweiterungspunkte beschrieben. • Im Logging Framework wurde ein LoggingAdapter als Erweiterungspunkt beschrieben.
AUS-4	... die Konfiguration und Integration des Frameworks beschreiben können.	• Die Beschreibung der Programmierschnittstelle (API) für Integration und Konfiguration wurde im Servlet Framework, im Server Faces Framework und im Logging Framework erfolgreich importiert und verlinkt.
AUS-5	... die Implementierung einer Erweiterung beschreiben können.	• Im Servlet Framework wurden die CodingUnits zu allen Erweiterungspunkten beschrieben. • Im Logging Framework wurde die CodingUnit für LoggingAdapter beschrieben.
AUS-6	... das Kommunikationsprotokoll zwischen Framework und Erweiterung beschreiben können.	• Im Servlet Framework wurde das Protokoll eines HTTP-Servlets und von ServletFiltern beschrieben. • Im Server Faces Framework wurde der Lebenszyklus von PhaseListenern als Protokolle beschrieben.
AUS-7	... die Bindung von Erweiterungen an das Framework beschreiben können.	• Bei Beschreibung von Erweiterungspunkten im Servlet Framework, im Server Faces Framework und im Logging Framework wurde der Bindungsmechanismus beschrieben.

ID	Anforderung an die Ausdrucksfähigkeit Eine Framework-Beschreibungssprache muss...	Referenzen der Umsetzung
AUS-8	... das Format und die Installation einer Erweiterung in das Framework beschreiben können.	• Bei Beschreibung von Erweiterungspunkten im Servlet Framework, im Server Faces Framework und im Logging Framework wurde der Installationsmechanismus beschrieben.
AUS-9	... einzuhaltende Bedingungen seitens einer Erweiterung beschreiben können.	• Im Servlet Framework wurden Bedingungen, die sich zum Beispiel auf Einschränkungen bei Nebenläufigkeit beziehen, bei Implementierung von Servlets angegeben.
AUS-10	... Varianten von Erweiterungspunkten beschreiben können.	• Im Servlet Framework wurden alle Varianten eines Servlets als Erweiterungspunkt gemäß der Spezifikation dokumentiert.

Aus den gegebenen Referenzen in Tabelle 6.3 geht hervor, dass wir für alle Anforderungen Beispiele in jeweils mindestens einer Fallstudie erfolgreich angewandt haben. Die erfolgreiche Anwendung bedeutet dabei, dass man die relevanten Informationen aus Sicht eines Framework-Entwicklers, der sein Framework beschreiben muss, erfassen und in Companian speichern konnte. Da Companian nur eine prototypische Implementierung darstellt, wurden kleinere Unzulänglichkeiten, die bei der Bedienung auftraten, außer Acht gelassen, wenn sie die grundsätzliche Funktionalität nicht beeinträchtigten.

Nicht zuletzt wurden auf Basis der durchgeführten Fallstudien Programmfehler im Companian-Editor entdeckt und behoben. Der praktische Einsatz hat in Einzelfällen auch zur direkten Verbesserung des FDMM geführt, wenn festgestellt wurde, das manche Informationen durch Anpassung des FDMM besser zu erfassen waren. So führte die Anwendung in Fallstudien zu einer inkrementellen Verbesserung des FDMM.

Anhand der Fallstudien lässt sich außerdem belegen, das mit dem FDMM und dessen Umsetzung im Companian-Editor die Anforderungen an die Anwendbarkeit aus Abschnitt 4.1.1 umgesetzt wurden. Wir greifen dies in Tabelle 6.4 auf und belegen die Umsetzung.

Die Fallstudien haben gezeigt, dass die Anforderungen an die Ausdrucksfähigkeit und Anwendbarkeit von Framework-Beschreibungssprachen mit dem FDMM und Companian erfolgreich umgesetzt wurden. Mit dem Companian-Editor haben wir das benötigte Werkzeug zur Erfassung des Ist-Zustands eines Frameworks in Form einer Framework-Beschreibung geschaffen und dessen praktische Einsatzfähigkeit in Fallstudien bestätigt.

Tabelle 6.4: Umsetzung der Anforderungen an die Anwendbarkeit

ID	Zielgruppe & Anforderung	Umsetzung im Companian-Editor
ANW-B	*Framework-Benutzer* Eine Framework-Beschreibungssprache muss Konzepte unterstützen, die einen Benutzer bei Verwendung einer Framework-Beschreibung, d.h. beim Explorieren und Erlernen eines Frameworks unterstützen.	Der Aufbau des FDMM ist bereits so gestaltet, dass diese Anforderungen berücksichtigt wird. In Companian kann ein Framework-Benutzer durch Folgen der Links das Framework Stück für Stück explorieren.
ANW-E	*Framework-Entwickler* Die Erstellung der Framework-Beschreibung muss sich so in den Entwicklungsprozess des Frameworks integrieren, dass der Aufwand der Entwickler für die Erstellung möglichst gering ist.	Wir werden in Abschnitt 6.5 im Detail eingehen. Es zeigt sich, dass sich die FDMM-basierte Beschreibung mit Companian nahtlos in Framework-basierte Entwicklungsprozesse integriert.
ANW-W	*Werkzeug-Entwickler* Die Informationen einer Framework-Beschreibung müssen für Werkzeug-Entwickler programmatisch auswertbar sein.	Das FDMM als Meta-Modell erlaubt es alle Informationen einer Framework-Beschreibung programmatisch auszuwerten. In Companian unterstützen wir dies durch ein XML-Format.

Im nachfolgenden Abschnitt 6.5 wollen wir uns nun damit befassen, wie eine FDMM-basierte Framework-Beschreibung mit Companian in etablierte Framework-basierte Softwareentwicklungsprozess integriert werden kann. Wir werden zeigen, dass mit dem FDMM die Frage beantwortet werden kann, wie ein Framework beschrieben werden sollte. Diese Frage wird in den existierenden Prozessen in der Regel nicht konkret aufgegriffen.

6.5 Integration in Framework-basierte Entwicklungsprozesse

Nachdem wir in den vorangegangenen Abschnitten das FDMM und seine Implementierung erläutert haben, befasst sich dieser Abschnitt damit, wie die Sprache methodisch eingesetzt werden kann.

Der Lösungsansatz aus Kapitel 3 verlangt, dass ein Framework vollständig mittels einer FDMM-basierten Sprache beschrieben ist. Nur wenn dies für jede Framework-Version der Fall ist, kann die Kompatibilität bei Aktualisierung auf eine neuere Framework-Version bestimmt werden. Bei diesem Vorgehen entstehen folgende Probleme, die von einer Methodik zur Framework-Beschreibung gelöst werden müssen.

- Da die Erstellung von Dokumentationen oftmals eine unliebsame und zeitaufwändige Aufgabe für einen Framework-Entwickler ist, muss die Erstellung der Framework-Beschreibung einfach sein.
- Eine Beschreibung muss vollständig in dem Sinne sein, dass alle Erweiterungspunkte beschrieben sind. Auch bei Weiterentwicklung des Frameworks muss sichergestellt werden, dass Änderungen an Erweiterungspunkten in der Dokumentation berücksichtigt werden. Dies stellt sicher, dass immer ein notwendiges Mindestmaß an Informationen über das Framework dokumentiert wird. Ohne dieses Mindestmaß könnte eine Kompatibilitätsanalyse nicht sinnvoll durchgeführt werden.
- Es muss sichergestellt werden, dass die Dokumentation des Frameworks nicht beschreibt, wie sich das Framework verhalten *soll*, sondern wie sich die konkrete Implementierung des Frameworks *real verhält*. Gefordert ist eine Beschreibung des Ist-Zustands. Diskrepanzen zwischen Implementierung und beschriebenem Verhalten müssen verhindert werden.

Das erste Problem kann dadurch angegangen werden, dass die Aufgabe der Dokumentation angemessen in den Framework-Entwicklungsprozess integriert wird. Meist ist die Aufgabe der Dokumentation einer der letzten Schritte im Entwicklungsprozess, der aus Zeitmangel oftmals vernachlässigt wird. Die Probleme, die aus einer schlechten Dokumentation entstehen, sind vielfältig und hinlänglich bekannt, was eigentlich schon Grund genug ist, um der Dokumentation einen deutlich höheren Stellenwert im Entwicklungsprozess zuzuweisen. Parnas benennt in [Parnas2011] das Problem einer präzisen Dokumentation gar als eine Schlüsselkomponente auf dem Weg zu besserer Software. Wie wichtig die Integration des Dokumentationsprozesses in den Entwicklungsprozess ist, zeigt auch der Beitrag von de Boer und van Vliet [Boer2009], in dem sie den Einfluss des Entwicklungsprozesses auf die Dokumentation und das Verständnis der Entwickler von der zu entwickelnden Software untersuchen. Sie zeigen in [Boer2009], dass Mitarbeiter ein gemeinsames Verständnis entwickeln, wenn sie im Entwicklungsprozess eng miteinander arbeiten, indem der eine ein Dokument nutzt, das der andere geschrieben hat.

Das Problem der Vollständigkeit kann nur durch eingehende Prüfungen sichergestellt werden. So wie das Framework als Software an sich qualitätsgesichert sein muss, so muss auch eine Framework-Beschreibung einer analytischen Qualitätssicherung unterzogen werden. Während einer solchen Qualitätssicherung kann die Vollständigkeit der Dokumentation geprüft werden. Eine solche Prüfung erfolgt idealerweise automatisiert, um den Prüfungsaufwand gering zu halten.

Im letzten Punkt wird auf eine besondere Eigenschaft von Dokumentationen verwiesen. Ein wichtiger Unterschied zwischen einer Softwarespezifikation und einer Dokumentation ist der, dass eine Spezifikation beschreibt wie sich eine Software verhalten *soll*. Eine Dokumentation hingegen muss erfassen, wie sich eine konkrete Implementierung der Software *real verhält*. Es ist denkbar, dass Softwareversionen herausgegeben werden, die noch keine vollständige

Implementierung einer Spezifikation sind. Eine Dokumentation muss daher inhaltlich den Teil der Spezifikation abdecken, der von Software in einer bestimmten Version erfüllt wird. Für die Dokumentation steht somit das konkrete System im Vordergrund und weniger die Spezifikation des Soll-Zustandes. Dies kann nur durch eine Kombination aus konstruktiven und analytischen Maßnahmen mit entsprechender Werkzeugunterstützung sichergestellt werden.

Es ist einleuchtend, dass Entwickler bei der Umsetzung konstruktiver und analytischer Techniken unterstützt werden müssen. Hierzu werden entsprechende Werkzeuge benötigt. Der richtige Einsatz maßgeschneiderter Werkzeuge führt zum einen dazu, dass die Akzeptanz bei den Beteiligten gestärkt wird, indem die gestellte Aufgabe leicht zu bewerkstelligen ist, und zum anderen zu einer effizienten Umsetzung, indem möglichst viele Schritte automatisiert werden.

Im Folgenden werden wir uns mit der Frage auseinandersetzen wie etablierte Framework-Entwicklungsprozesse durch das FDMM als Framework-Beschreibungssprache unterstützt werden können. Außerdem ist zu klären, wie die Aufgabe der Erstellung einer Framework-Beschreibung in diese Prozesse integriert ist.

In der Literatur zur Entwicklung und dem Einsatz von Frameworks findet man Hin- und Verweise auf die Existenz eines Framework-basierten Softwareentwicklungsprozesses (engl. framework-based software development) [Yang1998][Morisio1999][Bosch2000a], wobei meist nicht definiert wird, was genau darunter verstanden wird. Daher ist es häufig nicht eindeutig, wie unterschiedliche Autoren den Begriff auffassen. Wir verwenden hier eine Definition aus [Rogers1997].

> *[...] Framework-Based Software Development [...] is a methodology that relies on interconnected frameworks as the underpinning of applications.*
>
> **Framework-Based Software Development nach [Rogers1997]**

Eine Framework-basierte Entwicklung basiert darauf, dass für die eigentliche Anwendung ein oder mehrere unterliegende Frameworks entwickelt werden. Diese Frameworks liegen bei diesem Vorgehen nicht bereits vor, sondern entstehen während der Entwicklung der eigentlichen Anwendung mit. Es wird das Ziel verfolgt, dass man mit der gleichzeitigen Entwicklung eines Frameworks für nachfolgende Anwendungen möglichst viel Software wiederverwenden kann. Außerdem entsteht mit der Entwicklung von Frameworks eine sehr modulare Architektur, die sich positiv auf Softwarequalitäten wie die Wartbarkeit auswirkt. So kann die Entwicklung basierend auf Frameworks auch ein architektonisches Mittel sein.

Wir wollen an dieser Stelle den Framework-basierten Entwicklungsprozess beschreiben, wie er sich zusammengesetzt aus verschiedenen Quellen darstellt. Zunächst können wir festhalten, dass sich die existierenden Entwicklungsmethoden für Frameworks in zwei Herangehensweisen unterscheiden lassen [Fayad1999a]:

- Top-Down Ansätze starten mit einer fachlichen Analyse in der Domäne, um Gemeinsamkeiten und variable Aspekte zu identifizieren, die dann in einem technischen Framework-Entwurf verfeinert werden.
- Bottom-Up Ansätze entwickeln zunächst eine Reihe unterschiedlicher Anwendungen in derselben Domäne und entwickeln anschließend ein Framework aus den Gemeinsamkeiten.

Wir wollen im Folgenden die beiden unterschiedlichen Ansätze näher betrachten und jeweils untersuchen, wie sich die Dokumentation des Frameworks in diese Prozesse eingliedert. Es soll an dieser Stelle keine Wertung für oder gegen einen der existierenden Ansätze angestellt werden, denn beide scheinen sich je nach Ausgangssituation bewährt zu haben. Vielmehr soll aufgezeigt werden, wie die Erstellung einer FDMM-basierten Dokumentation in bestehende Entwicklungsprozesse integriert werden kann. Außerdem kann das FDMM dabei helfen das Problem des »Wie« zu lösen, wenn es um die Erstellung der Dokumentation geht. Viele Entwicklungsprozesse betrachten diesen wichtigen Punkt nur am Rande und gehen wenig detailliert auf ihn ein, was schon früh als Problem identifiziert, aber nach unserer Ansicht nie zufriedenstellend gelöst wurde [Butler1999][Bosch2000].

Es zeigt sich, dass eine FDMM-basierte Framework-Beschreibung den Entwicklungsprozess unterstützen kann. Damit wirkt sich der Einsatz des FDMM nicht nur positiv auf die inhaltliche Beschreibung eines Frameworks aus, als auch auf den Prozess zur Entwicklung von Frameworks. Eine Eigenschaft, die den Mehrwert des FDMM herausstellt. Gleichzeitig ist es ein Argument, dass Entwickler bereitwillig eine FDMM-basierte Framework-Beschreibung erstellen.

Top-Down-Frameworkentwicklung

Der Top-Down-Ansatz zur Framework-Entwicklung zeichnet sich dadurch aus, dass auf analytische Weise versucht wird, die Entwicklung einer Anwendung in einen generischen Kern in Form eines Frameworks und in die Entwicklung anwendungsspezifischer Erweiterungen zu trennen. Die Entscheidung über Gemeinsamkeiten im Kern und variabler Anteile wird in einer domänenspezifischen Analyse getroffen [Coplien1998].

Der Kern und gegebenenfalls Erweiterungen können für weitere Anwendungen wiederverwendet werden. Diese Form der Entwicklung kennt man aus verwandten Ansätzen zur Entwicklung von Software-Produktlinien, bei denen die Unterscheidung in gemeinsame und variable Anwendungsteile entscheidend ist [Clements2002]. Eine solche Methode besteht nach [Morisio1999] grundsätzlich aus zwei Prozessen:

- Entwicklung und Evolution des Frameworks
- Verwendung des Frameworks durch Entwicklung von Anwendungen

Diese sehr grobe Unterscheidung findet sich in allen in der Literatur beschriebenen Entwicklungsprozessen. Der Grundgedanke ist, dass man ein Framework nur (weiter-)entwickeln kann, wenn man es gleichzeitig benutzt

und über die Verwendung zu neuen Erkenntnissen gelangt. Die Entwicklung eines Frameworks durchläuft üblicherweise eine ganze Reihe von Iterationen aus Entwicklung und Verwendung bis es stabil ist. Das Konzept einer iterativen Entwicklung ist somit sehr zentral für die Framework-Entwicklung und wird in einer ganzen Reihe von Ansätzen immer wieder angewandt (vergl. [Yang1998] [Pree2000][Gurp2000][Markiewicz2001][Carey2002]).

Betrachtet man die Entwicklung als Phasen aus Analyse und Entwurf, wie in Abbildung 6.34 zu sehen, entsteht am Ende der Entwurfsphase das Framework. Das Framework kann dann in einer Reihe von Anwendungen verwendet werden.

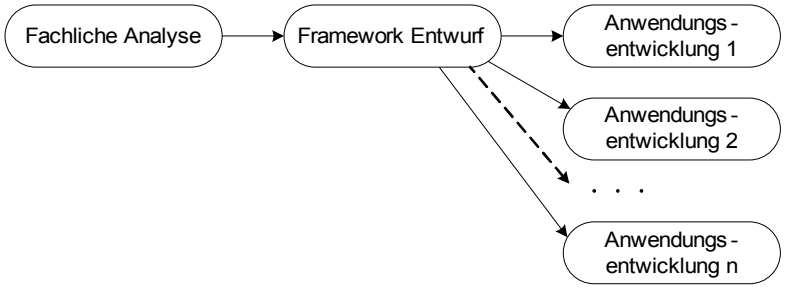


Abbildung 6.34: Entwicklungsprozess bestehend aus Analyse, Entwurf und Verwendung eines Frameworks

In [Carey2002] werden eine Reihe von Entwicklungsartefakten identifiziert, die innerhalb eines Framework-Entwicklungsprozesses entwickelt werden. Wie diese Artefakte modelliert werden ist nicht vorgegeben. Denkbar ist zum Beispiel eine durch die UML gestützte Modellierung, wie sie [Yang1998] vorgeschlagen wird. Abbildung 6.35 zeigt diese Artefakte.

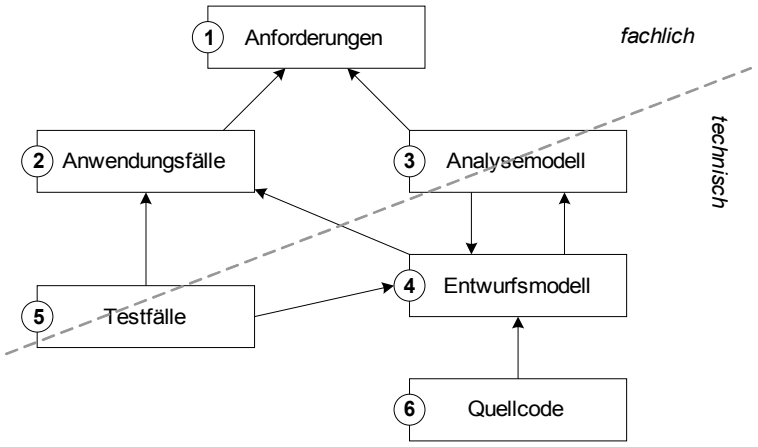


Abbildung 6.35: Entwicklungsartefakte und deren Abhängigkeiten nach [Carey2002]

Die Übersicht in Abbildung 6.35 ist unterteilt in fachliche und technische Artefakte. Der Entwicklungsprozess in [Carey2002] sieht nun vor, dass bei Erstellung des Analysemodells bereits auf fachlicher Ebene dokumentiert werden sollte, wo das Framework erweiterbar ist.

Die Analysemodelle basieren im Kontext von Software-Produktlinien häufig auf so genannten Feature-Graphen, die Abhängigkeiten zwischen festen und variablen Features einer Anwendung zeigen. Feature-orientierte Ansätze wie zum Beispiel »FORM« (Feature-Oriented Reuse Method) [Kang1998], sind somit naheliegend für die Analysephase während der Framework-Entwicklung. Ziel ist es, die fachlichen Erweiterungspunkte zu identifizieren. Ein weiterer Ansatz aus dem Bereich der Software-Produktlinien ist zum Beispiel »FAST« für »Family-Oriented Abstraction, Specification, and Translation« [Weiss1999]. Eine vergleichbare Vorgehensweise in einer neueren Arbeit beschreiben die Autoren Pohl, Böckle und van der Linden in [Pohl2005].

Diese fachlichen Erweiterungspunkte werden im Entwurfsmodell mit Hilfe von Design-Patterns [Gamma1993] umgesetzt. In dieser Phase sind die Erweiterungspunkte des Frameworks identifiziert und deren technische Umsetzung spezifiziert. Neben Design-Patterns können im Entwurf auch so genannte »Framelets« [Pree1998][Pree2000a] verwendet werden. Nach [Pree1998] ist „ein Framelet [...] ein kleiner, flexibler Architekturbaustein, welcher

- nicht den Hauptkontrollfluß einer Anwendung an sich zieht,
- klein (weniger als 10 Klassen) ist,
- eine klar definierte, einfache Schnittstelle hat.

Ansonsten weist ein Framelet die Eigenschaften eines Frameworks auf. Insbesondere beruht es auf [...] einer Whitebox-Architektur.“ Framelets als kleine Architekturbausteine eignen sich, um komplexe Frameworks aus ihnen zu komponieren.

Die finale Dokumentation wird im Entwicklungsprozess nach [Carey2002] erst nach der Erstellung des Quellcodes angefertigt. Es gibt aber den Hinweis, das man bei den vorhergehenden Artefakten wie Analyse- und Entwurfsmodell bereits immer so viel an Informationen erfassen sollte, dass man keine Informationen für die finale Dokumentation vergisst.

Eine Dokumentation besteht nach dem »IBM San Francisco Framework Development Process« in [Carey2002] aus folgenden Artefakten, die wir hier mit den vorhandenen Informationen einer FDMM-basierten Sprache in Verbindung setzen.

- Eine *funktionale Übersicht*, die wir im FDMM als architektonische Sichten modellieren können.
- Einen *Nachschlagewerk für Erweiterungen*, die im FDMM über Hot-Spots abgebildet werden.
- Ein *Glossar*, was der Sammlung aller Elemente einer FDMM-basierten Sprache entspricht.
- Ein *Benutzerhandbuch*, das im FDMM über die Beschreibung von Beispielen erzeugt werden kann.
- Eine *Anleitung zur Speicherung von Erweiterungen*, die im FDMM der Beschreibungen der Bindung und der Installation entsprechen.

Es wird deutlich, dass das FDMM alle benötigten Informationen abbilden kann, um die benötigten Dokumentationsartefakte zu erzeugen. Außerdem erkennt man in diesem Vorgehen, dass die ausführliche Dokumentation des Frameworks bereits integraler Bestandteil einer Top-Down-Frameworkentwicklung ist und nicht etwa neu hinzugefügt werden müsste.

Entscheidender Punkt in der Framework-Entwicklung ist die Identifikation der variablen Teile und damit der Erweiterungspunkte eines Frameworks. Manche Entwicklungsmethoden identifizieren diese erst in der Entwurfsphase, andere versuchen bereits in der Analysephase variable Anteile zu identifizieren. Unabhängig davon müssen die identifizierten Erweiterungspunkte mit dem Wissen aus der Entwicklungsphase dokumentiert werden. Das FDMM bietet hierzu alle Möglichkeiten.

Bereits während der Analysephase kann man auf abstrakter Ebene Analysearchitekturen entwerfen, die als Architekturübersichten in einer FDMM-basierten Dokumentation abgelegt werden können. Zu diesen Übersichten können bereits identifizierte Erweiterungspunkte dokumentiert und mit der Architektur in Beziehung gesetzt werden. Auf dieser abstrakten Ebene sind selbstverständlich noch nicht alle Informationen verfügbar, aber es ist möglich bereits vorhandenes Wissen abzulegen und in späteren Entwicklungsphasen zu vervollständigen. Wir wollen dies am Vorgehen des »Hot-Spot-Driven Framework Development« [Pree2000] verdeutlichen. Die folgende Abbildung 6.36 zeigt den Prozess nach [Pree1995][Pree2000].

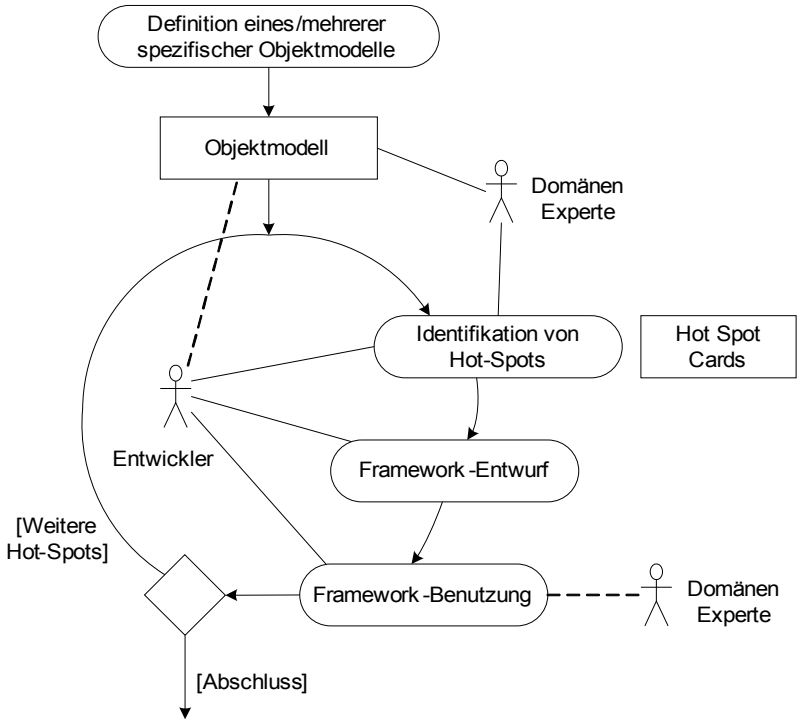


Abbildung 6.36: »Hot Spot Driven« Entwicklungsprozess

Beim Vorgehen des »Hot-Spot-Driven Framework Development« [Pree2000] sind die Erweiterungspunkte zentraler Bestandteil der Methode und ihre frühe

Identifikation der Schlüssel zur Entwicklung eines Frameworks. Die Entwicklung beginnt mit einer Analysephase, in der von Domänenexperten, häufig mit Unterstützung von Entwicklern, zunächst ein Analysemodell in Form eines oder mehrerer Objektmodelle erstellt wird. Dieses Analysemodell ist Voraussetzung, um mit dem sich anschließenden Framework-Entwicklungszyklus zu starten. Innerhalb des Zyklus werden in einem iterativen Vorgehen Erweiterungspunkte identifiziert, in den Entwurf übernommen. implementiert und anschließend benutzt. Die gesammelten Erfahrungen fließen in die Identifikation weiterer Erweiterungspunkte während der nächsten Iteration ein. Die folgende Abbildung 6.36 zeigt den Prozess nach [Pree1995][Pree2000].

Die Identifikation von Erweiterungspunkten (Hot-Spots) ist die entscheidende Phase im Entwicklungsprozess. Da es nicht gelingen wird alle Erweiterungspunkte in einem einzigen Durchlauf zu bestimmen, wird der Zyklus aus identifizieren, entwerfen und benutzen so lange durchlaufen, bis man keine weiteren Erweiterungspunkte mehr identifiziert.

Beteiligt an diesem Prozess sind sowohl Experten der Anwendungsdomäne des Frameworks als auch die Entwickler des Frameworks. Um bereits den Experten ein Mittel an die Hand zu geben, Erweiterungspunkte festhalten zu können, werden so genannte »Hot Spot Cards« eingesetzt. Auf einer solchen Karte kann ein Experte mit Unterstützung des Entwicklers vermerken, das eine bestimmte Funktionalität variabel gestaltet werden sollte. Abbildung 6.37 zeigt den schematischen Aufbau einer Hot-Spot-Karte.

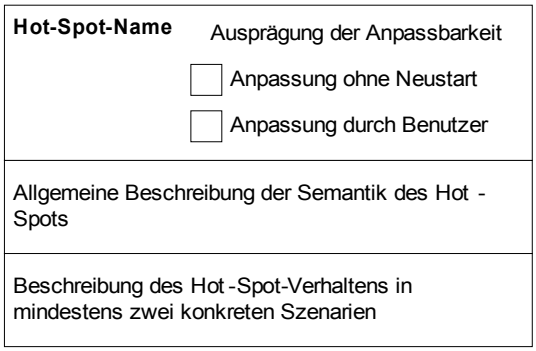


Abbildung 6.37: Schema einer Hot-Spot-Karte

Neben einem eindeutigen Namen für den identifizierten Erweiterungspunkt wird auf einer Hot-Spot-Karte zunächst festgehalten, wann und durch wen die Variabilität gesteuert werden kann. Als entscheidend werden hierbei die Fragen angesehen, ob das Verhalten im laufenden Betrieb anpassbar sein muss und ob der Benutzer das Verhalten verändern können soll. Wird bei beiden Feldern kein Haken gesetzt, so ist die Anpassung nur durch Neustart des Systems möglich und es muss ein Systemexperte bzw. Entwickler herangezogen werden, um die Anpassung durchzuführen.

Hot-Spot-Karten werden als Kommunikationsmittel zwischen Experten und Entwicklern eingesetzt. Der Entwickler enthält wichtige Informationen für den Framework-Entwurf und der Experte kann auf dieser Ebene fachliche Ent-

scheidungen treffen ohne überhaupt ein Verständnis von Frameworks zu benötigen.

Wir wollen zeigen, wie sich Hot-Spot-Karten mit einer FDMM-basierten Sprache modellieren lassen. Hierzu betrachten wir in Abbildung 6.38 das Beispiel einer Hot-Spot-Karte aus [Pree2000] für die variable Gestaltung der Kostenberechnung einer Rechnungserstellung.

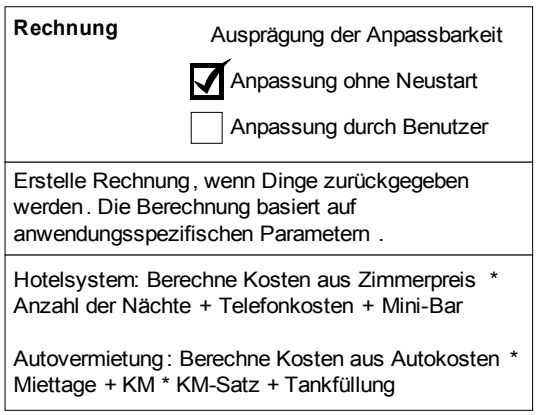


Abbildung 6.38: Hot-Spot-Karte »Rechnung«

Nun übertragen wir diese konkrete Hot-Spot-Karte »Rechnung« in eine FDMM-Instanz, um zu zeigen, wie sich die Informationen FDMM-basiert modellieren lassen. Im FDMM entspricht jede Hot-Spot-Karte einem HotSpot Objekt. Die Beschreibung der Semantik wird als Wert des Attributs description im FDMM abgelegt. Die »Ausprägung der Anpassbarkeit« kann über Constraints modelliert werden. Beschreibungen konkreter Szenarien für den Hot-Spot werden über unterschiedliche HotSpotExamples erfasst.

Konkret bedeutet dies, dass wir für die Hot-Spot-Karte »Rechnung« zunächst ein Objekt Rechnung vom Typ HotSpot anlegen. Das entstehende Objektdiagramm zeigen wir in Abbildung 6.39.

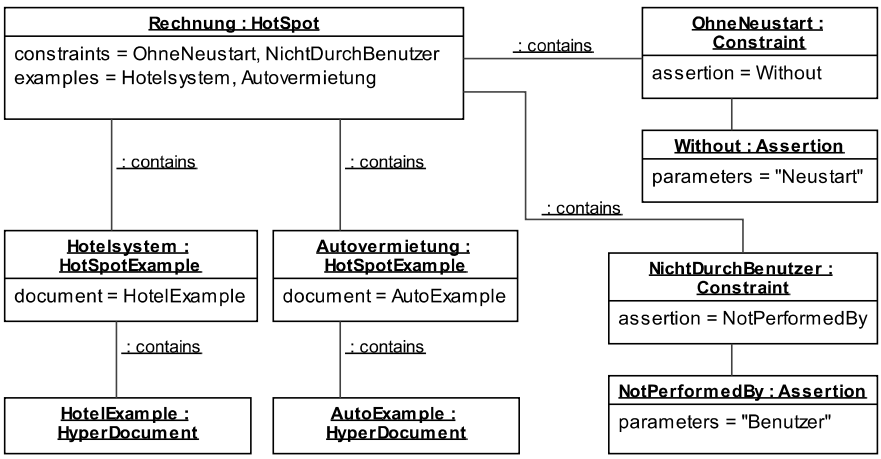


Abbildung 6.39: FDMM-Instanz für den Hot-Spot-Karte »Rechnung«

Das Objekt Rechnung ist mit zwei HotSpotExample Objekten verlinkt. Ein Objekt für das Beispiel Hotelsystem und ein Objekt für Autovermietung. Die Ausprägung der Anpassbarkeit können über Constraint-Objekte OhneNeustart und NichtDurchBenutzer modelliert und mit dem HotSpot Rechnung verlinkt werden. Die beschreibenden Inhalte der Beispiele werden in den HyperDocuments der HotSpotExample-Objekte hinterlegt.

Wir sehen, dass eine FDMM-basierte Beschreibung alle Informationen zur Identifikation von Hot-Spots in einem »Hot Spot Driven« Entwicklungsprozesses aufnehmen kann. Darüber hinaus kann diese Beschreibung während der Entwurfsphase mit weiteren Informationen angereichert werden.

Während des Framework-Entwurfs werden die identifizierten Hot-Spots mit Hilfe von Entwurfsmustern in Programmcode überführt. Je nach Ausprägung des Hot-Spots werden hierzu unterschiedliche Entwurfsmuster eingesetzt. Die exakten Details der internen Implementierung des Frameworks werden in den Entwurfsdokumenten erfasst. Es gilt zu beachten, dass das FDMM nicht dazu geschaffen wurde, um die Informationen des Entwurfs aufzunehmen, sondern Informationen zur Benutzung des Frameworks. Dennoch können viele Informationen aus dem Entwurf direkt in eine Benutzungsbeschreibung überführt bzw. mit dieser verknüpft werden. So können zum Beispiel die Beschreibungen der verwendeten Hook-Methoden gleichzeitig für den Entwurf und die Benutzung im FDMM erfasst werden. Die Dokumentation der API dient ebenfalls sowohl dem Entwurf als auch der Benutzung. Andere Details des Entwurfs wie die intern eingesetzten Entwurfsmuster sind hingegen irrelevant für die Benutzung und werden daher im FDMM nicht erfasst.

Es wird deutlich, dass sich das FDMM nahtlos in einen Top-Down-Entwicklungsprozess integrieren lässt. Bisher mangelte es jedoch an konkreten Modellierungsmöglichkeiten für die vorliegenden Informationen innerhalb des Prozesses. Das FDMM schließt diese Lücke und unterstützt die Top-Down-Entwicklung eines Frameworks. Im Folgenden wollen wir uns der Bottom-Up-Entwicklung zuwenden und untersuchen wie das FDMM hier unterstützen kann.

Bottom-Up-Frameworkentwicklung

Bei der Bottom-Up-Methodik versucht man aus einer Reihe existierender Anwendungen den gemeinsamen Kern in Form eines Frameworks abzuleiten. Die Grundüberlegung bei diesem Vorgehen ist, dass man nur von konkreten Lösungen zu allgemeinen Konzepten gelangen kann. Der Gedanke ist, dass man in konkreten Anwendungen erkennt, welche Teile sich für eine Wiederverwendung in Form eines Frameworks eignen. Alle Versuche den korrekten Abstraktionsgrad am Reißbrett zu finden ohne realistische Anwendungen einzubeziehen seien nach [Roberts1998] zum Scheitern verurteilt: „Every attempt to determine the correct abstractions on paper without actually developing a running system is doomed to failure. No one is that smart.“

Beim Bottom-Up-Ansatz versucht man auf konstruktive Weise, konkrete Anwendungen in einen generischen Kern und in anwendungsspezifische Erweiterungen zu trennen. Der Kern und gegebenenfalls einige Erweiterungen können so für weitere Anwendungen wiederverwendet werden. Dieses Vorge-

hen kann auch für eine einzelne Anwendung zielführend sein, wenn man die Entwicklung eines Frameworks als architektonisches Mittel begreift. Vergleiche hierzu auch [Bouassida2002] und [Ben-Abdallah2004].

Im Folgenden beziehen wir uns auf den Bottom-Up-Ansatz wie er in [Roberts1998] beschrieben wird, da dieses Vorgehen auch von neueren Arbeiten noch immer als grundlegend angesehen wird [Cunningham2006][Cunningham2006a].

Das Vorgehen beginnt damit, dass man mindestens drei Anwendungen der selben Domäne entwickelt. Diese konkreten Anwendungen sind der Ausgangspunkt für die weiteren Schritte, die in Form von anleitenden Mustern beschrieben sind. Mit jedem angewandten Muster im Prozess kommt man dem Ziel eines Frameworks ein Stück näher. Die Entwicklung konkreter Anwendungen im ersten Schritt ist dabei auch ein angewendetes Muster. Abbildung 6.40 zeigt alle angewandten Muster Bottom-Up über die Zeit.

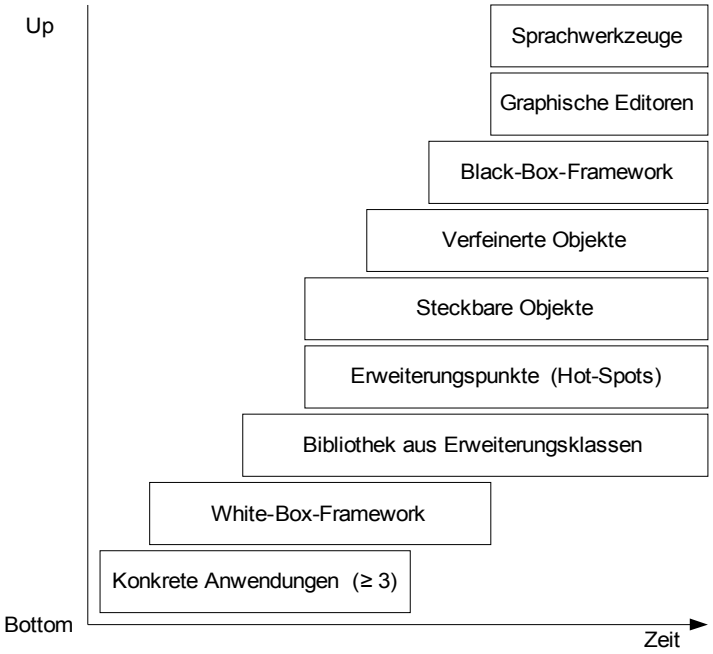


Abbildung 6.40: Muster beim Bottom-Up-Vorgehen

Zur Entwicklung der geforderten konkreten Anwendungen kann man nach [Roberts1998] grob vier Vorgehen unterscheiden.

- *Sequentiell aufbauende Entwicklung*
  Bei einer sequentiell aufbauenden Entwicklung entwickelt das selbe Team eine Reihe von konkreten Anwendungen nacheinander. Bereits ab der zweiten Anwendung wird versucht Teile aus der ersten Anwendung wiederzuverwenden.

- *Parallele Entwicklung*
  Bei einer parallelen Entwicklung werden die Anwendungen von unterschiedlichen Teams parallel entwickelt, um möglichst viele unterschiedliche Einflüsse für das spätere Framework zu generieren.

- *Unabhängige Entwicklung*
  Bei einer unabhängigen Entwicklung sind die Anwendungen nicht im Hinblick auf eine Framework-Entwicklung entstanden, sondern während oder nach der Entwicklung einer Reihe von Anwendungen entstand erst der Gedanke, dass man ein Framework entwickeln sollte.

- *Prototypentwicklung*
  Es werden eine Reihe von Anwendungen nur prototypisch entwickelt, um hierbei zu lernen, worauf bei einem späteren Framework zu achten ist. Die Prototypen können außerdem dazu dienen, potentiellen Kunden das spätere Framework in unterschiedlichen Anwendungen zu demonstrieren.

Hat man konkrete Anwendungen zur Hand, kann man mit einer Reihe von Mustern den anstehenden Framework-Entwurf immer weiter verfeinern und ergänzen. Zunächst bietet es sich an, anhand einer konkreten Anwendung mit der Entwicklung eines Whitebox-Frameworks (vgl. Abschnitt 2.2.1) zu starten, wobei die Anpassungen für die konkreten Anwendungen vornehmlich durch Vererbungen realisiert werden. Im Whitebox-Framework werden die abstrakten Klassen abgelegt, von denen in der konkreten Anwendung geerbt wird.

Mit einer ersten Version eines Whitebox-Frameworks kann nun begonnen werden die weiteren konkreten Anwendungen umzustellen. Hierbei wird man vermutlich feststellen, dass man bereits erstellte konkrete Erweiterungen als abstrakte Klassen wiederverwenden möchte. Hierzu setzt man das Muster einer Bibliothek ein, in der konkrete Erweiterungsklassen abgelegt werden. Mit der Bibliothek hat man zusätzlich zum Whitebox-Framework eine Sammlung wiederverwendbarer Erweiterungsklassen angelegt.

Umso mehr man nun das Whitebox-Framework mit Erweiterungsklassen benutzt, umso mehr wird man feststellen, dass man ähnlichen Code immer wieder schreiben muss. Diese Codestellen wurden von Wolfgang Pree als Hot-Spots [Pree1995] bezeichnet. Wir sprechen an dieser Stelle von Erweiterungspunkten, die nun identifiziert werden und durch den Einsatz von Entwurfsmustern [Gamma1995] implementiert werden. Hierbei wird die zuvor vorherrschende Technik der Vererbung mehr und mehr durch Komposition in Entwurfsmustern ersetzt, die eine bessere Wiederverwendung ermöglichen.

Leider werden in [Roberts1998] keine Muster zur Dokumentation des Frameworks beschrieben, aber mit der Identifikation der Erweiterungspunkte haben wir auch in diesem Prozess den Punkt erreicht, an dem wir zeigen können, wie eine FDMM-basierte Dokumentation den weiteren Prozess unterstützen kann. Bereits mit der Einführung des Whitebox-Frameworks sollte damit begonnen werden die API zu dokumentieren. Durch den parametrisierten Aufbau des FDMM können diese API Informationen direkt mit den nun zu dokumentierenden Erweiterungspunkten verknüpft werden.

Sobald Erweiterungspunkte mit Hilfe von Entwurfsmustern und dem Muster »Steckbare Objekte« identifiziert und im Framework refaktorisiert werden, beginnt auch die konkrete Dokumentation des Frameworks. Mit einer FDMM-basierten Sprache können, analog zum Top-Down-Vorgehen, alle vorliegenden Informationen abgelegt werden. Mit steckbaren Objekten meint man hier ein

Vorgehen zur Verfeinerung von Klassen hin zu dynamisch anpassbaren Konstrukten, die zum Beispiel auch Veränderungen zur Laufzeit ermöglichen. All die entstehenden Details zur korrekten Benutzung der so entstehenden Erweiterungspunkte, inklusive funktionaler und nicht-funktionaler Einschränkungen, lassen sich direkt FDMM-basiert festhalten.

Nach Abschluss dieser Arbeiten ist das Framework so weit entwickelt, dass es von Entwicklern in neuen Anwendungen eingesetzt werden kann. Die FDMM-basierte Dokumentation hilft bei der Benutzung des Frameworks. Die nachfolgenden Schritte im Vorgehen nach [Roberts1998] beziehen sich darauf, dass man das Framework so weit weiterentwickelt, dass auch Nicht-Entwickler, z.B. Domänenexperten, mit dem Framework arbeiten können. Hierzu versucht man die steckbaren Objekte weiter zu verfeinern und das Whitebox-Framework mehr und mehr in ein Blackbox-Framework (vgl. Kapitel 2.2.1) umzuwandeln. Dann wird damit begonnen graphische Editoren und domänenspezifische Werkzeuge zu bauen. So gelangt man zu dem Ziel, dass auch Nicht-Entwickler, ohne programmieren zu müssen, in ihrer domänenspezifischen Sprache mit dem Framework arbeiten können.

In dieser Phase kann bei Überführung von einem Whitebox- zu einem Blackbox-Framework eine FDMM-basierte Dokumentation unterstützen. Erweiterungspunkte, die bisher noch nach Whitebox-Prinzip aufgebaut waren, können systematisch in Blackbox-Varianten überführt und dabei entsprechend dokumentiert werden. Die FDMM-basierte Dokumentation hilft alle Informationen im Überblick zu behalten und alle dokumentierten Eigenschaften der Erweiterungspunkte zu berücksichtigen.

Die Dokumentation für Nicht-Entwickler wird von einer FDMM-basierten Sprache nicht primär berücksichtigt, so dass hier erweiterte Mechanismen geschaffen werden müssen. Dadurch, dass alle Informationen im FDMM strukturiert abgelegt sind, können diese in eine Vorlage für ein Benutzerhandbuch für Nicht-Programmierer eingesetzt werden. Da im FDMM jedoch keine Informationen über die Benutzung etwaiger graphischer Editoren abgelegt werden können, müssen diese Informationen dem Handbuch noch hinzugefügt werden. Dennoch kann hier eine FDMM-basierte Sprache helfen sicherzustellen, dass im Handbuch alle Informationen über das Framework berücksichtigt wurden. So kann eine FDMM-basierte Dokumentation eine qualitätssichernde Rolle im Entwicklungsprozess einnehmen.

Abschließend betrachten wir in Abbildung 6.41 noch einmal die angewandten Muster nach [Roberts1998] und vervollständigen das Bild um den Aspekt der Dokumentation. Die FDMM-basierte Dokumentation hilft die Bottom-Up-Entwicklung zu unterstützen und das entstehende Framework für Dritte benutzbar zu machen.

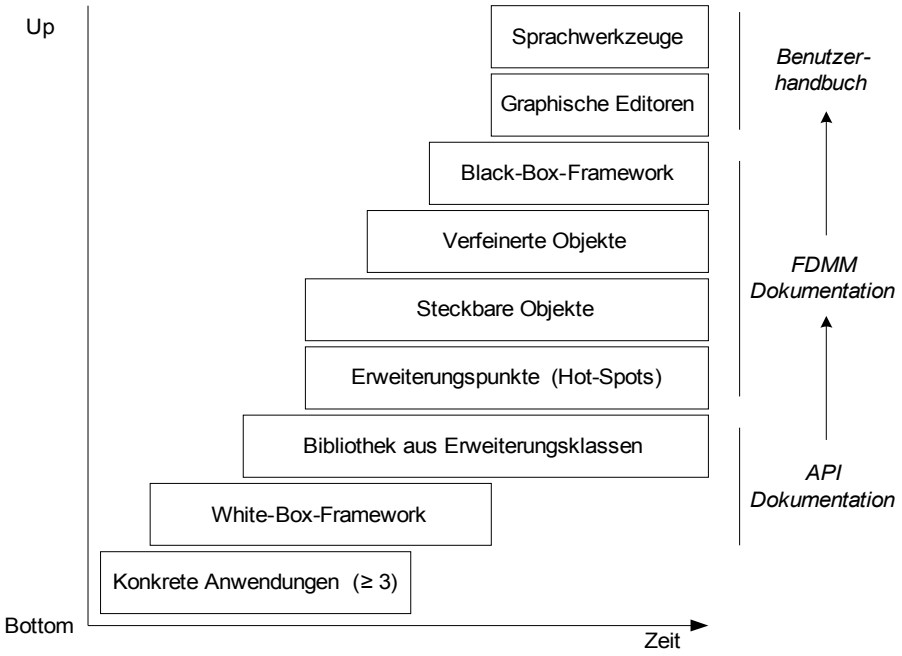


Abbildung 6.41: Muster und Dokumentation beim Bottom-Up-Vorgehen

Wir haben gesehen wie eine FDMM-basierte Dokumentation auch das Bottom-Up-Vorgehen direkt unterstützen kann. Die Erkenntnis ist auch hier, dass es keine neue Aufgabe ist, Informationen zu erfassen und abzulegen, aber mit dem FDMM hat man eine konkret anwendbare Struktur zur Hand, die einen bei dem »Wie« zur Erfassung der Informationen entscheidend hilft.

Im Folgenden wollen wir uns nun einem weiteren Problem bei der Erstellung von Dokumenten widmen. Wir haben gesehen, wie sich die Erstellung einer FDMM-basierten Dokumentation in Entwicklungsprozesse integriert. Wir müssen darüber hinaus die Qualität der Dokumentation sichern. Entscheidend für unseren Ansatz ist, dass die FDMM-basierte Dokumentation immer den Ist-Zustand des Frameworks in seiner aktuellen Version beschreibt und nicht einen gewünschten Soll-Zustand.

Eine Grundlage für die Übereinstimmung von Ist und Soll haben wir durch eine konstruktive Integration in den Entwicklungsprozess geschaffen. Die Dokumentation entsteht größtenteils parallel zur Entwicklung, so dass sämtliche Einflüsse auf die Entwicklung sich auch auf die Dokumentation auswirken. Leider ist dies nur in der Theorie die perfekte Lösung, da es immer wieder passiert, das Dinge zwar in der Entwicklung geändert, aber nicht dokumentiert werden. Um diese Änderungen nachvollziehen zu können, muss man zu weiteren Maßnahmen und Werkzeugen greifen.

Wie bereits in Kapitel 3 skizziert, verwenden wir zur Lösung der Problemstellung in dieser Arbeit statische Codeanalysen, um die Benutzungsbeziehung zwischen einer Anwendung und einem Framework zu bestimmen. Details dieser Analyse werden wir in Kapitel 7.2 darstellen. An dieser Stelle wollen wir darauf eingehen, dass uns statische Codeanalysen helfen können die Erweiterungspunkte in einem Framework automatisch zu identifizieren.

Wie wir in unserer Analyse Framework-basierter Entwicklungsprozesse gesehen haben, werden für die Implementierung von Erweiterungspunkten in objektorientierten Frameworks in der Regel Entwurfsmuster nach [Gamma1995] eingesetzt. Dieses Wissen kann man sich nun zu Nutze machen, um nach entsprechenden Hinweisen für die Verwendung von Entwurfsmustern zu suchen. Theoretisch kann ein implementiertes Entwurfsmuster im Framework einem Erweiterungspunkt entsprechen, der selbstverständlich auch dokumentiert sein müsste. Eventuell lässt sich die Liste der zu suchenden Entwurfsmuster einschränken, um die Suche zu optimieren.

Leider hat die Suche nach Erweiterungspunkten über Entwurfsmuster den Nachteil, dass Entwurfsmuster nicht nur bei der Implementierung von Erweiterungspunkten verwendet werden. Der Einsatz von Entwurfsmustern hat sich zu einer etablierten Technik im Softwareentwurf entwickelt, so dass man gegebenenfalls viele Entwurfsmuster in einem Framework finden wird, die jedoch nicht alle in Zusammenhang mit Erweiterungspunkten stehen. Die Anzahl falsch-positiver Treffer kann unter Umständen hoch sein.

Das Ziel einer strukturierten Qualitätssicherung lässt sich jedoch auch erreichen, wenn die automatische Suche nach Erweiterungspunkten (noch) nicht perfekt funktioniert. Es steigt lediglich der Anteil manuell zu prüfender Erweiterungspunkte. Eine Kombination aus konstruktiver Qualitätssicherung während des Entwicklungsprozesses und analytischer Qualitätssicherung vor Auslieferung des Frameworks kann sicherstellen, dass das Framework gemäß seines Ist-Zustands vollständig dokumentiert wurde.

Analytisch kann man Überdeckungsmetriken anlegen, die den Grad der Dokumentation in Bezug auf Programmcode und Erweiterungspunkte prüfen. Ebenso lassen sich etwaige Dokumentationen eines Soll-Zustands aufdecken, wenn etwa ein Erweiterungspunkt bereits beschrieben, aber kein Verweis im Quellcode gefunden werden kann. Die Qualitätssicherung ist eng verknüpft mit der Fähigkeit Programmcode mit der Dokumentation zu verlinken, wie es bereits existierende Ansätze wie [Demeyer2000] versuchen. Hier versucht man die Verbindung zwischen Code und Dokumentation zu bestimmen und diese mit Hilfe von »Hypermedia Links« zu modellieren.

Da die automatische Bestimmung der Links Probleme bereitet, kann nur eine enge Verzahnung der konstruktiven Maßnahmen im Entwicklungsprozess mit analytischen Prüfungsmöglichkeiten sowie einer optimalen Werkzeugunterstützung die Qualität einer FDMM-basierten Framework-Beschreibung gewährleisten. Im Abschnitt Fehler: Referenz nicht gefunden wollen wir die Möglichkeiten einer Werkzeugunterstützung betrachten, die den Entwickler mit möglichst wenig zusätzlichem Aufwand dabei unterstützt, eine konsistente Beschreibung des Ist-Zustandes anzufertigen.

Grundsätzlich denken wir, dass bei Erkennung des Mehrwerts einer FDMM-basierten Dokumentation eine zentrale Voraussetzung gegeben ist, dass FDMM-basierte Sprachen Einzug in die Praxis halten. Dies gibt Anlass zur Hoffnung, dass in der zukünftigen Praxis Frameworks FDMM-basiert dokumentiert werden.

6.6 Zusammenfassung

In diesem Kapitel haben wir das »Framework Description Meta-Model« (FDMM) als eine ganzheitliche Framework-Beschreibungssprache eingeführt, die alle gestellten Anforderungen aus den Bereichen Anwendbarkeit, Ausdrucksfähigkeit und Erweiterbarkeit erfüllt. Mit dem Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen und einem parametrisierten Meta-Modell sind wir in der Lage existierende Sprachen als Teil des FDMM wiederzuverwenden. So konnte eine nahtlose Integration von UML-Aktivitätendiagrammen, UML-Protokollzustandsautomaten und der JavaDoc-Sprache erreicht werden.

Auf Werkzeugseite bietet der Companian-Editor alle Funktionalitäten, um eine FDMM-basierte Framework-Beschreibung anzulegen. Durch die Wiederverwendung existierender Sprachen konnten auch existierende Werkzeuge für diese Sprachen wiederverwendet werden. Im Falle der UML-Diagramme konnte etwa das Werkzeug »MagicDraw« eingesetzt werden, um die Diagramme zu erstellen.

Die Durchführung einiger Fallstudien zur FDMM-basierten Beschreibung etablierter Frameworks bestätigte die praktische Einsetzbarkeit des FDMM und dessen Umsetzung im Editor von Companian. Damit haben wir mit dem FDMM eine Framework-Beschreibungssprache und mit Companian eine Werkzeugunterstützung geschaffen, die wir nun für die Umsetzung der eigentlichen Problemstellung einer automatischen Kompatibilitätsprüfung im nachfolgenden Kapitel 7 einsetzen können.

7

Automatische Kompatibilitätsprüfung mit Companian

„One of the things I learned was
how much software occupies the brain.
It was a much more difficult task
than I expected.“

-- Donald Knuth in [Seibel2009]

In Kapitel 3 haben wir den Ansatz einer automatischen Kompatibilitätsprüfung beschrieben. Eine Voraussetzung für diesen Ansatz war es, eine geeignete Framework-Beschreibungssprache zu haben, mit der sich der Ansatz umsetzen lässt. Auf Basis der Grundlagen zu Framework-Beschreibungssprachen in Kapitel 4 und dem Ansatz zur anforderungsbasierten Wiederverwendung von Sprachen in Kapitel 5, haben wir mit dem »Framework Description Meta-Model« (FDMM) die benötigte Framework-Beschreibungssprache definiert.

Das in dieser Arbeit entwickelte Werkzeug »Companian« implementiert das FDMM und ermöglicht die Erstellung von Framework-Beschreibungen als »Framework Description Models« (FDM) mit dem Companian-Editor. Wir

wollen uns in diesem Kapitel der Umsetzung der automatischen Kompatibilitätsprüfung in Companian widmen.

7.1 Companian Übersicht

Für eine prototypische Umsetzung des Ansatzes in Companian beschränken wir uns auf Anwendungen und Frameworks, die in der Programmiersprache Java entwickelt wurden. Der Grund hierfür liegt darin, dass Java im Umfeld betriebswirtschaftlicher Informationssysteme weit verbreitet ist. Somit ist die Unterstützung dieser Technologie sinnvoll für eine Vielzahl existierender Anwendungen und Frameworks.

In Abbildung 7.1 sehen wir ein Bildschirmfoto der Startseite von Companian. Companian ist eine Web-Anwendung, die über einen Web-Browser bedient wird. Als solche kann Companian zentral auf einem Server installiert und von mehreren Benutzern gleichzeitig verwendet werden. In einem Unternehmen könnte Companian so als zentrale Dokumentations- und Analyseplattform dienen.

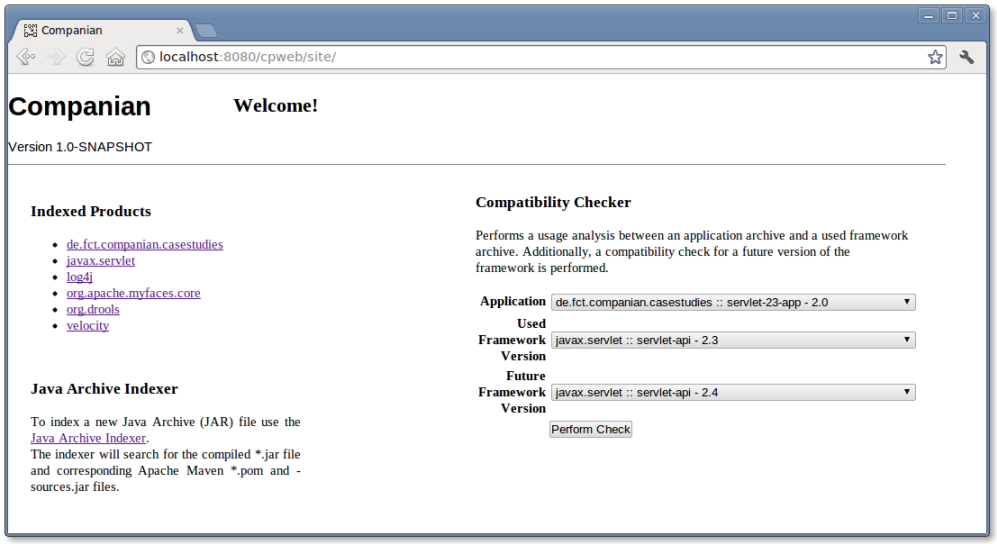


Abbildung 7.1: Companian - Startseite

Companian besteht aus drei Funktionsbausteinen. Die drei Bausteine ergeben sich aus der Zielsetzung Anwendungen und insbesondere Frameworks in Companian zu erfassen, Frameworks FDMM-basiert zu beschreiben und der Möglichkeit die automatische Kompatibilitätsprüfung durchzuführen.

- Der *Companian - Java Archive Indexer* importiert Java-Anwendungen und Frameworks auf Basis einer statischen Codeanalyse. Der Import ist Voraussetzung für die Erstellung einer Framework-Beschreibung und für die Durchführung der Kompatibilitätsprüfung.
- Der *Companian – Editor* ermöglicht die Erstellung von Framework-Beschreibungen.

- Der *Companian - Compatibility Checker* führt die Kompatibilitätsprüfung zwischen indexierten Anwendungen und Frameworks durch.

Die Softwarearchitektur von Companian implementiert die drei Funktionsbausteine in vier Komponenten.

- cpweb – Erzeugt die Benutzungsoberfläche und implementiert insbesondere den Companian-Editor.
- cpcompare – Implementiert die automatische Kompatibilitätsprüfung
- cpmodel – Implementiert das FDMM.
- cpanalyze – Implementiert die Indexierung und speichert die Informationen in einer Datenbank.

Abbildung 7.2 zeigt die Architektur von Companian und veranschaulicht den Zusammenhang der vier Komponenten cpweb, cpcompare, cpanalyze und cpmodel. Die Benutzungsoberfläche kommuniziert über eine RESTful [Fielding2000] API auf Basis des HTTP-Protokolls mit der cpweb Komponente. Die Komponente cpweb ist für die Generierung der Web-Oberfläche verantwortlich. Die Verwendung einer RESTful API ist ein technisches Detail der Umsetzung. Sie ermöglicht eine saubere Trennung zwischen den angebotenen Services und Ressourcen in cpweb und der Benutzungsoberfläche. Es ist eine serviceorientierte Technologie zur Implementierung von Web-Anwendungen.

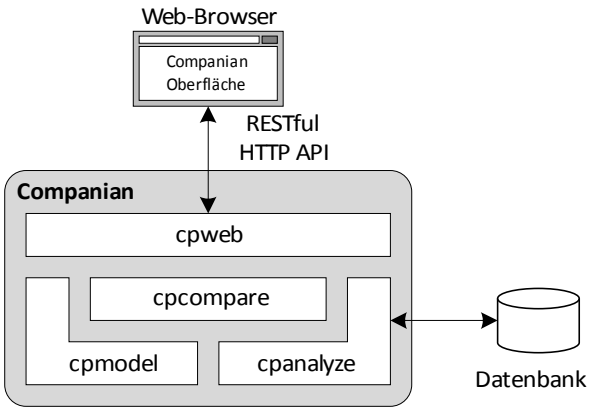


Abbildung 7.2: Companian Architektur

Über cpweb erhält die Oberfläche Zugriff zur Analysekomponente cpanalyze und zum FDMM, das in cpmodel implementiert ist. Über cpanalyze können die erfassten Informationen in einer MySQL-Datenbank[6] gespeichert werden. Die Komponente cpcompare implementiert die eigentliche Kompatibilitätsprüfung und verwendet hierzu die Informationen aus cpanalyze und cpmodel.

Companian ist für die Beschreibung von Frameworks ausgelegt, die in der Programmiersprache Java erstellt wurden. Dementsprechend bietet Companian Funktionen an, die spezielle technische Eigenschaften einer Java-Anwendung

6 http://www.mysql.com

ausnutzen. Insbesondere macht Companian die Annahme, dass ein Framework in Form eines Java-Archivs, kurz JAR, zur Verfügung steht.

Der erste Schritt zur Erstellung einer Framework-Beschreibung besteht darin, dass der Benutzer das Java-Archiv eines Frameworks in Companian importiert. Hierzu wird der »Java Archive Indexer« verwendet. Während des Imports wird das Framework einer Codeanalyse unterzogen und alle implementierten Klassen, Schnittstellen sowie deren Methoden in der Companian-Datenbank gespeichert.

Neben dem Java-Archiv des Frameworks müssen weitere Meta-Informationen in Form eines »Project Object Models« (POM) des Build-Management-Werkzeuges »Apache Maven«[7] vorliegen. Apache Maven ist ein Standardwerkzeug für die Kompilierung von Java-Programmen und dem Erzeugen von Java-Archiven. In einem POM speichert Apache Maven Meta-Informationen wie den Namen des Artefaktes, die Version, Lizenzbedingungen und Abhängigkeiten zu weiteren benötigten Java-Archiven. Diese Meta-Informationen werden für die Analyse in Companian vorausgesetzt.

Das FDMM ist im `cpmodel` von Companian implementiert. Es wurde als wiederverwendbare Bibliothek konzipiert, die auch in anderen Anwendungen wiederverwendet werden kann. Zur Speicherung eines FDM wird dieses in ein XML-Format serialisiert.

Die Implementierung der Parametrisierung des FDMM erforderte eine spezielle Behandlung der Meta-Modelle der einzelnen Parameter. Die Klassen der Parameter wurden als Java-Schnittstellentypen (engl. interfaces) implementiert. Damit ist in Companian das Meta-Modell eines Parameters nicht direkt instantiierbar. Stattdessen implementieren Adapter die Schnittstellen des Parameters. Über die Adapter werden die Meta-Modelle konkreter Sprachen an die Parameter gebunden. Für jeden Parameter des FDMM und jede konkrete Sprache wurde ein Adapter implementiert. Zur Veranschaulichung betrachten wir die Implementierung eines Adapters für UML-Aktivitätendiagramme in Abbildung 7.3.

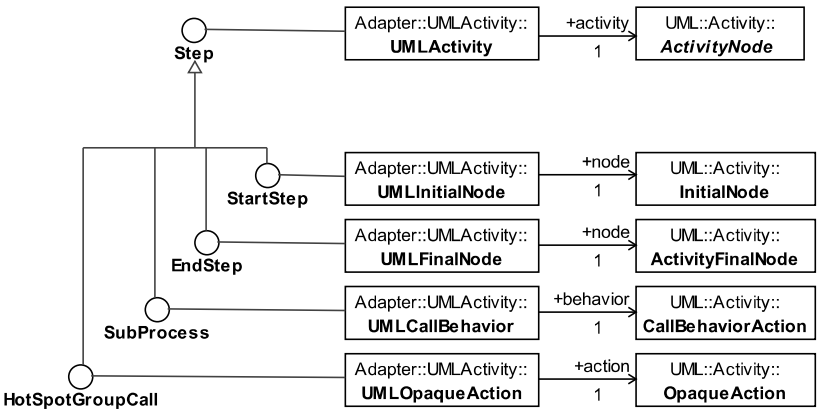


Abbildung 7.3: FDMM Adapter für UML::Activity

7 http://maven.apache.org

Die Klassen der BehaviorDL wurden wie erläutert als Schnittstellen implementiert. So entstehen die Schnittstellen Step, StartStep, EndStep usw. Für jede Schnittstelle gibt es eine implementierende Adapterklasse. Der Adapter implementiert so die geforderte Struktur der BehaviorDL und realisiert gleichzeitig die Bindung an die entsprechenden Klassen des UML::Activity Meta-Modells.

Wird zum Beispiel in einem FDM auf einen Step der BehaviorDL verwiesen, kann an diese Stelle der Adapter UMLActivity treten, der den Step implizit in einen ActivityNode übersetzt. Diese implizite Übersetzung ist eine Möglichkeit die Parameterersetzung zu realisieren. Für eine FDMM-basierte Anwendung sind die Adapter transparent, so dass die Anwendung von der Anbindung an UML-Aktivitätendiagramme keine Kenntnis hat.

Mit dieser Technik wurden alle Parameter des FDMM implementiert. So werden Instanzen der konkreten Sprache in Instanzen des Meta-Modells des Parameters überführt. Auf diese Weise ermöglicht der Adapter einen transparenten Zugriff auf die Instanzen einer konkreten Sprache.

Um die Arbeit der Erstellung einer Framework-Beschreibung zu erleichtern, bietet Companian eine Import-Funktion für eine bereits vorliegende API-Dokumentation eines Frameworks an. Gemäß dem FDMM wird eine API-Dokumentation über die APIDL spezifiziert. In Abschnitt 6.3.3 haben wir gesehen wie wir das JavaDoc-Format an den Meta-Modell-Parameter der APIDL binden, so dass wir direkt die API-Dokumentation im JavaDoc-Format in Companian integrieren können. Der Import einer bereits existierenden API-Dokumentation erfolgt durch cpanalyze mit dem »*Java Archive Indexer*«. Hierzu muss zum Java-Archiv zusätzlich das Quellcode-Archiv des Frameworks vorliegen. Die API-Dokumentation wird direkt aus den Kommentaren im Quellcode generiert und ebenfalls in Companian gespeichert. Nach Import des Frameworks kann die eigentliche Framework-Beschreibung im Companian-Editor angelegt werden.

Sind für zwei Versionen eines Frameworks entsprechende Framework-Beschreibung angelegt, kann die Kompatibilitätsprüfung durchgeführt werden. Wie in Kapitel 3 beschrieben besteht die automatische Kompatibilitätsprüfung aus drei Schritten: *Benutzungsanalyse*, *Differenzanalyse* und *Kompatibilitätsanalyse*. Wir zeigen die Umsetzung des Ablaufs dieser drei Schritte in Abbildung 7.4.

Die Benutzungsanalyse wird in Companian in der Komponente cpanalyze implementiert. Sie erhält als Eingabe den Bytecode eine Java-Anwendung, die das betrachtete Framework in Version 1 (FW_1) benutzt und eine Beschreibung von FW_1 als FDM. Das Ergebnis der Benutzungsanalyse ist die Menge aller von der Anwendung benutzten Erweiterungspunkte des Frameworks. Die Details dieser Analyse beschreiben wir in Abschnitt 7.2.

Die Differenz- und Kompatibilitätsanalyse werden in Companian in der Komponente cpcompare implementiert. Die Implementierung beider Analysen wurde in einer Komponente zusammengefasst, da beide funktional eng miteinander verbunden sind.

Die Differenzanalyse bestimmt die Unterschiede zwischen den FDMs der Frameworks FW_1 und FW_2. Das Ergebnis ist die Menge aller Unterschiede, die den Veränderungen am Framework zwischen Version 1 und 2 entsprechen. Wir beschreiben die Differenzanalyse in Companian in Abschnitt 7.3.

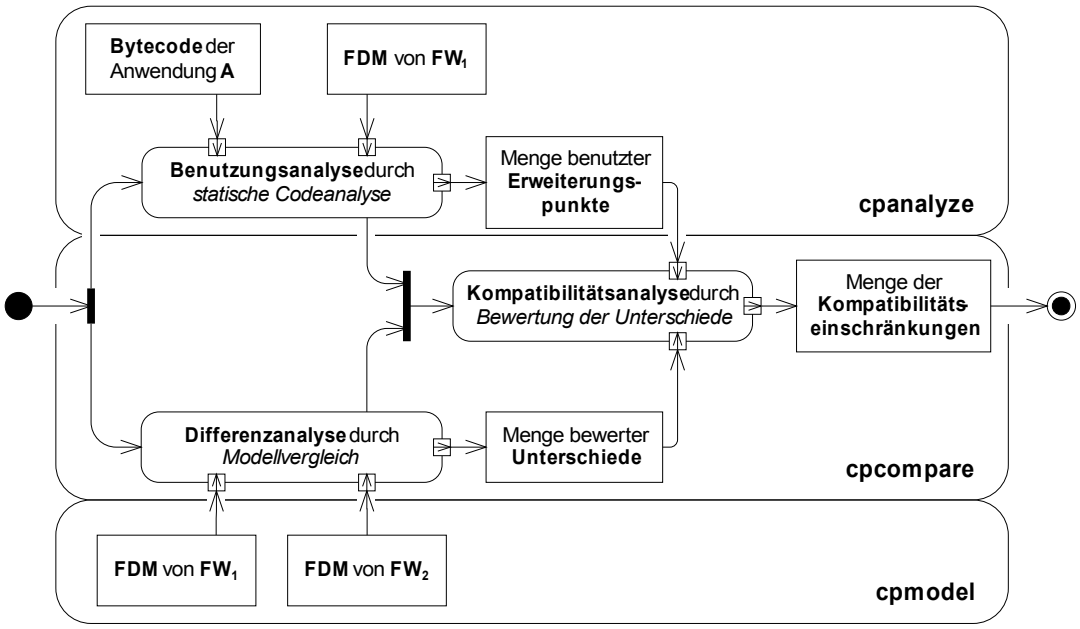


Abbildung 7.4: Umsetzung der Kompatibilitätsprüfung in Companian

Die abschließende Kompatibilitätsanalyse nimmt die Menge der verwendeten Erweiterungspunkte und die Menge der Unterschiede als Eingabe und filtert die relevanten Unterschiede im Hinblick auf mögliche Inkompatibilitäten. Wir widmen uns den Details in Abschnitt 7.4. Die Menge der Kompatibilitäts-einschränkungen ist das Ergebnis der Kompatibilitätsanalyse und damit auch das Ergebnis der automatischen Kompatibilitätsprüfung.

In den nachfolgenden Abschnitten werden wir auf die Details der Umsetzung der einzelnen Analyseschritte eingehen. Abschließend geben wir in Abschnitt 7.5 ein Beispiel für eine Kompatibilitätsprüfung, das den Mehrwert dieses Verfahrens verdeutlichen wird.

7.2 Umsetzung der Benutzungsanalyse

Die Benutzungsanalyse ermittelt welche Erweiterungspunkte (HotSpots) von einer Anwendung verwendet werden. Im FDMM definiert eine CodingUnit, welche Hook-Methoden (HookCalls) für diese CodingUnit zu implementieren sind. Jede CodingUnit ist dabei genau einem HotSpot zugeordnet. Den Zusammenhang zwischen HotSpot, CodingUnit und HookCalls im FDMM zeigt einmal die Abbildung 6.10 und die Abbildung 6.11.

Mit einer statischen Codeanalyse können wir bestimmen, ob eine Anwendung die von einer CodingUnit geforderten Hook-Methoden implementiert. Haben wir die CodingUnits bestimmt, wissen wir gleichzeitig welche HotSpots von der Anwendung benutzt werden.

Die Benutzungsanalyse ist in Companian innerhalb der cpanalyze Komponente implementiert. Das Vorgehen bei der Umsetzung der Benutzungsanalyse

zeigen wir in Abbildung 7.5. Es ist eine Verfeinerung des Konzeptes zur Benutzungsanalyse in Abbildung 3.2 aus Abschnitt 3.1.

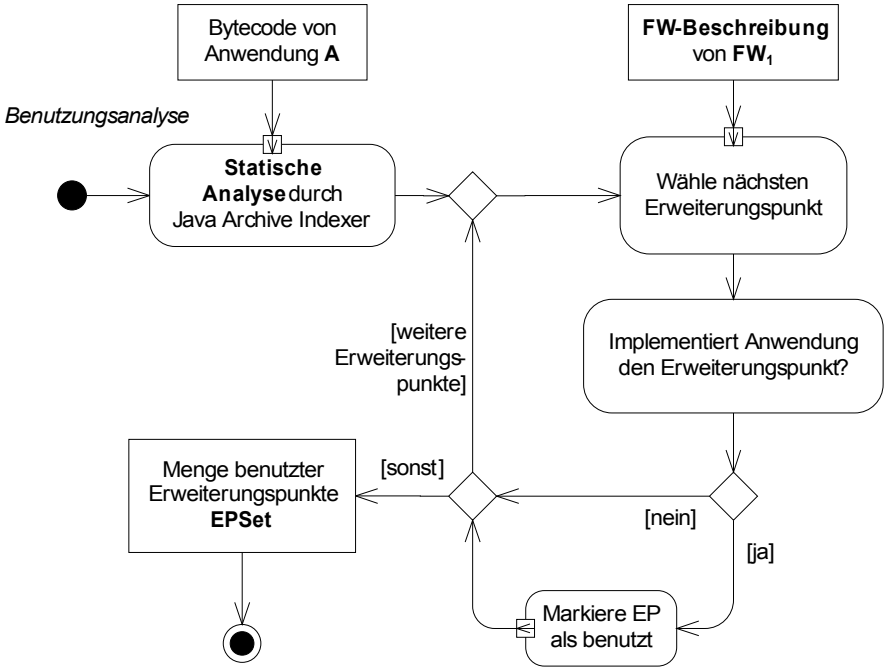


Abbildung 7.5: Umsetzung der Benutzungsanalyse in Companian

Die zentrale Frage innerhalb des Vorgehens in Abbildung 7.5 ist, ob die Anwendung einen bestimmten Erweiterungspunkt implementiert. Auf Basis der statischen Codeanalyse können wir diese Frage nicht immer eindeutig beantworten. Stattdessen berechnen wir einen so genannten Überdeckungswert, der uns bei der Beantwortung der Frage hilft.

Ein HotSpot kann aus mehr als einer CodingUnit bestehen. CodingUnits wiederum definieren eine Menge von HookCalls. Durch Ausnutzung von Vererbungsbeziehungen muss eine Anwendung nicht alle definierten HookCalls implementieren, um einen HotSpot zu benutzen. Daher liefert die Benutzungsanalyse einen Überdeckungswert, der angibt, zu welchem Anteil eine Anwendung einen bestimmten HotSpot implementiert.

Der Überdeckungswert bestimmt, wie viele HookCalls einer CodingUnit von einer Klasse der Anwendung implementiert werden. Werden alle HookCalls einer CodingUnit von einer einzigen Klasse der Anwendung implementiert, erhalten wir als Überdeckungswert 1.0.

Unseren Algorithmus zur Berechnung des Überdeckungswertes (`coverage`) zeigen wir in Listing 7.1.

Als Eingabe für den Algorithmus benötigen wir die Framework-Beschreibung (`fdm`) der aktuell verwendeten Framework-Version sowie den Programmcode der Anwendung selbst. Der Algorithmus bestimmt für jeden HotSpot `hs` und jede CodingUnit `cu` die zu implementierenden HookCalls `hc` für die jeweilige CodingUnit `cu`.

Listing 7.1: Algorithmus der Benutzungsanalyse

```
1  foreach HotSpot hs in fdm
2    foreach CodingUnit cu in hs.units
3      foreach HookCall hc in cu.hooks
4        set ic = computeClassesImplementing(hc)
5        foreach Class c in ic
6          c.implementedHooks.add(hc)
7      foreach Class c where c.implementedHooks.size > 0
8        c.coverage = c.implementedHooks.size / cu.hooks.size
```

Für jeden HookCall `hc` wird in Zeile 4 berechnet, welche Klassen eine Methode implementieren, deren Signatur mit der Signatur der `hc` übereinstimmt. Für jede dieser Klassen (`ic`) merken wir uns, welchen `hc` diese Klasse implementiert (Zeilen 5+6). Abschließend berechnen wir den Überdeckungswert für jede Klasse, indem für jede Klasse, die HookCalls implementiert, die Anzahl der implementierten HookCalls durch die Anzahl der erwarteten HookCalls für die CodingUnit (`cu`) geteilt wird (Zeilen 7+8).

Nach dieser Analyse haben wir für jede CodingUnit eines HotSpots eine Menge mit Klassen und deren Überdeckungswert ermittelt. Mit diesen Grundlagen können wir die Benutzung eines Erweiterungspunktes durch eine Anwendung wie folgt festlegen:

> *Ein Erweiterungspunkt (`HotSpot`) wird genau dann von einer Anwendung benutzt, wenn in der Anwendung für jede `CodingUnit` des Erweiterungspunktes mindestens eine Klasse existiert, deren Überdeckungswert größer einem benutzerdefinierten Schwellwert ist.*

Nachdem wir den Algorithmus zur Benutzungsanalyse auf Pseudocode-Ebene erläutert haben, betrachten wir nun seine konkrete Umsetzung in Companian. Zur Durchführung der Benutzungsanalyse wird die Anwendung in Companian importiert und dabei einer statischen Codeanalyse unterzogen. Im vorherigen Abschnitt haben wir bereits den Import von Frameworks in Companian mit dem »Java Archive Indexer« vorgestellt. Diesen verwenden wir auch für den Import von Anwendungen.

Die durchgeführte Codeanalyse während des Imports basiert auf dem Open-Source-Werkzeug »Dependency Finder«[8]. Der Dependency Finder enthält mit der Komponente des »Dependency Extractors«[9] ein Werkzeug zur statischen Codeanalyse. Dieses Werkzeug ist selber ein Framework, so dass es durch Implementierung eigener Erweiterungen an die Anforderungen in Companian angepasst wurde.

Ursprünglich ist der Dependency Extractor dazu gedacht Code-Abhängigkeiten zwischen Java-Paketen zu analysieren. Wir verwenden diese Fähigkeit, um die Klassen und Methoden einer Anwendung zu analysieren.Während des Imports speichert Companian alle Klassen und Schnittstellen sowie deren Methoden in der Companian-Datenbank. Wir bezeichnen diesen Vorgang als

8 http://depfind.sourceforge.net/
9 http://depfind.sourceforge.net/Components.html#extractor

Indexierung. Als Ergebnis ist die Anwendung in einem Datenbankschema abgelegt. Das verwendete Datenbankschema zeigen wir als Entity-Relationship (ER) Diagramm in Abbildung 7.6.

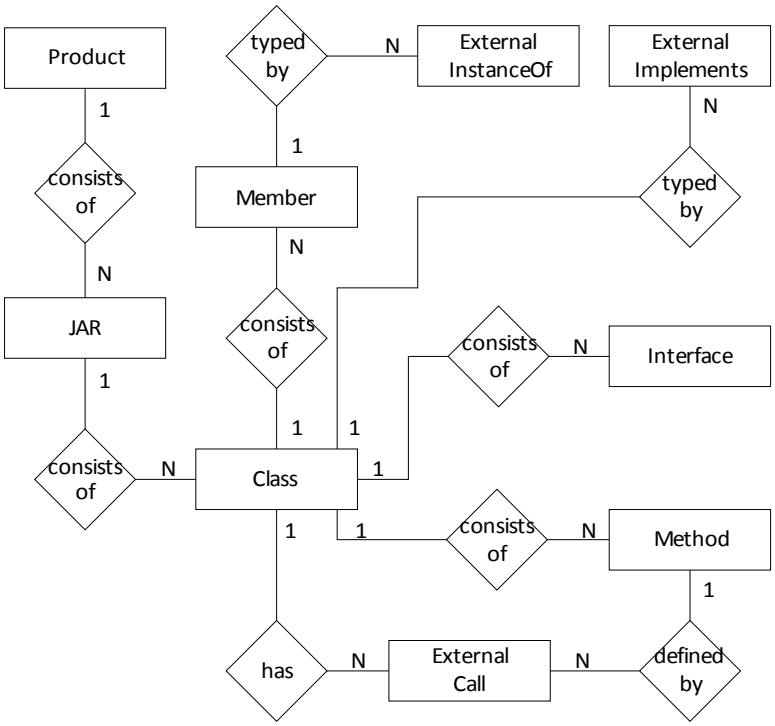


Abbildung 7.6: Entity-Relationship Modell der Companian Datenbank

Das ER-Modell beschreibt, dass zu einem Produkt (Product), wie einem Framework, eine Menge von Java-Archiven (JAR) zugeordnet sind. Ein JAR wiederum besteht aus einer Menge von Java-Klassen (Class). Zu jeder Klasse indexieren wir deren Variablen (Member), Methoden (Method) und Schnittstellen (Interface).

Neben den implementierten Klassen und Schnittstellen in einem JAR indexieren wir außerdem Informationen über externe Beziehungen zu anderen Klassen, die nicht innerhalb des JAR aufgelöst werden können. Das Prinzip ist, dass jede Verwendung von Klassen, die nicht im JAR enthalten sind, als externe Beziehung indexiert wird. Auf dieser Grundlage lassen sich Code-Beziehungen zwischen JARs analysieren.

Eine External InstanceOf im ER-Modell beschreibt, dass der Typ eines Member in einer externen Klasse definiert ist. Ebenso verhält es sich mit External Implements, die einen externen Typ für eine Klasse beschreibt. Außerdem werden die externen Methodenaufrufe (External Call) einer Klasse indexiert.

Der Vorteil eine Anwendung im gegebenen Datenbankschema zu indexieren liegt darin, dass wir mittels einer Datenbankabfrage ermitteln können, welche Klassen der Anwendung zum Beispiel bestimmte Methoden implementieren. Wir werden dies zur Bestimmung implementierter HookCalls wie folgt ausnutzen.

Durch die Indexierung haben wir in der Companian-Datenbank alle Informationen zur Bestimmung von `computeClassesImplementing(hc)` aus Listing 7.1 hinterlegt. Wir bestimmen die Klassen, die eine bestimmte Hook-Methode implementieren mit Hilfe einer SQL-Abfrage, die wir in Listing 7.2 zeigen.

Listing 7.2: SQL-Abfrage zur Bestimmung von `computeClassesImplementing(HookCall)`

```
1  SELECT m.methodId, m.accessFlags, m.constructor,
   m.returnType, m.signature, m.classId
2  FROM Method m
3  JOIN Class c ON (m.classId = c.classId)
4  JOIN JAR j ON (c.jarId = j.jarId)
5  WHERE m.signature=? AND j.jarId=?
```

Die SQL-Abfrage ermittelt alle Klassen eines Java Archivs, die eine Methode mit einer bestimmten Hook-Methoden-Signatur implementieren. Der Vergleich basiert auf einem String-Vergleich der Methoden-Signaturen.

Die Identifizierung von Erweiterungspunkten funktioniert, wenn alle Hook-Methoden eines Erweiterungspunktes in derselben Klasse oder innerhalb der Klassenhierarchie dieser Klasse implementiert wurden. Dies ist der typische Fall für die Implementierung von Erweiterungspunkten eines Java-Frameworks. Wie sich in den in Abschnitt 6.4 vorgestellten Fallstudien gezeigt hat, ließen sich mit diesem Vorgehen die benutzten Erweiterungspunkte eines Frameworks identifizieren. Die Wahrscheinlichkeit, dass eine Klasse Methoden implementiert, die exakt den erwarteten Methoden-Signaturen von Hook-Methoden entsprechen und dabei keinen Erweiterungspunkt implementieren, ist als gering einzustufen. In den durchgeführten Fallstudien konnte dieser Fall nicht beobachtet werden.

Problematisch ist die Implementierung von Hook-Methoden über mehrere assoziierte Klassen hinweg. In einem solchen Fall werden die Hook-Methoden nicht unbedingt in einer einzigen Klasse implementiert, sondern durch Delegation auf verschiedene Klassen verteilt. Den korrekten Zusammenhang dieser Klassen zu erkennen und dabei den implementierten Erweiterungspunkt zu identifizieren, ist mit der in Companian umgesetzten Benutzungsanalyse nicht möglich.

Eine weitere Einschränkung der Verwendung einer statischen Codeanalyse liegt darin, dass man nicht entscheiden kann, ob ein Erweiterungspunkt zur Laufzeit wirklich von der Anwendung verwendet wird. Hierzu wäre eine dynamische Codeanalyse notwendig, die das tatsächliche Verhalten bei Ausführung bestimmt. Es ist möglich, dass ein statischer Zusammenhang existiert, der bei Ausführung jedoch nicht zum Tragen kommt. In solchen Fällen würde die Kompatibilitätsprüfung falsch-positive Aussagen treffen, da sie immer davon ausgeht, dass ein Erweiterungspunkt verwendet wird, wenn es eine statische Abhängigkeit gibt.

Dieses theoretische Problem sollte in der Praxis jedoch nicht auftreten, da es keine statischen Abhängigkeiten geben sollte, wenn die Erweiterung nicht verwendet wird. Eine nicht verwendete statische Abhängigkeit bedeutet, dass die

Anwendung toten Code enthält. Toter Code ist jedoch ein Anti-Pattern im Software-Entwurf, das vermieden werden sollte. Toter Code kann zu unentdeckten Fehlern führen, weil niemand damit rechnet, dass dieser Code sich in der Anwendung befindet.

Mit dieser Argumentation denken wir, dass die Beschränkung auf eine statische Codeanalyse ein valides Vorgehen ist. In den durchgeführten Fallstudien konnten keine Hinweise auf Probleme mit diesem Vorgehen gefunden werden.

Das Ergebnis der Benutzungsanalyse liefert eine Menge aller benutzten Erweiterungspunkte des Frameworks. In Abbildung 7.7 sehen wir ein Bildschirmfoto der Benutzungsanalyse in Companian. Es zeigt die Analyse einer Beispielanwendung, die das Java Servlet Framework verwendet. Die Analyse ergibt, dass die Anwendung die Erweiterungspunkte (Hotspots) »Generic Servlet«, »Session Listener« und »Simple HTTP Servlet« implementiert. Für die ersten beiden Erweiterungspunkte wurde ein Überdeckungswert (Coverage) von 1.0 berechnet. Für das »Simple HTTP Servlet« liegt der Überdeckungswert hingegen bei etwa 0.2. Außerdem werden jeweils die implementierenden Klassen angegeben.

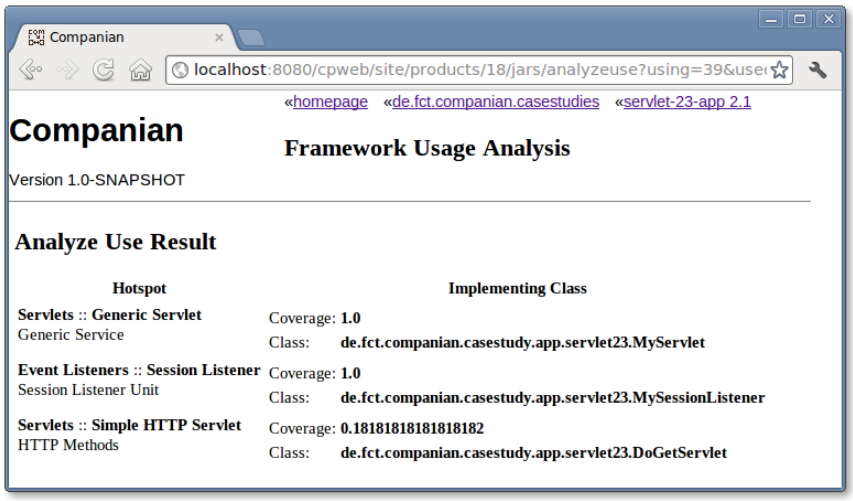


Abbildung 7.7: Companian - Benutzungsanalyse

Die Menge benutzter Erweiterungspunkte ist eine Eingabe für die Kompatibilitätsanalyse, die wir in Abschnitt 7.4 erläutern werden. Die zweite Eingabe für die Kompatibilitätsanalyse liefert die Differenzanalyse aus Abschnitt 7.3.

7.3 Umsetzung der Differenzanalyse

Die Differenzanalyse berechnet die Veränderungen zwischen zwei Versionen einer FDMM-basierten Framework-Beschreibung. Sie basiert auf dem Modellvergleich zweier FDMs. Die Umsetzung des Differenzanalyseprozesses zeigen wir in Abbildung 7.8.

Wir stützen unser Vorgehen beim Modellvergleich auf die Arbeiten von Brun und Pierantonio [Brun2008], die als Grundlage für das Werkzeug »EMF Com-

pare«[10] dienten. Gemäß dieser Arbeiten erzeugen wir im ersten Schritt in Abbildung 7.8 zunächst ein *Matching* zwischen den beiden Modellen, um die Korrespondenzen zu bestimmen. Die Korrespondenzen bestimmen, welche Objekte sich in beiden Modellen entsprechen. Hierauf aufbauend können mit dem *Differencing* die Unterschiede ermittelt werden.

Ein solcher Vergleich betrachtet drei mögliche Unterschiede, die im Objektmodell des FDMs erkannt werden können: *Neuanlage*, *Änderung* und *Löschung* von Objekten. Diese drei Arten von Unterschieden werden im letzten Schritt im Hinblick auf die Kompatibilität *bewertet*, so dass im Ergebnis die Menge bewerteter Unterschiede ausgegeben wird.

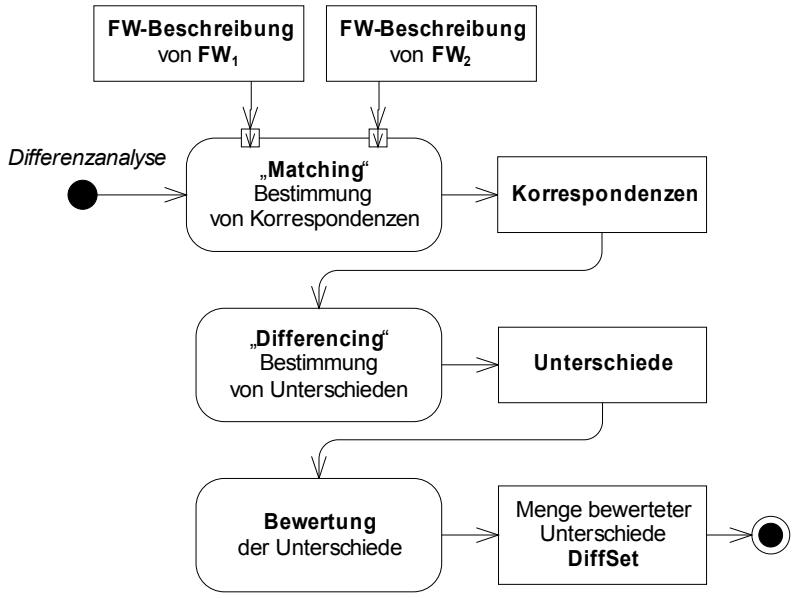


Abbildung 7.8: Umsetzung der Differenzanalyse in Companian

Die *Neuanlage* eines Objektes im FDM hat für sich genommen keine Auswirkungen im Hinblick auf die Kompatibilität. Stattdessen muss die Neuanlage im Kontext anderer Objekte betrachtet werden. Betrachten wir hierzu ein Beispiel:

Es wird ein neues Objekt vom Typ HookCall erkannt. Diese Neuerung wird nun aus unterschiedlichen Blickwinkeln analysiert, nämlich zum einen aus Sicht eines verlinkten HotSpots und aus Sicht von verlinkten HookProtocols. Je nachdem ob ein verlinkter HotSpot bzw. ein HookProtocol in der alten Framework-Version benutzt wurde, hat die Neuanlage eines HookCalls Auswirkungen auf die Kompatibilität und muss entsprechend bewertet werden.

Das Erkennen einer *Änderung* bezieht sich auf das Erkennen von veränderten Links der Objektstruktur im FDM. Ein Objekt gilt als verändert, wenn sich die Objektstruktur der Objekte verändert hat, aus denen das Objekt komponiert wird. Betrachten wir auch hier ein Beispiel:

Eine BindingCapability kann aus verschiedenen BindingDeclarations bestehen. Wenn zum Beispiel ein neue BindingDeclaration hinzugefügt wurde, so impliziert

10 http://www.eclipse.org/emf/compare/

diese Neuanlage eine Änderung der BindingCapability. Analog verhält es sich mit der Löschung von Elementen. Wird eine BindingDeclaration verändert, indem Neuanlagen oder Löschungen in der abhängigen Objektstruktur erfolgen, so ist damit auch die BindingCapability verändert worden. So werden Ereignisse wie Neuanlage oder Löschung im Kontext als Änderungen erkannt.

Wie Neuanlagen werden auch *Löschungen* im Kontext betrachtet. Die Löschung eines Objektes im FDM führt zur Löschung von Links zu Objekten im Kontext. Bei der Analyse werden Links betrachtet, die zuvor zwei Objekte miteinander verbanden. Das Fehlen der Verbindung kann häufig so interpretiert werden, dass hier Teile der Framework-Beschreibung entfernt wurden, weil die zugehörige Funktionalität nicht mehr vorhanden ist.

Wir wollen die Behandlung von Löschungen in einem Beispiel verdeutlichen. Einem HotSpot war in der alten Version des Frameworks ein bestimmter HookCall zugeordnet, der von einer Anwendung implementiert wurde. In der neuen Framework-Version ist diese Zuordnung nicht mehr gegeben, weil das Objekt, das den HookCall beschreibt, gelöscht wurde. Dies wird so interpretiert, dass der HotSpot eine Änderung erfahren hat und eine Kompatibilitätseinschränkung vorliegt, da für die Anwendung der Aufruf dieses alten HookCalls eventuell von Bedeutung war.

In Companian ist die Differenzanalyse in der cpcompare Komponente implementiert. Zur Definition der möglichen Inkompatibilitäten werden in cpcompare alle denkbaren Konstellationen von Unterschieden zwischen zwei FDMs betrachtet und in einem Kompatibilitätskatalog zusammengestellt. In der Aufstellung des Kompatibilitätskataloges wird die Kompatibilität so bewertet, als würde eine Anwendung genau den betrachteten Teil des Frameworks benutzen. Dann wird die Frage gestellt, ob eine Veränderung wie Neuanlage, Änderung oder Löschung Auswirkung auf die Kompatibilität haben kann. Wenn Auswirkungen denkbar sind, wird der Unterschied als Kompatibilitätseinschränkung bewertet.

Das Eintreten einer Veränderung wird im Kontext des betrachteten Objektes bewertet und auf ein Tupel bestehend aus Bewertung und Hinweistext abgebildet. Dies entspricht der Abbildung $\textit{Veränderung} \rightarrow \textit{Bewertung} \times \textit{Erläuterung}$. In Tabelle 7.1 erläutern wir die einzelnen Bestandteile dieser Zuordnung.

Tabelle 7.1: Bewertungsschema einer Veränderung zwischen zwei Frameworks

Veränderung	Eine der möglichen Veränderungen aus Neuanlage, Änderung oder Löschung.
Bewertung	Bei der Bewertung ist zu beachten ist, dass nicht jeder Unterschied zu einer Kompatibilitätseinschränkung führt. Zur Bewertung unterscheiden wir gemäß unserer Klassifikation in Kapitel 3 in • *Hinweise* auf Veränderungen • *Warnungen* vor potentiellen Kompatibilitätseinschränkungen • *Konflikte* durch gesicherte Kompatibilitätseinschränkungen
Erläuterung	Die Erläuterung beschreibt in natürlicher Sprache die Art der Kompatibilitätseinschränkung und welche Beschreibungselemente hiermit in Beziehung stehen. Die Erläuterungen werden automatisch generiert.

In Companian erhält man als Ergebnis der Differenzanalyse eine Menge mit Veränderungen sowie deren Bewertung und Erläuterung. Dabei stellt Companian teilweise die neuen und alten Werte der verglichenen Versionen gegenüber, um dem Benutzer die Möglichkeit zu geben, die Veränderung nachvollziehen zu können. In Abbildung 7.9 sehen wir einen Auszug aus der Differenzanalyse zwischen dem Java Servlet Framework in Version 2.3 und 2.4.

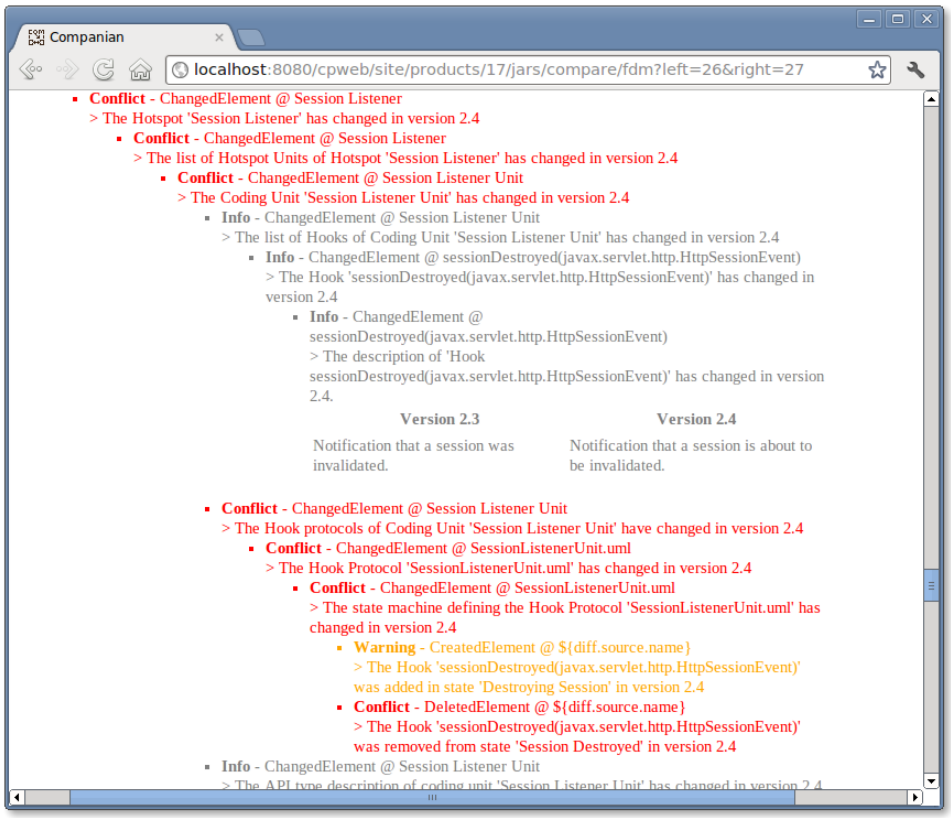


Abbildung 7.9: Companian - Ausschnitt aus einer Differenzanalyse

In Companian werden die Unterschiede gemäß ihrer Bewertung farblich markiert. Rote Unterschiede markieren Konflikte, Unterschiede in orange kennzeichnen Warnungen vor möglichen Inkompatibilitäten und graue Unterschiede sind Hinweise auf Veränderungen.

Die Menge der Unterschiede wird in Companian als Baum organisiert. Die Wurzel bilden Veränderungen der Framework-Beschreibung, auf der nächsten Ebene folgen Veränderungen an Erweiterungspunkten, dann von Teilen von Erweiterungspunkten und so weiter. Falls auf einer unteren Ebene eine Veränderung erkannt wurde, wird dies auf die darüber liegende Ebenen propagiert. Bei mehreren Änderungen wird immer die Veränderung mit der höheren Bewertung nach oben weitergereicht. In diesem Sinne ist die höchstmögliche Bewertung ein Konflikt, dann folgen Warnungen und schließlich Hinweise. So werden die Bewertungen von unten nach oben aufgebaut. Auf der obersten Ebene kann man so zum Beispiel aussagen, dass Veränderungen gefunden wurden, die zu einem Konflikt führen. Steigt man dann die Ebenen tiefer hinab, erkennt man, welche Veränderung die Ursache des Konflikts ist.

Einige Teile des FDMM sind für die Differenzanalyse nicht von belang, da man über ihre Veränderungen keine automatischen Schlüsse auf die Kompatibilität ziehen kann. So werden Attribute vom Typ String wie »name« oder »description« für die Analyse nicht berücksichtigt, da man keine Aussagen zur Kompatibilität treffen kann, wenn sich diese Attribute ändern. Der textuelle Inhalt dieser Attribute müsste interpretiert werden können, um Veränderungen hinsichtlich der Kompatibilität semantisch zu bewerten, was für einen beliebigen Text nicht möglich ist. Aus diesem Grund werden Veränderungen an textuellen Inhalten nur als Hinweise auf Veränderungen bewertet.

Für die ProtocolDL und die BehaviorDL ist die Prüfung auf Veränderungen hingegen komplexer. Für beide Sprachen starten wir eine Bisimulation. Dabei versuchen wir die zu vergleichenden Instanzen der Sprachen parallel auszuführen, um dabei Unterschiede zu erkennen. Wenn wir Unterschiede feststellen, bewerten und markieren wir diese.

Für die ProtocolDL haben wir eine Ausführungssemantik angelehnt an die Semantik von Zustandsautomaten definiert. Dies geht aus den beschriebenen Verhaltensmustern hervor. Die Bisimulation beginnt für Instanzen der ProtocolDL ausgehend vom Startzustand beider Zustandsautomaten. Die implementierte Bisimulation ist in der Lage einfache Veränderungen wie das Einfügen eines neuen Zustandes zu erkennen. Außerdem erkennen wir Veränderungen an den Zustandsübergängen. Die Bisimulation endet, wenn alle Zustandsübergänge der ursprünglichen ProtocolDL-Instanz besucht wurden. Für die Implementierung dieser Bisimulation werden Zustandsübergänge und Zustände als gleich angesehen, wenn sie dieselben Bezeichner tragen.

Die BehaviorDL verhält sich, entsprechend der Verhaltensmuster, analog zur Semantik von UML-Aktivitätendiagrammen. Für die Bisimulation versuchen wir parallel alle Aktivitäten der zu vergleichenden BehaviorDL-Instanzen zu erreichen. Dabei ist die implementierte Bisimulation in der Lage, eingefügte und gelöschte Aktivitäten zu erkennen. Aktivitäten werden als gleich angesehen, wenn die Bezeichner gleich sind. Erkannte Unterschiede werden bewertet und protokolliert.

Das Ergebnis der Differenzanalyse ist die Menge aller bewerteten Unterschiede, die sich auf Neuanlage, Änderung oder Löschung von Objekten der FDMs beziehen. Im nachfolgenden Abschnitt 7.4 werden wir betrachten, wie diese Unterschiede im Sinne einer Kompatibilitätsanalyse zu filtern sind.

7.4 Umsetzung der Kompatibilitätsanalyse

Die Differenzanalyse in Abschnitt 7.3 hat mit dem Modellvergleich die Vorarbeiten für die Kompatibilitätsanalyse geleistet. Die Kompatibilitätsanalyse nimmt die berechneten Unterschiede aus der Differenzanalyse und filtert diese Unterschiede gemäß Relevanz in Bezug auf die Benutzungsanalyse. Den unterliegenden Prozess zeigt Abbildung 7.10 und das Ergebnis der Umsetzung in Companian sehen wir Abbildung 7.11.

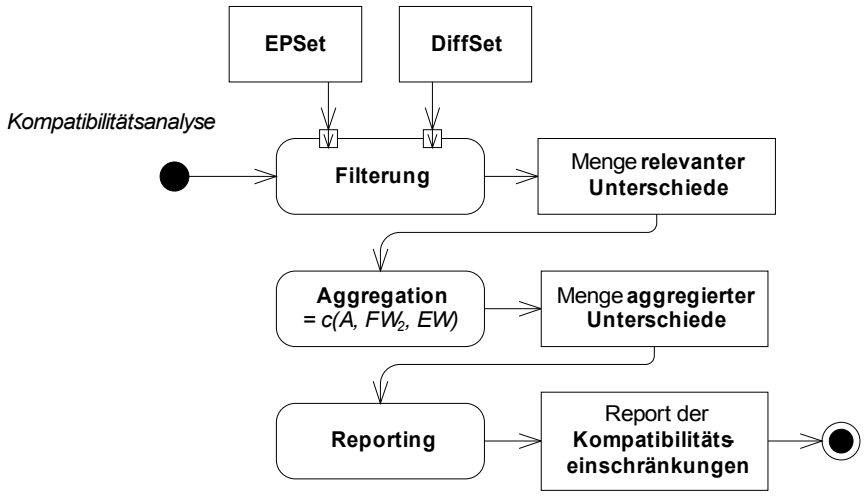


Abbildung 7.10: Umsetzung der Kompatibilitätsanalyse in Companian

Durch die *Filterung* im ersten Schritt werden diejenigen Unterschiede aus dem DiffSet verworfen, die nicht in Bezug zu benutzten Erweiterungspunkten aus dem EPSet stehen. Jedem Eintrag im DiffSet ist ein Erweiterungspunkt zugeordnet. Dies ist eine Eigenschaft des FDMMs. Auf diese Weise können wir diejenigen Unterschiede identifizieren, die relevant in Bezug auf die Benutzung sind. Die restlichen Unterschiede verwerfen wir.

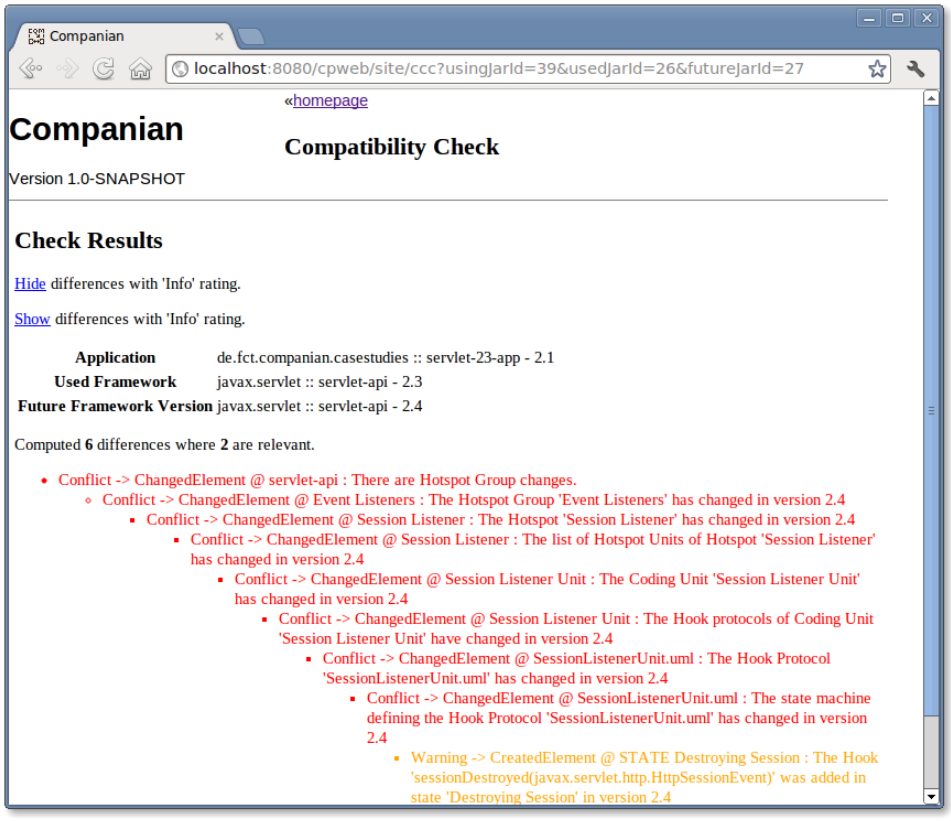


Abbildung 7.11: Companian - Kompatibilitätsanalyse

Nachdem wir aus der Gesamtmenge der Differenzen die nicht relevanten Unterschiede verworfen haben, müssen wir die Ergebnisse durch eine *Aggregation* neu zusammenfassen. Durch Verwerfen einiger Unterschiede kann es notwendig sein Bewertungen konsistent anzupassen.

Im abschließenden *Reporting* erzeugen wir ein Dokument, das die relevanten Unterschiede auflistet. Hierzu verwenden wir die Hinweistexte, die während der Differenzanalyse zu jedem Unterschied angelegt wurden.

In Companian wird das Ergebnis einer Kompatibilitätsanalyse, wie in Abbildung 7.11 zu sehen, vergleichbar zum Ergebnis der Differenzanalyse dargestellt. Der Unterschied besteht darin, dass die nicht relevanten Unterschiede nicht mehr angezeigt werden. Dies verkleinert die Menge zu betrachtender Unterschiede unter Umständen erheblich. Wir werden dies im nachfolgenden Abschnitt an einem konkreten Beispiel verdeutlichen.

Mit der Benutzungs-, der Differenz- und der Kompatibilitätsanalyse haben wir die Umsetzung der Kompatibilitätsprüfung, wie sie in dieser Arbeit konzeptioniert wurde, in Companian erfolgreich umgesetzt. Im nachfolgenden Abschnitt 7.5 wollen wir anhand eines Beispiels einer Kompatibilitätsprüfung erläutern, wie Benutzer mit Companian arbeiten.

7.5 Beispiel einer Kompatibilitätsprüfung

Wir wollen anhand eines Beispiels zeigen, wie mit Companian gearbeitet wird und welche Vorteile die automatische Kompatibilitätsprüfung konkret bietet. Hierzu betrachten wir das »Java Servlet Framework«. Dieses Framework wird in unzähligen Java-Web-Anwendungen verwendet, so dass dieses Framework ein realistisches Beispiel darstellt.

Als Voraussetzung für eine Kompatibilitätsprüfung müssen unterschiedliche Versionen des Java Servlet Frameworks FDMM-basiert mit Companian beschrieben werden. Der Companian-Editor, beschrieben in Abschnitt 6.4, stellt hierfür alle benötigten Funktionen zur Verfügung. Um mehrere Versionen eines Frameworks zu beschreiben bietet Companian die Möglichkeit das FDM einer Framework-Version zu kopieren und als Grundlage zur Dokumentation der nächsten Version zu verwenden. So müssen nur diejenigen Stellen angepasst werden, an denen sich das Framework verändert hat.

Für unser Beispiel haben wir aus den Spezifikationsdokumenten des Java Servlet Frameworks mit dem Companian-Editor eine Framework-Beschreibung für die Versionen 2.3 und 2.4 erstellt. Wir können somit Anwendungen analysieren, die Version 2.3 verwendet haben und auf Version 2.4 aktualisieren wollen.

In diesem Beispiel betrachten wir eine fiktive Beispielanwendung, die das Java Servlet Framework Version 2.3 benutzt. Die Anwendung implementiert zwei Servlets sowie einen Listener, der auf Veränderungen von Benutzersitzungen (Sessions) reagiert. Wir wollen prüfen, welche Kompatibilitätsprobleme bei Aktualisierung auf das Java Servlet Framework Version 2.4 zu erwarten sind.

Um die Vorteile der Kompatibilitätsprüfung mit Companian deutlich zu machen, analysieren wir zunächst die Veränderungen zwischen den Framework-Versionen auf API-Ebene. Eine solche Analyse entspricht dem Stand der Technik gemäß unserer Abgrenzung zu verwandten Arbeiten in Abschnitt 2.5. Companian bietet für diese Analyse eine API-Vergleichsfunktion (»API Compare«). Der API-Vergleich erkennt neue und gelöschte Klassen, sowie neue und gelöschte Methoden der API eines Frameworks. In Abbildung 7.12 sehen wir den API-Vergleich zwischen Java Servlet Version 2.3 und Version 2.4.

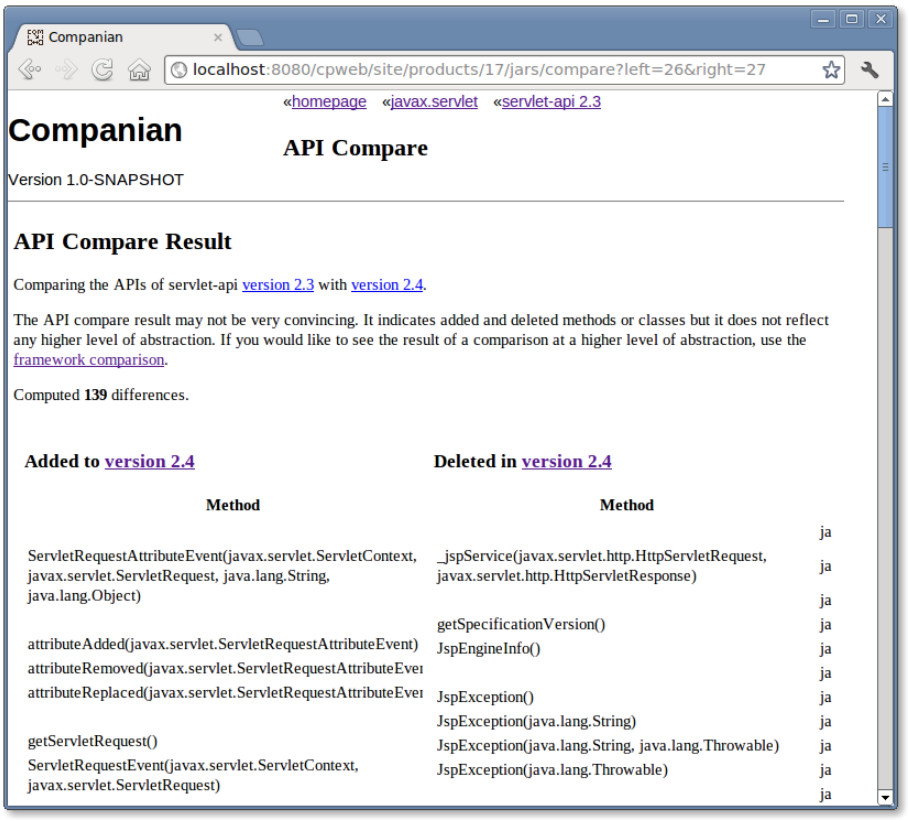


Abbildung 7.12: Companian - API-Vergleich

Die Analyse hat *139* Veränderungen zwischen beiden Versionen berechnet. Den größten Teil dieser Veränderungen betreffen gelöschte Klassen. Die Klassen wurden gelöscht, da sie nicht direkt zum Framework gehörten und daher im Zuge einer Refaktorisierung des Frameworks verschoben wurden. Es ist jedoch nicht ersichtlich, ob diese Klassen für die Anwendung relevant waren.

Betrachten wir nun die Anzahl der Unterschiede, die sich beim Vergleich der beiden Framework-Beschreibungen ergeben. In Abbildung 7.9 haben wir bereits ein Bildschirmfoto einer solchen Differenzanalyse in Companian gezeigt. Für das Servlet-Beispiel berechnet die Differenzanalyse *27* Veränderungen zwischen den Framework-Versionen. In diesen Veränderungen sind sowohl Hinweise als auch Warnungen und erkannte Konflikte enthalten. Zieht man von diesen Veränderungen diejenigen ab, die nur als Hinweise für Verän-

derungen bewertet wurden, bleiben *5* Veränderungen mit Bewertung »Warnung« oder »Konflikt« übrig, die wir als relevant ansehen.

Führen wir schließlich eine Kompatibilitätsprüfung gemäß unserem Ansatz durch, können wir die Unterschiede auf Basis des Ergebnisses der Benutzungsanalyse weiter einschränken. Die Benutzungsanalyse, wie wir sie in Abbildung 7.7 gezeigt haben, ergibt für unsere Beispielanwendung, dass die folgenden Erweiterungspunkte benutzt werden:

- »Generic Servlet« mit Überdeckungswert 1.0
- »Simple HTTP Servlet« mit Überdeckungswert 0.18
- »Session Listener« mit Überdeckungswert 1.0

Dies entspricht dem erwarteten Ergebnis, denn die Beispielanwendung implementiert zwei Servlets auf unterschiedliche Art und Weise sowie einen Listener. Für die Kompatibilitätsanalyse müssen wir festlegen ab welchem Schwellwert für den Überdeckungswert wir einen Erweiterungspunkt als benutzt ansehen. Im Beispiel wählen wir als Schwellwert 0.1. Es ist ratsam einen niedrigen Schwellwert zu wählen, da dies im Zweifel lediglich die falsch-positiven Treffer erhöht, aber es vermindert die Gefahr, dass die Kompatibilitätsprüfung die Benutzung eines Erweiterungspunktes übersieht.

Die Kompatibilitätsprüfung für die Beispielanwendung, wie wir sie in Abbildung 7.11 zeigen, berechnet *6* Unterschiede. Streichen wir hiervon wieder diejenigen Unterschiede, die lediglich als Hinweise bewertet wurden, bleiben nunmehr *2* relevante Unterschiede mit Bewertung »Konflikt« oder »Warnung«. Wir wollen diese Zahlen in Tabelle 7.2 gegenüberstellen.

Tabelle 7.2: Gegenüberstellung von Analyseverfahren am Servlet-Beispiel

	API-Vergleich	FDM Differenz-anaylse	Kompatibilitäts-prüfung
Unterschiede Gesamt	**139**	27	6
davon Konflikte und Warnungen	-	5	**2**

Aus Tabelle 7.2 ist zu ersehen, dass der API-Vergleich im Gegensatz zur Differenzanalyse und zur Kompatibilitätsprüfung sehr viele Treffer liefert. Die Differenzanalyse reduziert diese Zahl bereits auf ein Fünftel. Erst die Kompatibilitätsprüfung weist den Benutzer jedoch direkt auf die wenigen Veränderungen am Framework hin, die für die Anwendung von Interesse sind.

Betrachten wir die durch die Kompatibilitätsprüfung ermittelten relevanten Veränderungen genauer, ergibt sich ein zweiter entscheidender Vorteil der Kompatibilitätsprüfung. Die Veränderungen beziehen sich auf das Hook-Protokoll der »Session Listener Unit«. Betrachten wir einen Auszug aus dem Ergebnis der Kompatibilitätsprüfung in Listing 7.3.

Listing 7.3: Auszug aus einer Kompatibilitätsprüfung

- The Hotspot 'Session Listener' has changed in version 2.4
- The Coding Unit 'Session Listener Unit' has changed in version 2.4
- The Hook Protocol 'SessionListenerUnit.uml' has changed in version 2.4
  - The Hook 'sessionDestroyed(javax.servlet.http.HttpSessionEvent)' was removed from state 'Session Destroyed' in version 2.4
  - The Hook 'sessionDestroyed(javax.servlet.http.HttpSessionEvent)' was added in state 'Destroying Session' in version 2.4

Zusätzlich veranschaulichen wir in Abbildung 7.13 die erkannte Veränderung grafisch. Diese grafische Aufarbeitung ist jedoch (noch) kein Teil der Implementierung in Companian und wurde nur für diese Arbeit erstellt.

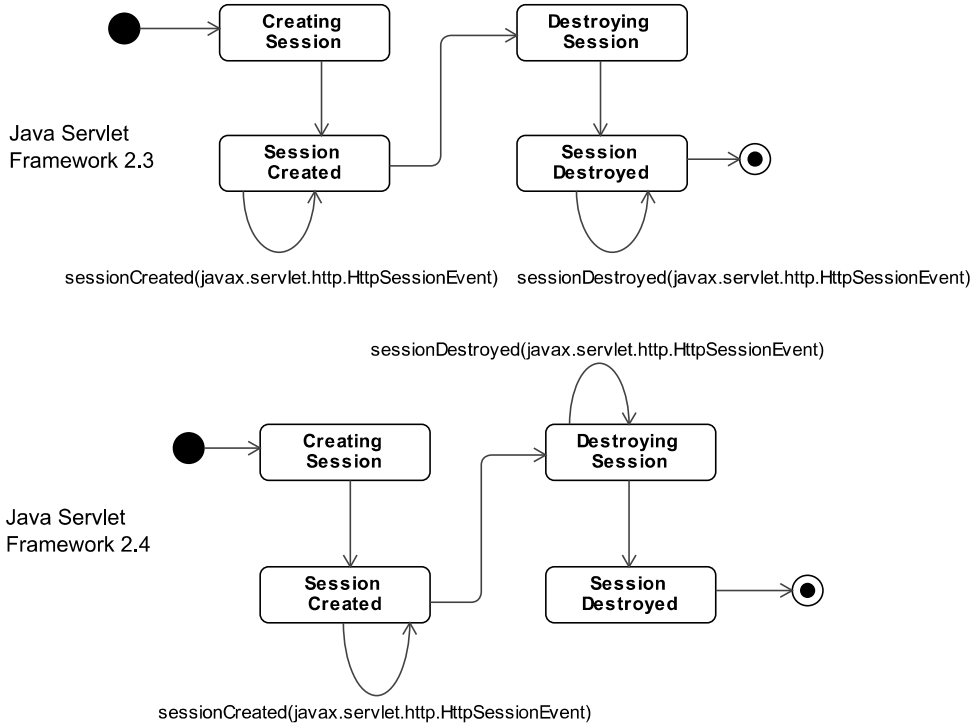


Abbildung 7.13: Unterschied im Hook-Protokoll der »Session Lister Unit« zwischen Version 2.3 und 2.4 des Java Servlet Frameworks

Aus Listing 7.3 erkennen wir, dass die Hook-Methode `sessionDestroyed()` innerhalb des Hook-Protokolls `SessionListenerUnit.uml` verschoben wurde. Zunächst ist die Hook-Methode im Zustand »Session Destroyed« entfernt und dann im Zustand »Destroying Session« hinzugefügt worden. Es hat sich somit der Zeitpunkt des Aufrufes der Hook-Methode verändert.

Bei Betrachtung der grafischen Repräsentation ist die Veränderung am Hook-Protokoll in Abbildung 7.13 offensichtlich. Mit der Änderung des Zeitpunktes, an dem die Hook-Methode aufgerufen wird, hat sich deren Semantik verändert. Anwendungen, die in Version 2.3 darauf beruhten, dass die Hook-Methode

`sessionDestroyed()` aufgerufen wird *bevor* die Session zerstört ist, könnten sich mit Aktualisierung auf Version 2.4 nun fehlerhaft verhalten.

Vergleichen wir in diesem Zusammenhang die Kommentare in der API-Dokumentation zu `sessionDestroyed()`, finden wir folgende unscheinbare Veränderung:

- Version 2.3: »Notification that a session was invalidated.«
- Version 2.4: »Notification that a session is about to be invalidated.«

Die Kompatibilitätsanalyse entdeckt diese Veränderung an der API-Dokumentation, bewertet die Veränderung eines Kommentars jedoch als Hinweis, da nicht beurteilt werden kann, was die Veränderung semantisch aussagt. In der FDMM-Beschreibung wurde der Lebenszyklus einer Session jedoch als Hook-Protokoll erfasst. Die Möglichkeiten einer Protokollbeschreibung durch das FDMM machen dann die erweiterten Analysen möglich.

Betont sei auch, dass diese Veränderung keine Veränderung der API nach sich gezogen hat. In den 139 erkannten API-Veränderungen durch den API-Vergleich ist diese Veränderung nicht enthalten. Erst die Differenzanalyse auf Basis des FDMM ist in der Lage derartige Veränderungen zu erkennen und zu bewerten.

Das gezeigte Beispiel zeigt die Stärken der automatischen Kompatibilitätsprüfung. Selbst bei einem Framework mit relativ wenigen Erweiterungspunkten ergeben sich messbare Vorteile. Wir können daraus schließen, dass umso umfangreicher das Framework ist, und damit unüberschaubarer die Auswirkungen bei Aktualisierung der Framework-Version, umso hilfreicher ist der Einsatz einer automatischen Kompatibilitätsanalyse.

7.6 Zusammenfassung

In diesem Kapitel haben wir die Umsetzung der automatischen Kompatibilitätsprüfung im Werkzeug Companian vorgestellt. Wir haben die einzelnen Analyseschritte der Prüfung erörtert und sind auf ihre Umsetzung eingegangen.

Anhand einer konkreten Beispielanwendung konnten wir die Vorteile der automatischen Kompatibilitätsprüfung demonstrieren. Es zeigt sich, dass mit diesem Vorgehen mögliche Inkompatibilitäten in Bezug auf die konkrete Benutzung eines Frameworks aufgedeckt werden können. Dabei ist die Analyse existierenden Verfahren deutlich überlegen, da zum einen nur relevante Unterschiede betrachtet werden und zum anderen die Qualität der erkannten Unterschiede höher ist als bei einem API-Vergleich.

So haben wir gezeigt, dass mit dem Vergleich auf Basis einer FDMM-basierten Framework-Beschreibung Veränderungen erkannt werden können, die mit einem API-Vergleich nicht entdeckt werden. Damit verdeutlicht das Beispiel die Überlegenheit des entwickelten Ansatzes gegenüber den bisherigen Möglichkeiten.

8

Zusammenfassung und Ausblick

Die Zukunft soll man nicht voraussehen wollen,
sondern möglich machen.

-- Antoine de Saint-Exupery

In dieser Arbeit haben wir den Ansatz einer automatischen Kompatibilitätsprüfung Framework-basierter Anwendungen vorgestellt. Die Aufgabenstellung bestand darin, ein Verfahren zu entwickeln, das automatisch die Kompatibilität einer Anwendung zu einer neuen Framework-Version analysiert.

Mit dem beschriebenen Ansatz und dessen Umsetzung im Werkzeug Companian haben wir die Aufgabenstellung gelöst. In Kapitel 2 haben wir den Begriff der Kompatibilität einer Anwendung zu einem Framework präzisiert. Mit diesem Grundverständnis der Kompatibilität entwickelten wir in Kapitel 3 das Verfahren aus Benutzungs-, Differenz- und Kompatibilitätsanalyse. Voraussetzung für dieses Verfahren ist eine geeignete Framework-Beschreibung, deren Anforderungen an eine ganzheitliche Framework-Beschreibungssprache wir in Kapitel 4 erarbeitet haben. Zur Definition dieser Sprache haben wir in Kapitel 5 das Konzept der Wiederverwendung von Sprachen mit den dazugehörigen Modellierungs- und Prüftechniken vorgestellt. Dieses Konzept haben wir dann in Kapitel 6 zur Definition des FDMM erfolgreich eingesetzt. In

Kapitel 7 haben wir die Umsetzung des Ansatz der automatischen Kompatibilitätsprüfung in Companian vorgestellt und belegt, dass diese Umsetzung die Problemstellung löst.

Companian ist ein prototypisch implementiertes Werkzeug, das, eine FDMM-basierte Framework-Beschreibung vorausgesetzt, auf Knopfdruck einen Kompatibilitätsbericht erstellt, der die relevanten Kompatibilitätseinschränkungen aufschlüsselt. Ein solcher Kompatibilitätsbericht bietet eine verlässliche Grundlage für weitere Entscheidungen und Analysen. Die automatische Kompatibilitätsprüfung verbessert die Möglichkeiten zur Beurteilung möglicher Inkompatibilitäten zwischen einer Anwendung und einem Framework in der industriellen Praxis [Christ2011].

Die beschriebenen Fallstudien, insbesondere zum Java Servlet Framework, haben gezeigt, dass der Ansatz für praxisrelevante Frameworks eingesetzt werden kann und die gewünschten Ergebnisse liefert.

Der Ansatz setzt eine FDMM-basierte Framework-Beschreibung beider Framework-Versionen voraus. Erst die zusätzlichen Informationen, die nun erstmals in einer Framework-Beschreibungssprache erfasst werden können, ermöglichen die beschriebenen Analyseverfahren. Liegen diese Informationen nicht vor, kann die Kompatibilitätsprüfung nicht durchgeführt werden.

Wir haben argumentiert, dass die Erstellung einer Framework-Beschreibung ein fester Bestandteil Framework-basierter Entwicklungsprozesse ist. Mit dem FDMM haben wir eine Sprache definiert, die auf Basis wissenschaftlicher Erkenntnisse festlegt, wie man ein Framework beschreibt. Nichtsdestotrotz bleibt die Erstellung von Dokumentationen eine unliebsame Aufgabe, die durch ausgereifte Werkzeuge unterstützt werden muss. Der Companian-Editor ist eine prototypische Umsetzung für die Erstellung von Framework-Beschreibungen. Für eine industriell einsetzbare Lösung müsste mehr Wert auf Benutzbarkeit und eine Integration in vorhandene Entwicklungswerkzeuge gelegt werden.

Die Notwendigkeit zur Entwicklung des FDMM [Christ2010] entstand, da bestehende Ansätze die gestellten Anforderungen nicht erfüllen konnten. Damit ist das FDMM als ganzheitliche Framework-Beschreibungssprache eine Verbesserung im Vergleich zu bestehenden Ansätzen zur Framework-Beschreibung.

Die Definition des FDMM als ganzheitliche Framework-Beschreibungssprache eröffnete ein weiteres Problem, da es bestehenden Modellierungsansätzen an Unterstützung für die Wiederverwendung von Sprachen mangelt. Aus diesem Grund haben wir den neuen Ansatz einer anforderungsbasierten Wiederverwendung von Sprachen mit der Modellierungstechnik der parametrisierten Meta-Modelle entwickelt.

Dieser neue Ansatz zur Definition einer Sprache, die bestehende Sprachen als Teilaspekte integriert, ist in zweierlei Hinsicht innovativ. Zum einen führt es das Konzept des Meta-Modell-Parameters ein. Ein Meta-Modell-Parameter wird als formaler Parameter durch einen konkreten Parameter ersetzt, der durch das Meta-Modell der bestehenden Sprache gegeben ist. Voraussetzung für die Ersetzung ist eine valide Bindung der bestehenden Sprache an den Parameter. Zur Prüfung der Bindung haben wir als zweite Innovation den semantischen Bindungstest etabliert, der das erwartete Ausführungsverhalten und damit die

Semantik einer gebundenen Sprache gemäß der festgelegten Bindung testet. So können nur Sprachen wiederverwendet werden, die den Sprachanforderungen des Parameters genügen.

Der praktische Nutzen dieses Vorgehens besteht darin, dass Sprachen gemäß einer Sprachanforderung auf ihre Eignung getestet werden können. Im Falle bestandener Tests kann die Sprache auf Ebene der Meta-Modelle integriert werden. So können bestehende Sprachen und insbesondere ihre Werkzeuge wiederverwendet werden. Bei Erstellung des FDMM haben wir diese Technik eingesetzt, um UML-Protokollzustandsdiagramme zur Beschreibung von Hook-Protokollen und UML-Aktivitätendiagramme zur Beschreibung des Framework-Verhaltens wiederverwenden zu können.

Auf Grundlage des parametrisierten FDMM können die UML-Diagramme in existierenden UML-Werkzeugen angelegt und anschließend im Companian-Editor importiert werden. Durch die Parametrisierung ist der Zugriff auf diese Modelle in Companian transparent.

Alle in dieser Arbeit eingeführten Ansätze haben den Anspruch praktisch einsetzbar zu sein. Durch Implementierung der Verfahren in Companian haben wir die praktische Einsetzbarkeit prototypisch demonstriert. Nichtsdestotrotz können einige Verbesserungen in zukünftigen Entwicklungen berücksichtigt werden.

Die Benutzungsanalyse in Companian basiert auf einer einfachen statischen Codeanalyse zur Indexierung von Anwendungen und Frameworks. Diese Analyse könnte man zur Reduzierung falsch-positiver und nicht erkannter Benutzungsbeziehungen verbessern. Ein Weg wäre die Analyse auf Konfigurationsdateien innerhalb der Anwendung und des Frameworks auszuweiten. In Konfigurationsdateien stehen häufig relevante Informationen, die Aufschluss über die Benutzung des Frameworks geben können. Diese Konfigurationsdateien sind jedoch nicht standardisiert, was eine generische Lösung erschwert.

Eine weitere mögliche Verbesserung liegt darin, dass man die Analyse dynamisch gestaltet. Hierbei analysiert man das tatsächliche Verhalten der Anwendung zur Laufzeit. Eine solche Analyse kann in dem Sinne exaktere Ergebnisse liefern, als dass sie analysiert, welche Teile der Anwendung tatsächlich ausgeführt werden. Eine statische Analyse kann dies nicht. Die Annahme ist, dass bei einer dynamischen Analyse weniger falsch-positive Treffer in der Benutzungsanalyse auftreten. Dem lässt sich jedoch entgegnen, dass die einfache statische Analyse in den Fallstudien ausreichend war. Dennoch wäre dies eine potentielle Stelle für Verbesserungen.

Die Differenzanalyse bietet ebenfalls Potential für Verbesserungen. Für den Vergleich zweier Hook-Protokolle oder Verhaltensbeschreibungen eines Frameworks haben wir jeweils eine einfache Bisimulation in Companian implementiert. Diese könnte durch Optimierungen verbessert werden, so dass man die Auswirkungen von Veränderungen besser interpretieren kann. Im Rahmen dieser Arbeit wurde dieser Pfad jedoch nicht weiter vorangetrieben.

Die automatische Kompatibilitätsprüfung hat neben der wissenschaftlichen Betrachtungsweise auch große Relevanz in der Praxis. Die persönlichen Erfahrungen in industriellen und Open-Source Software-Projekten haben immer wie-

der bestätigt, dass dieses Verfahren helfen kann, um tägliche Fragen und Probleme von Entwicklern schnell zu beantworten. Damit besitzt das Verfahren Potential für eine kommerzielle Verwertung.

Denkbar wäre zum Beispiel das Verfahren als Service anzubieten und es Kunden zu ermöglichen ihre Anwendungen testen zu lassen. Eine große Auswahl an dokumentierten und unterstützten Frameworks vorausgesetzt, kann dieser Service Analysen anbieten, die einem Projektleiter unter Umständen viel Geld sparen und Entscheidungen erleichtern kann.

Eine weitere Anwendungsdomäne für eine Kompatibilitätsprüfung wäre die Zertifizierung von Erweiterungen. Angenommen man betreibt eine Software-Plattform, für die Drittanbieter Erweiterungen implementieren können. Bevor diese Erweiterungen auf der Plattform installiert werden, sollen sie zertifiziert werden, um ihre Unbedenklichkeit zu dokumentieren. Eine solche Zertifizierung bescheinigt unter anderem die Kompatibilität der Erweiterung mit der Plattform, wofür sich die automatische Kompatibilitätsprüfung ideal eignen würde.

Das Problem einer Kompatibilitätsprüfung Framework-basierter Anwendungen ist relevant in Praxis und Wissenschaft. Die vorgestellte Lösung orientiert sich dabei primär an ihrer Praxistauglichkeit. Darüber hinaus liefern wir mit dieser Arbeit neue Erkenntnisse zur Lösung des Problems im wissenschaftlichen Sinne. Die eingeführten Modellierungstechniken zur Sprachdefinition lassen sich für eine Vielzahl von Sprachen einsetzen, die ähnlich gelagerte Anforderungen haben. Für eine ingenieurmäßige Umsetzung der Aufgabenstellung war die Erarbeitung dieser Grundlagen unabdingbar. Im Ergebnis ermöglichen es erst diese Erkenntnisse das gestellte Problem im Sinne einer Ingenieursdisziplin zu lösen.

Literaturverzeichnis

[Attiogbe2006] Christian Attiogbé, Pascal Undré und Gilles Ardourel. *Checking Component Composability.* In W. Lowe and M. Südholt (Hrsg.): Proceedings of the 5th International Conference on Software Composition, LNCS 4089, Springer, 2006

[Bass2007] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice.* Addison Wesley, 2007

[Ben-Abdallah2004] Hanêne Ben-Abdallah und Nadia Bouassida und Faiez Gargouri und Abdelmajid Ben Hamadou. *A UML based Framework Design Method.* Journal of Object Technologie, Vol. 3, 2004

[Bezivin2006] Jean Bézivin und Salim Bouzitouna und Marcos Del Fabro und Marie-Pierre Gervais und Fréderic Jouault und Dimitrios Kolovos und Ivan Kurtev und Richard Paige. *A Canonical Scheme for Model Composition.* In Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations und Applications (ECMDA-FA'06), Springer, 2006

[Blanc2005] Xavier Blanc, Franklin Ramalho und Jacques Robin. *Metamodel Reuse with MOF.* In Proceedings of the Model Driven Engineering Languages und Systems Conference (MODELS'05), Springer, 2005

[Boer2009] Remo C. de Boer und Hans van Vliet. *Writing und Reading Software Documentation: How the Development Process May Affect Understunding*. In Proceedings of the ICSE Workshop on Cooperative and Human Aspects on Software Engineering, IEEE, 2009

[Bosch1997] Jan Bosch, Peter Molin, Michael Mattsson und Perolof Bengtsson. *Object-Oriented Frameworks - Problems & Experiences.* 1997

[Bosch2000] Jan Bosch. *Design & Use of Software Architecture: Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000

[Bosch2000a] Jan Bosch, Peter Molin, Michael Mattsson und PerOlof Bengtsson. *Object-oriented Framework-based Software Development: Problems und Experiences*. ACM Computing Survey, Vol. 32, Nr. 1, ACM, 2000

[Bouassida2001] Nadia Bouassida, Hanêne Ben-Abdallah und Faiez Gargouri. *A UML based Design Language for Framework Reuse.* In Proceedings of the 7th International Conference on Object-Oriented Information Systems (OOIS'01), Springer, 2001

[Bouassida2002] Nadia Bouassida, Hanêne Ben-Abdallah, Faiez Gargouri und Abdemajid Ben Hamadou. *Stepwise framework design by application unification*. In Proceedings of the International Conference on Systems, Man and Cybernetics, IEEE, 2002

[Bouassida2003] Nadia Bouassida, Hanêne Ben-Abdallah, Faiez Gargouri und Abdemajid Ben Hamadou. *Formalizing the framework design language F-UML.* In Proceedings of the 1st International Conference on Software Engineering und Formal Methods (SEFM'03), IEEE, 2003

[Bouassida2004] Nadia Bouassida, Hanêne Ben-Abdallah und Abdemajid Ben Hamadou. *F-UMLTool for the formal design of frameworks.* In Proceedings of the XXIIème Congrés INFORSID, 2004

[Brown1997] A.W. Brown und K. Short. *On Components und Objects: The Foundations of Component-Based Development.* In Proceeding of the 5th International Symposium on Assessment of Software Tools (SAST '97), IEEE, 1997

[Brun2008] Cédric Brun und Alfonso Perantonio. *Model Differences in the Eclipse Modelling Framework.* UPGRADE, Vol. 9, Nr. 2, 2008

[Brunet2006] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu und Mehrdad Sabetzadeh. *A manifesto for model merging*. In Proceedings of the International Workshop on Global Integrated Model Management (GaMMa'06), ACM, 2006

[Butler1999] G. Butler und P. Dénommée. *Documenting Frameworks.* In M. Fayad, D. Schmidt und R. Johnson (Hrsg.): Building Application Frameworks, John Wiley & Sons, 1999

[Butler2000] Greg Butler, Rudolf Keller und Hafedh Mili. *A framework for framework documentation*. ACM Computing Survey, Vol. 32, ACM, 2000

[Campo2005] Marcelo R. Campo, J. Andrés Díaz Pace und Federico U. Trilnik. *„Computer, please, tell me what i have to do...“: an approach to agent-aided application composition.* Journal of Systems und Software, Vol. 74, Elsevier, 2005

[Canfora2007] Gerardo Canfora und Massimiliano Di Penta. *New Frontiers of Reverse Engineering*. In Proceedings of the Future of Software Engineering (FOSE '07), IEEE, 2007

[Carey2002] James Carey und Brent Carlson. *Framework Process Patterns.* Addison-Wesley, 2002

[Chikofsky1990] Elliot J. Chikofsky und James H. Cross II. *Reverse Engineering und Design Recovery: A Taxonomy.* Software, Vol. 7, IEEE, 1990

[Christ2008] Fabian Christ und Stefan Sauer. *Open Source Stacks.* In Michael Asche und Wilhelm Bauhus und Burckhard Kaddatz und Bernd Seel (Hrsg.): Open Source: Kommerzialisierungsmöglichkeiten und Chancen für die Zusammenarbeit von Hochschulen und Unternehmen, Waxmann, 2008

[Christ2010] Fabian Christ, Jan-Christopher Bals, Gregor Engels, Christian Gerth und Markus Luckey. *A Generic Meta-Model-based Approach for Specifying Framework Functionality und Usage*. In Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS'10), LNCS 6141, Springer, 2010

[Christ2011] Fabian Christ und Jan-Christopher Bals. *Kompatibilitätsanalyse bei Evolution framework-basierter Anwendungen.* In Proceedings of the 3rd Design for Future Workshop held at the Software Engineering 2011 Conference (SE2011), LNI P-184, GI, 2011

[Clarke1986] E. M. Clarke, E. A. Emerson und A. P. Sistla. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.* ACM Transactions on Programming Languages und Systems, Vol. 8, ACM, 1986

[Clements1996] Paul C. Clements. *A Survey of Architecture Description Languages.* In Proceedings of the 8th International Workshop on Software Specification und Design (IWSSD'96), IEEE, 1996

[Clements2002] Paul Clements und Linda M. Northrop. *Software Product Lines : Practices und Patterns.* Addison-Wesley, 2002

[Coplien1998] James Coplien, Daniel Hoffman und David Weiss. *Commonality und Variability in Software Engineering.* Software, Vol. 15, IEEE, 1998

[Cortes2006] Mariela Cortés, Marcus Fontoura und Carlos Lucena. *Framework Evolution Tool.* Journal of Object Technology, Vol. 5, 2006

[Cortes2006a] M. Cortés, M. Fontoura und C. Lucena. *A Rule-Based Approach to Framework Evolution.* Journal of Object Technology, Vol. 5, 2006

[Crnkovic2011] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis und Michel R.V. Chaudron. *A Classification Framework for Software Component Models.* Transactions on Software Engineering, Vol. 37, IEEE, 2011

[Cunningham2006] H. Conrad Cunningham und Pallavi Tadepalli. *Using Function Generalization to Design a Cosequential Processing Framework.* In Proceedings of the 39th Annual Hawaii International Conference on System Sciences, IEEE, 2006

[Cunningham2006a] H. Conrad Cunningham, Yi Liu und Pallavi Tadepalli. *Framework design using function generalization: a binary tree traversal case study.* In Proceedings of the 44th Annual Southeast Regional Conference, ACM, 2006

[Dagenais2008] Barthélémy Dagenais und Martin P. Robillard., *Recommending Adaptive Changes for Framework Evolution*. In Proceedings of the 13th International Conference on Software Engineering (ICSE'08), IEEE, 2008

[Dagenais2011] Barthélémy Dagenais und Martin P. Robillard, *Recommending Adaptive Changes for Framework Evolution*, Transactions on Software Engineering Methodology, Vol. 20, ACM, 2011

[Demeyer2000] Serge Demeyer, Koen De Hondt und Patrick Steyaert. *Consistent framework documentation with computed links and framework contracts.* Computing Surveys, Vol. 32, ACM, 2000

[Deutsch1989] L. P. Deutsch. *Design reuse und frameworks in the smalltalk-80 system.* Software Reusability, Vol. 2, 1989

[Dig2005] Danny Dig und Ralph Johnson. *The role of refactorings in API evolution.* In Proceedings of the 21st International Conference on Software Maintenance (ICSM'05), IEEE, 2005

[DoD1995] Defense Printing Service. *Performance Specification Guide.* United States of America - Department of Defense, 1995

[Emerson2006] Matthew Emerson und Janos Sztipanovits. *Techniques for Metamodel Composition.* In Proceedings of the 6th Workshop on Domain Specific Modeling of the OOPSLA Conference, 2006

[Engels2000] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel und Stefan Sauer. *Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML.* In Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 2000), Springer, 2000

[Escoffier2007] Clement Escoffier und Richard Hall. *Dynamically Adaptable Applications with iPOJO Service Components*. In Markus Lumpe und Wim Vunderperren (Hrsg.): Software Composition, Springer, 2007

[Fayad1997] Mohamed Fayad und Douglas C. Schmidt. *Object-oriented application frameworks.* Communications of the ACM, Vol. 40, ACM, 1997

[Fayad1999] Mohammed E. Fayad, Douglas C. Schmidt und Ralph E. Johnson. *Implementing Application Frameworks.* John Wiley & Sons, 1999

[Fayad1999a] Mohammed E. Fayad, Douglas C. Schmidt und Ralph E. Johnson. *Building application frameworks: object-oriented foundations of framework design.* John Wiley & Sons, 1999

[Fielding2000] Roy Thomas Fielding. *Architectural Styles und the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[Fleurey2008] Franck Fleurey, Benoit Baudry, Robert France und Sudipto Ghosh. *A Generic Approach for Automatic Model Composition.* In Proceedings of the MODELS 2007 Workshop on Models in Software Engineering, Springer, 2008

[Fontoura2000] Marcus Fontoura, Wolfgang Pree und Bernhard Rumpe. *UML-F: A Modeling Language for Object-Oriented Frameworks*. In Proceedings of the European Conference on Object-Oriented Programing (ECOOP2000), 2000

[Fontoura2001] Marcus Fontoura und Carlos Lucena. *Extending UML to Improve the Representation of Design Patterns.* Journal of Object Oriented Programming, Vol. 13, 2001

[Fontoura2002] Marcus Fontoura, Wolfgang Pree und Bernhard Rumpe. *The UML Profile for Framework Architectures.* Addison Wesley, 2002

[Forster2009] Alexander Förster. *Pattern based business process design und verification.* Dissertation, Universität Paderborn, 2009

[Froehlich1997] Gary Froehlich, H. James Hoover, Ling Liu und Paul Sorenson. *Hooking into object-oriented application frameworks.* In Proceedings of the 19th International Conference on Software Engineering (ICSE'97), ACM, 1997

[Froehlich2000] Garry Froehlich, H. James Hoover und Paul G. Sorenson. *Choosing an object-oriented domain framework.* Computing Surveys, Vol. 32, ACM, 2000

[Gamma1993] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Abstraction und Reuse of Object-Oriented Design.* In Proceedings of the European Conference on Object-Oriented Programming (ECOOP'93), LNCS 707, Springer, 1993

[Gamma1995] Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995

[Gangopadhyay1995] Dipayan Gangopadhyay und Subrata Mitra. *Understunding Frameworks by Exploration of Exemplars.* In Proceedings of the International Workshop on Computer-Aided Software Engineering (CASE'95), 1995

[Gosden1968] John A. Gosden. *Software compatibility: what was promised, what we have, what we need.* In Proceedings of the Fall Joint Computer Conference, Part I (AFIPS '68), ACM, 1968

[Groene2005] Bernhard Gröne, Andreas Knöpfel und Peter Tabeling. *Component vs. Component: Why We Need More Than One Definition.* In Proceedings of the 12th International Conference und Workshops on the Engineering of Computer-Based Systems (ECBS'05), IEEE, 2005

[Gurp2000] J. van Grup und Jan Bosch. *Design, Implementation und Evolution of Object Oriented Frameworks: Concepts & Guidelines.* Software – Practice and Experience, Vol. 31, John Wiley & Sons, 2000

[Hamu1998] David S. Hamu und Mohamed E. Fayad. *Achieving bottom-line improvements with enterprise frameworks.* Communications of the ACM, Vol. 41, ACM, 1998

[Hausmann2005] Jan Hendrik Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages.* Dissertation, Universität Paderborn, 2005

[Heinemann2001] George T. Heinemann und William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together.* Addison Wesley, 2001

[Helm1990] Richard Helm, Ian M. Hollund und Dipayan Gangopadhyay. *Contracts: Specifying Behavioral Compositions in Object-oriented Systems.* SIGPLAN Notices, Vol. 25, 1990

[Henkel2005] Johannes Henkel und Amer Diwan. *CatchUp!: Capturing and Replaying Refactorings to Support API Evolution.* In Proceedings of the 27th International Conference on Software Engineering (ICSE'05), ACM, 2005

[Hornkamp2009] Markus Hornkamp. *A Formal, Graph-Based Semantics for UML Activities*. Diplomarbeit, Universität Paderborn, 2009

[Hudak1998] Paul Hudak. *Modular Domain Specific Languages und Tools.* In Proceedings of 5th International Conference on Software Reuse, IEEE, 1998

[ISO/IEC10746-2] *Information technology - Open Distributed Processing - Reference Model: Foundations*, 10746-2, ISO/IEC, 1996

[Jackson2000] Kirk Jackson, Robert Biddle und Ewan Tempero. *Understunding Frameworks through Visualisation.* In Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00), IEEE, 2000

[Johnson1988] Ralph E. Johnson und Brian Foote. *Designing Reusable Classes.* Journal of Object-Oriented Programming, Vol. 1, 1988

[Johnson1992] Ralph E. Johnson. *Documenting Frameworks using Patterns.* In Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'92), ACM, 1992

[Johnson1997a] Ralph E. Johnson. *Frameworks = (Components + Patterns).* Communications of the ACM, Vol. 40, ACM, 1997

[Juergenfriedrich2011] Daniel Jürgenfriedrich. *Unterstützung von Framework-Versionen in Framework-Beschreibungssprachen für mobile Anwendungen.* Diplomarbeit, Universität Paderborn, 2011

[Kang1998] Kyo Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin und Moonhang Huh. *FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures*, Annals of Software Engineering, Vol. 5, 1998

[Kastenberg2006] Harmen Kastenberg und Arend Rensink. *Model Checking Dynamic States in GROOVE.* In Proceedings of the 13th International SPIN Workshop on Model Checking Software, Springer, 2006

[Khaluf2010] Lial Khaluf. *Business Process Quality Assurance.* Diplomarbeit, Universität Paderborn, 2010

[Khaluf2011] Lial Khaluf, Christian Gerth und Gregor Engels. *Pattern-Based Modeling und Formalizing of Business Process Quality Constraints.* In Proceedings of the 23rd International Conference on Advanced Information System Engineering (CAiSE'11), Springer, 2011

[Kirk2007] Douglas Kirk, Marc Roper und Murray Wood. *Identifying and Addressing Problems in Object-oriented Framework Reuse*, Empirical Software Engineering, Vol. 12, 2007

[Kolovos2006] Dimitrios S. Kolovos, Richard F. Paige und Fiona A.C. Polack. *Model Comparison: a Foundation for Model Composition and Model Transformation Testing.* Proceedings of the 2006 International Workshop on Global Integrated Model Management (GaMMa'06), ACM, 2006

[Kolovos2006a] Dimitrios Kolovos, Richard Paige und Fiona Polack. *The Epsilon Object Language (EOL).* In Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations und Applications (ECMDA-FA'06), Springer, 2006

[Krahn2008] Holger Krahn, Bernhard Rumpe und Steven Völkel. *MontiCore: Modular Development of Textual Domain Specific Languages*. In Proceedings of the 46th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'08), LNBIP 11, Springer, 2008

[Krasner1988] Glenn E. Krasner und Stephen T. Pope. *A cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, Vol. 1, 1988

[Kurtev2006] Ivan Kurtev, Jean Bézivin, Frédéric Jouault und Patrick Valduriez. *Model-based DSL Frameworks.* In Proceedings of the 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06), ACM, 2006

[Lajoie1994] Richard Lajoie und Rudolf K. Keller. *Design und Reuse in Object-Oriented Frameworks: Patterns, Contracts, und Motifs in Concert.* In Proceedings of the Colloquium on Object Orientation in Databases und Software Engineering (COODBSE'94), World Scientific, 1995

[Lara2010] Juan de Lara und Esther Guerra. *Generic Meta-modelling with Concepts, Templates und Mixin Layers.* In Proceedings of the International Conferene on Model Driven Engineering Languages und Systems (MODELS'10), Springer, 2010

[Ledeczi2001] A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi und M. Maroti. *On Metamodel Composition.* In Proceedings of the International Conference on Control Applications (CCA'01), IEEE, 2001

[Markiewicz2001] Marcus Eduardo Markiewicz und Carlos J.P. Lucena. *Object Oriented Framework Development.* Crossroads, Vol. 7, Nr. 4, 2001

[Mattsson1997] M. Mattsson und J. Bosch. *Framework Composition: Problems, Causes und Solutions.* In Proceedings of the 23rd Conference on Technology of Object-Oriented Languages und Systems (TOOLS'97), IEEE, 1997

[Mccamant2004] Stephen McCamant und Michael D. Ernst. *Early Identification of Incompatibilities in Multi-component Upgrades.* In Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004), Springer, 2004

[Meng2012] Sichen Meng, Xiaoyin Wang, Lu Zhang und Hong Mei. *A history-based matching approach to identification of framework evolution.* In Proceedings of the International Conference on Software Engineering (ICSE'12), IEEE, 2012

[Meusel1997] Matthias Meusel, Krzysztof Czarnecki und Wolfgang Köpf. *A Model for Structuring User Documentation of Object-oriented Frameworks using Patterns und Hypertext.* In Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97), Springer, 1997

[MOF20] Object Management Group. *Meta Object Facility (MOF) Core Specification*, Version 2.0, OMG, 2006

[Morisio1999] Maurizio Morisio, Daniele Romano und Corrado Moiso. *Framework based Software Development: Investigating the Learning Effect.* In Proceedings of the 6th International Software Metrics Symposium (METRIC'99), IEEE, 1999

[Muller2005] Alexis Muller, Olivier Caron, Bernard Carré und Gilles Vanwormhoudt. *On Some Properties of Parameterized Model Application.* In Proceedings of the 1st European Conference on Model Driven Architecture - Foundations und Applications (ECMDA-FA'05), Springer, 2005

[Nesterow2010] Viktor Nesterow. *Eine formale, graphbasierte Semantik für UML Statemachines.* Diplomarbeit, Universtität Paderborn, 2010

[OCL20] Object Management Group. *UML OCL Specification,* Version 2.0, OMG, 2003

[OMG2005] Object Management Group. *Unified Modeling Language, Superstructure*. Version 2.1.2, OMG, 2005

[Opdahl2006] Andreas L. Opdahl und Giuseppe Berio. *Interoperable Language and Model Management Using the UEML* Approach. In Proceedings of the International Workshop on Global Integrated Model Management (GaMMa'06), ACM, 2006

[Ortigosa1999] A. Ortigosa, M. Campo und R. Moriyn. *Enhancing Framework Usability Through Smart Documentation.* In Proceedings of the Argentinian Symposium on Object Orientation, 1999

[Ortigosa1999a] Alvaro Ortigosa und Marcelo Campo. *SmartBooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation.* In Proceedings of the Conference on Technology of Object-Oriented Languages und Systems (TOOLS'99), IEEE, 1999

[Parnas2011] David Lorge Parnas. *Precise Documentation: The Key to Better Software.* In Sebastian Nanz (Hrsg.): The Future of Software Engineering, Springer, 2011

[Pohl2005] Klaus Pohl, Günter Böckle und Frank van der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques.* Springer, 2005

[Pree1994] Wolfgang Pree. *Meta Patterns—A Means For Capturing the Essentials of Reusable Object-Oriented Design.* In Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94), Springer, 1994

[Pree1995] Wolfgang Pree. *Design Patterns for Object–Oriented Software Development.* Addison Wesley, 1995

[Pree1996] Wolfgang Pree und Gustav Pomberger. *Framework Component Systems: Concepts, Design Heuristics, und Perspectives.* In Proceedings of the 2nd International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Springer, 1996

[Pree1998] Wolfgang Pree. *Framelets als handliche Architekturbausteine*, 1998

[Pree2000] Wolfgang Pree. *Hot-Spot-Driven Framework Development*. In M. Fayad, D. Schmidt und R. Johnson (Hrsg.): Building Application Frameworks: Object-Oriented Foundations of Framework Design, John Wiley & Sons, 2000

[Pree2000a] Wolfgang Pree und Kai Koskimies. *Framelets\—small und loosely coupled frameworks.* Computing Surveys, Vol. 32, ACM, 2000

[Richner1998] Tamar Richner. *Describing Framework Architectures: more than Design Patterns.* In Proceedings of the European Conference on Object-Oriented Programming (ECOOP'98), 1998

[Roberts1998] Don Roberts und Ralph Johnson. *Patterns for Evolving Frameworks.* In Robert Martin und Dirk Riehle und Frank Buschmann (Hrsg.): Pattern Languages for Program Design 3, Addison Wesley, 1998

[Rogers1997] Gregory F. Rogers. *Framework-Based Software Development in C++.* Prentice Hall, 1997

[Savga2007] Ilie Savga und Michael Rudolf. *Refactoring-based support for binary compatibility in evolving frameworks.* In Proceedings of the 6th International Conference on Generative Programming and Component Engineering, ACM, 2007

[Schaefer2008] Thorsten Schäfer, Jan Jonas und Mira Mezini. *Mining Framework Usage Changes from Instantiation Code.* In Proceedings of the 13th International Conference on Software Engineering (ICSE'08), IEEE, 2008

[Schappert1995] Albert Schappert, Peter Sommerlad und Wolfgang Pree. *Automated Support for Software Development with Frameworks.* In Proceedings of the Symposium on Software Reusability (SSR'95), ACM, 1995

[Schmid1997] Han Albrecht Schmid. *Systematic Framework Design by Generalization.* Communications of the ACM, Vol. 40, ACM, 1997

[Schmidt2003] Douglas C. Schmidt und Frank Buschmann. *Patterns, Frameworks, und Middleware: Their Synergistic Relationships.* In Proceedings of the 25th International Conference on Software Engineering (ICSE'03), 2003

[Seibel2009] Peter Seibel. *Coders at Work.* apress, 2009

[Silva2000] Antonio Rito Silva, Francisco Assis Rosa und Teresa Goncalves. *Framework Description using Concern-specific Design Patterns Composition*, Computing Survey, Vol. 32, ACM, 2000

[Soltenborn2009] Christian Soltenborn und Gregor Engels. *Towards Test-Driven Semantics Specification.* In Proceedings of the MODELS'09 Conference, Springer, 2009

[Sun2003] Sun Microsystems, Inc., *Java(TM) Servlet API Specification Version 2.4*, Sun Microsystems, Inc., 2003

[Szyperski2002] Clemens Szyperski, Dominik Gruntz und Stephan Murer. *Component Software - Beyond Object-Oriented Programming.* Addison Wesley, 2002

[Terrasse2006] Marie-Noelle Terrasse, Marinette Savonnet, Eric Leclercq, Thierry Grison und George Becker. *Do We Need Metamodels and Ontologies for Engineering Platforms?.* In Proceedings of the International Workshop on Global Integrated Model Management (GaMMa'06), ACM, 2006

[Thomas2006] Dave Thomas. *The API Field of Dreams - Too Much Stuff! It's Time to Reduce und Simplify APIs!.* Journal of Object Technology, Vol. 5, 2006

[Tonella2005] Paolo Tonella. *Reverse Engineering of Object Oriented Code.* In Proceedings of the 27th International Conference on Software Engineering (ICSE'05), ACM, 2005

[Tonella2005a] Paolo Tonella und Alessundra Potrich. *Reverse Engineering of Object Oriented Code.* Springer, 2005

[UMLINF23] Object Management Group. *Unified Modeling Language (OMG UML) Infrastructure.* Version 2.3, OMG, 2010

[UMLSUP23] Object Management Group. *Unified Modeling Language (OMG UML) Superstructure.* Version 2.3, OMG, 2010

[W3CSC] W3C. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*, W3C, 2010

[Walter2009] Tobias Walter und Jürgen Ebert. *Combining DSLs und Ontologies Using Metamodel Integration.* In Proceedings of DSL 2009 Conference, Springer, 2009

[Weisemoeller2008] Ingo Weisemöller und Andy Schürr, *Formal Definition of MOF 2.0 Metamodel Components and Composition.* In Model Driven Engineering Languages and Systems, LNCS 5301, Springer, 2008

[Weiss1999] David Weiss und Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process.* Addison Wesley, 1999

[White2009] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale und Douglas C. Schmidt. *Improving Domain-Specific Language Reuse with Software Product Line Techniques.* Software, Vol. 26, IEEE, 2009

[Wirfs-Brock1990] Rebecca J. Wirfs-Brock und Ralph E. Johnson. *Surveying Current Research in Object-oriented Design.* Communications of the ACM, Vol. 33, ACM, 1990

[Wu2010] Wei Wu, Yann-Gael Guéhéneuc, Giuliano Antoniol und Miryung Kim. *AURA: a Hybrid Approach to Identify Framework Evolution.* In Proceedings of the 32nd International Conference on Software Engineering (ICSE'10), ACM, 2010

[Yang1998] Young Joon Yang, Soon Yong Kim, Gui Ja Choi, Eun Sook Cho, Chul Jin Kim und Soo Dong Kim. *A UML-based Object-oriented Framework Development Methodology.* In Proceedings of the Asia Pacific Software Engineering Conference (Taipei), IEEE, 1998

Abbildungsverzeichnis

Tabellenverzeichnis