Ausarbeitung zum Thema

Architekturmuster

(Layers, Pipes and Filters, Blackboard, Broker, Microkernel, Reflection)

im Rahmen des Seminars „Softwaretechnik“ im WS 04/05

Daniel Onnebrink

Themensteller: Prof. Dr. Herbert Kuchen
Betreuer: Prof. Dr. Herbert Kuchen
Institut für Wirtschaftsinformatik
Praktische Informatik in der Wirtschaft

Inhaltsverzeichnis

1 Wozu dienen Muster?

Umfangreiche Software von Grund auf neu zu entwickeln findet heutzutage nur noch selten statt. Dies wäre im Zusammenhang mit der heutigen Schnelllebigkeit und den ständig wachsenden Anforderungen nicht mehr wirtschaftlich. Daher werden Muster und ihre Beschreibungen eingesetzt, um ein erprobtes Lösungsschema als Vorgabe zu nutzen und entlang dieses Rahmens entwickeln zu können. Muster bestehen aus Beschreibungen von Problem-Lösungspaaren, welche über die Zeit von Experten zusammengetragen worden sind und ermöglichen so auch Neueinsteigern ohne langjährige Erfahrungen im Softwareentwicklungsbereich, häufig auftretende Entwurfsprobleme schneller zu lösen. Um komplexe Softwaresysteme entwickeln zu können sind Erfahrungswerte notwendig. Diese lassen sich gerade von Neulingen nicht im Hand umdrehen erlernen, sondern nehmen viel Zeit in Anspruch. Die allgemein zugänglichen Muster stellen einen mentalen Baustein dar, der diese Erfahrungslücke schließen kann.

Softwarearchitektur entscheidet mit über die Qualität des Systems. Eine gute Architektur ist gekennzeichnet durch Flexibilität. Gerade in der heutigen Zeit spielt Anpassbarkeit an ständige Änderungen und neue Ansprüche eine große Rolle. So kann es in einem bestimmten Kontext beispielsweise sinnvoll sein, ein großes System in mehrere, voneinander unabhängige Komponenten zu zerlegen um die Wartbarkeit der einzelnen Komponenten möglichst zur Laufzeit gewährleisten zu können, ohne alle Schnittstellen der zu wartenden Komponente beachten zu müssen. Die Wartbar- bzw. Anpassbarkeit des Systems würde in diesem Fall die „Kraft“ oder auch gewünschte Eigenschaft des Systems darstellen. Häufig stehen Muster nicht isoliert voneinander dar, sondern treten gemeinsam auf, da beispielsweise die Anwendung eines bestimmten Musters zu neuen Problemen führen kann, denen mit der Beschreibung anderer Muster begegnet wird. Desweiteren kann der Grund für die Anwendung von mehreren Mustern für einen Kontext darin liegen, dass ein Muster nicht alle „Kräfte“/ Anforderungen zufrieden stellend ausgleichen kann, sondern nur einige. Ein anderes Muster kann dann eingesetzt werden, um die unberücksichtigten Kräfte zu behandeln.

Im Zusammenhang mit der Softwareentwicklung von komplexen Systemen werden von den Entwicklern häufig Architekturmuster zur Komplexitätsbewältigung eingesetzt.

Sie zeigen ein erprobtes Lösungsschema auf und beschreiben die beteiligten Komponenten mit ihren Beziehungen und Zuständigkeiten.

Bei der Entwicklung von Software werden mehrere Phasen durchlaufen, bis schließlich die „Freigabe“ des Softwaresystems erfolgen kann. [SW04] Die Softwarearchitektur wird in der Phase „IT-Entwurf“ festgelegt. Hier kristallisiert sich auf Grund der vorherigen Analysen heraus, welches Muster zum Entwurf beitragen kann. Die Entscheidung für ein bestimmtes, oder mehrere Muster hängt von Fachkonzept und Pflichtenheft ab und hat entscheidende Auswirkungen auf die gesamte Entwicklung.

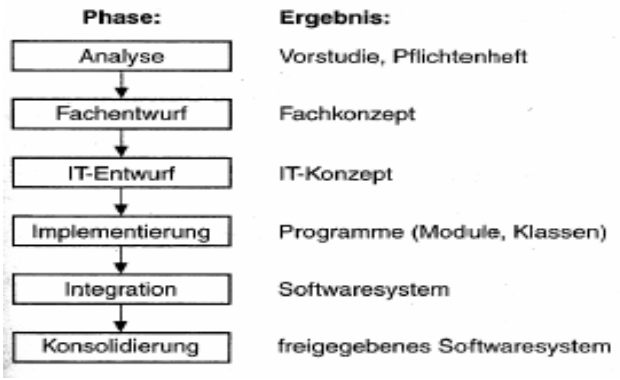

Abb. 1: Phasen der Softwareentwicklung

Muster lassen sich in verschiedene Abstraktionsebenen einteilen. Architekturmuster liegen auf der höchsten Ebene und sollen das Grundgerüst für eine Systemarchitektur schafften. Auf der darunter liegenden Ebene stehen Entwurfsmuster. Sie dienen der Verfeinerung von Komponenten oder Subsystemen und haben grundsätzlich keine Auswirkungen auf die Struktur des Gesamtsystems. Idiome stellen schließlich spezifische Muster für bestimmte Programmiersprachen dar. Die vorliegende Arbeit befasst sich mit Architekturmustern. Auf sechs ausgewählte Muster wird dazu in Kapitel 3 näher eingegangen. Im nachfolgenden Kapitel wird die bewährte und standardisierte Beschreibungstechnik von Mustern erläutert und begründet. Im weiteren Verlauf werden die Strukturierungsmuster „Layers“, „Pipes and Filters“ sowie „Blackboard“ beschrieben. Aus der Kategorie Verteilte Systeme das Muster „Broker“ und schließlich für Adaptierbare Systeme die beiden Muster „Microkernel“ und „Reflection“. In der Schlussbetrachtung erfolgt eine kurze Zusammenfassung mit Fazit. Nachfolgend wird der Aufbau der Arbeit grafisch dargestellt. (Abb. 1)

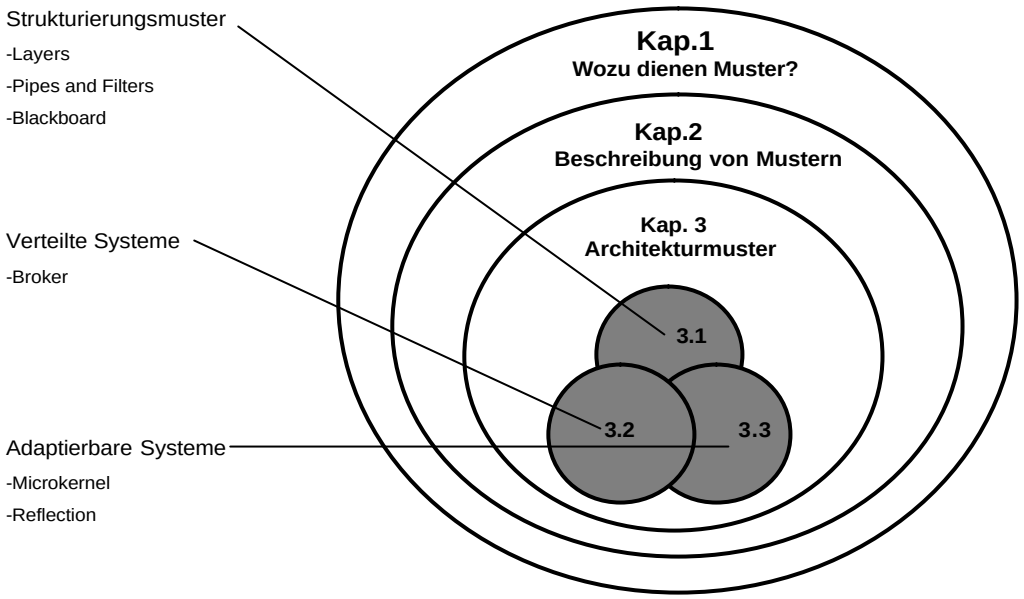

Abb. 2: Aufbau der Seminararbeit

Die Schnittmengen zwischen den verschiedenen Kategorien sollen aufzeigen, dass die bearbeiteten Muster teilweise auch Anforderungen anderer Kategorien abdecken. Auf Grund ihrer Schwerpunktsetzung und der Hauptanwendungsgebiete jedoch eindeutig kategorisiert wurden.

2 Beschreibung von Mustern

Da Muster in Zusammenarbeit von mehreren Experten entstehen und ständige Diskussionsgrundlage zur weiteren Verbesserung sind, benötigen sie eine eindeutige, einheitliche und verständliche Beschreibung. Für den Anwender der Muster muss gewährleistet werden, dass der Kern des Musters deutlich und so auch das mögliche Einsatzgebiet erkennbar wird. Durch eine einheitliche Beschreibung werden Muster vergleichbar und können voneinander abgegrenzt werden. Nur so können Problem-Lösungs-Paare identifiziert und das richtige Muster für das zu entwickelnde System oder Problem ausgewählt werden.

In der Literatur hat man sich auf die Struktur „Kontext - Problem - Lösung“ geeinigt. Der *Kontext* beinhaltet die Situation in der ein bestimmtes Problem auftritt und das Muster anwendbar ist. In der *Problem*beschreibung werden Probleme beschrieben, die in dem jeweiligen Kontext häufig auftreten und zu deren Lösung das Muster beitragen soll. Wünschenswerte Eigenschaften und andere Anforderungen an das System, werden mit dem Begriff „Kräfte“ umschrieben. Da unter Umständen Kräfte dabei sind, die sich nicht ergänzen sondern gegeneinander ausschließen oder abschwächen, besteht das Ziel darin, die Kräfte mit Hilfe eines oder mehrerer Muster möglichst auszugleichen. Bei der zum Schluss beschriebenen *Lösung* handelt es sich um die Darstellung einer erprobten Problemlösung. Hierzu werden *Strukturen* spezifiziert und das *dynamische Verhalten* zur Laufzeit dargestellt.

Die Beschreibungsstruktur lässt sich um Beispiele, Diagramme, verschiedene Szenarien, Implementierungsrichtlinien und andere Informationen zu dem Muster wie Vor- und Nachteile ergänzen. In der vorliegenden Arbeit wird aus Platzgründen darauf verzichtet, alle verfügbaren Informationen zu einem Muster ausführlich zusammenzutragen, sondern nur die prägnantesten. Ziel soll es sein, die Hauptidee der einzelnen Softwarearchitekturmuster und ihre möglichen Anwendungsgebiete aufzuzeigen.

3 Architekturmuster

Muster und ihre Beschreibungen tragen in unterschiedlichem Umfang und für unterschiedliche Systeme zur Softwareentwicklung bei. Während „Idiome“ Muster auf Programmiersprachenebene liefern, befassen sich „Entwurfsmuster“ mit der Verfeinerung von Subsystemen und Architekturmuster schließlich mit der Strukturierung von diesen. Die Muster befinden sich somit auf drei verschiedenen Abstraktionsebenen, wobei die Architekturmuster auf der höchsten Abstraktionsebene liegen. Sie liefern die grundsätzliche Strukturierung von Softwaresystemen und definieren des weiteren Subsysteme, deren jeweilige Zuständigkeiten, sowie ihre Interaktionen mit Hilfe von Regeln.

Die in dieser Arbeit beschriebenen Architekturmuster lassen sich wiederum in drei unterschiedliche Kategorien aufteilen: *Strukturierungsmuster* („Layers“, „Pipes and Filters“ und „Blackboard“), Muster für *verteilte Systeme* („Broker“), sowie *adaptierbare Systeme* („Microkernel“ und „Reflection“).

3.1 Strukturierungsmuster

Zu Beginn des Softwareentwurfs werden Anforderungen analysiert und spezifiziert. Zum anderen sollen auch so genannte weiche Kriterien berücksichtigt werden wie Integrierbarkeit, Wartbarkeit, Änderbarkeit, Portierbarkeit und Skalierbarkeit.[Si03, Kap. 6] Um zum einen den qualitativen Anforderungen gerecht zu werden, zum Anderen aber auch das Fachkonzept eines komplexen Systems zu berücksichtigen, bedarf es vorab einer Zerlegung in Teilaufgaben, um die Komplexität zu verringern. Mit dem Architekturmuster „Layers“ sollen Teilaufgaben auf gleichen Abstraktionsebenen geschichtet strukturiert werden. Das „Pipes and Filters“ Muster hingegen strukturiert Systeme in denen Datenströme verarbeitet werden sollen. Die einzelnen Verarbeitungsschritte werden dabei in Filtern gekapselt und die Daten werden über Kanäle (engl.: pipes) von der Datenquelle zur Datensenke weitergegeben. Bei dem „Blackboard“ Muster liegt meist eine Situation vor, in der es (noch) keine ausgereiften Lösungsstrategien gibt. Hier versucht man mit einer Menge von spezialisierten Subsystemen näherungsweise ans Ziel zu kommen.

3.1.1 Layers

Der *Kontext* für die Anwendung dieses Musters bildet ein komplexes System, dessen Größe eine Zerlegung erforderlich macht, um das System und seine Teilaufgaben zu strukturieren. Netzwerkprotokolle entsprechen diesem Kontext, da sie eine Vielzahl von Regeln beinhalten, die bei der Kommunikation in Netzwerken eingehalten werden müssen. Die genaue Definition der Regeln und Konventionen führt dazu, dass sich das System in Teilaufgaben zerlegen lässt. Bei dem ISO/OSI - Referenzmodell (Open System Interconnection) behandelt jede Schicht einen bestimmten Aspekt der Kommunikation und greift auf Dienste darunter liegender Schichten zurück. Netzwerkprotokolle werden so in sieben Schichten zerlegt.

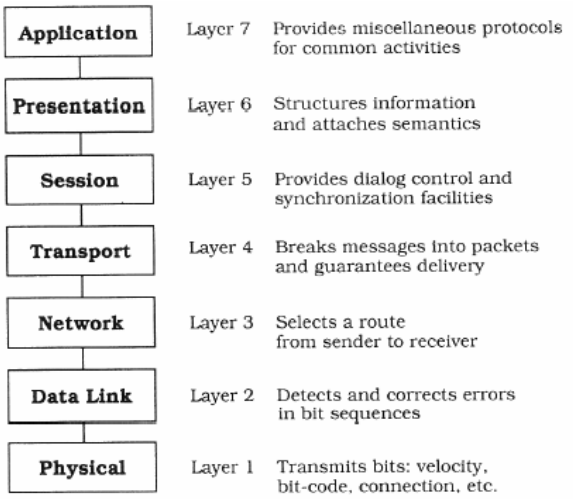

Abb. 3: ISO/ OSI – 7 – Schichten - Referenzmodell

Es soll also ein System entworfen werden, in dem voneinander unabhängige Operationen die zusammen auf einer Abstraktionsebene liegen, Operationen von niedrigeren Ebenen nutzen. Aufgaben wie Hardware-Ansteuerung oder Bitverarbeitung sollen so mit Aufgaben auf höchster Abstraktionsebene wie Anwenderfunktionalität und Strategie verknüpft werden. Die Anfragen laufen abwärts durch die Ebenen, während der Ergebnisdatenstrom in die Gegenrichtung läuft. Diese *Problem*stellung bringt unter anderem folgende Kräfte mit sich, die ausgeglichen werden müssen: Wartbarkeit (Quellcode- Änderungen sollen nicht das ganze System betreffen/ nur die zu ändernden Komponenten), stabile bzw. standardisierte Schnittstellen, Änderbarkeit (Isolierung von Komponenten für unabhängigen Austausch), Komplexitätsreduktion durch weitere Zerlegung von Komponenten, Wiederverwendbarkeit von Komponenten auf niedrigen Abstraktionsschichten.

Zur *Lösung* des Problems bildet man von einer Schicht mit den Grundfunktionalitäten ausgehend weitere Schichten, wobei der Abstraktionsgrad zunimmt. Das heißt, die Schichten werden Stück für Stück übereinander angeordnet, bis die höchste gewählte Abstraktionsebene erreicht ist. Eine Schicht darf hierbei nur Komponenten gleichen Abstraktionsgrades enthalten.

Bei der *Struktur* des Schichtenmodells ist zu beachten, dass eine Schicht nur Dienste für die direkt über ihr liegende Schicht bereithält und selbst wiederum Aufgaben an unter ihr liegende Schichten weitergeben kann. So wird ein Direktzugriff über mehrere Schichten hinweg verhindert. Für das *dynamische Verhalten* gibt es verschiedene Szenarien. Bei dem bekanntesten Szenario stellt ein Client eine „Dienstanfrage" an die oberste Schicht N. Diese wiederum gibt die Anfrage an die Schicht N-1 weiter, bis zur Schicht 1 runter.

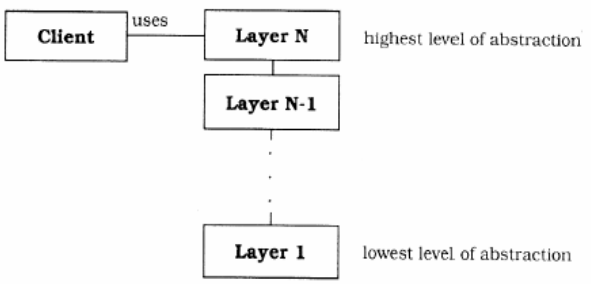

Abb. 4: Abstraktionsebenen

Ergebnisse werden, wenn nötig zurückgegeben bis hin zur obersten Schicht. Da die Anfragen von übergeordneten Schichten abstrakter sind, als die aufgerufenen Dienste der jeweils untergeordneten Schicht, nehmen die einzelnen Schichten meist mehrere Dienste von ihren untergeordneten Schichten in Anspruch. Bei dem zweiten Szenario läuft der „Nachrichtenfluss" hingegen „bottom-up". Ausgangspunkt ist hier eine Nachricht der ersten Ebene, welche bis hin zur obersten Schicht weitergegeben wird. Schließlich wird der Benutzer benachrichtigt. Weitere Szenarien beschreiben Nachrichten- bzw. Dienstanforderungsweitergaben, die nicht alle Schichten durchlaufen, sondern zum Beispiel bezüglich des Eingangsszenarios nur so lange hinunter, bis eine Schicht ein Ergebnis zurückliefert. Also nicht bis zur ersten Schicht. Ein weiteres Szenario zeigt die Kommunikation zwischen zwei n-Schichten Stacks. Hier startet die Dienstanforderung in einem der beiden Stacks auf oberster Ebene, wird bis zur ersten Ebene weitergegeben, dann zum anderen Stack übertragen und dort bis

zur höchsten Ebene transportiert, um von dort wieder den gleichen Weg zurückzugehen zum Start.

Das TCP/IP Kommunikationsprotokoll stellt eine *Beispiellösung* dar. Hier werden die 7 Schichten des ISO/OSI Referenzmodells auf vier reduziert. Die Kommunikation erfolgt ähnlich wie im letzten „Szenario" beschrieben (s.o.). Weitere *Anwendungsbeispiele* von Schichtenarchitekturen sind virtuelle Maschinen; die JVM liefert plattformunabhängigen Code und rückt so die Isolierung höherer von unteren Schichten in den Vordergrund. Ein weiteres Beispiel sind APIs, die ihre große Anzahl an häufig verwendeten Funktionen in Form von Schichten auf unterer Ebene strukturieren und so einkapseln. Bei Informationssystemen spricht man häufig von einer zwei Schichten Architektur, wobei die Datenbank als unterste Schicht die Datengrundlage für die verarbeitende bzw. präsentierende Oberschicht bietet.

Für das Layers Muster sprechen die *Vorteile* der Wiederverwendbar- und Austauschbarkeit von Schichten auf Grund von geringen Abhängigkeiten. Standardisierung von Aufgaben und Schnittstellen wird durch festgelegte Abstraktionskriterien möglich. Da Äbhängigkeiten lokal innerhalb der Schicht bleiben werden Codeänderungen, die nur die einzelne Schicht betreffen möglich. Geringe Effizienz auf Grund der Datenübertragung entlang mehrerer Schichten, sowie die Schwierigkeit bei der Festlegung von Abstraktionsebenen stellen die *Nachteile* des Layers Muster dar.

3.1.2 Pipes and Filters

Das Pipes-and-Filters Muster strukturiert Systeme, in dem *Kontext* „Verarbeitung von Datenströmen". Die Verarbeitungsschritte werden in Filter eingekapselt und lassen sich so beliebig anordnen und getrennt voneinander entwickeln. Daten kommen von der Datenquelle sollen mit Hilfe von einem oder mehreren Verarbeitungsschritten transformiert und schließlich zur Datensenke weitergegeben werden. Folgende *Kräfte* müssen bei diesem *Problem* ausgeglichen werden: Die zukünftige Anpassung soll durch Neuanordnung oder kompletten Austausch einzelner Komponenten gewährleitstet sein. Bei der Wiederverwendung spielt die Größe der Komponenten eine Rolle. Kleinere Komponenten sind flexibler einsetzbar. Die heterogenen Eingabequellen sollen auf zu verschieden darstellbaren Ergebnissen führen. Informationsaustausch erfolgt nur zwischen aufeinander folgenden Filtern, während die Verarbeitung in den Filtern auch

parallel möglich sein soll. Das Zwischenspeichern von Ergebnissen in Dateien ist fehleranfällig und sollte von daher nach Möglichkeit vermieden werden.

Eine *Lösung* wird durch die Anordnung von Verarbeitungsschritten und die Verbindung von zwei Filtern durch einen Kanal (engl.: pipe) erzielt. Die Reihenfolge der nacheinander angesteuerten Filter ergibt den Weg der so genannten „Verarbeitungspipeline“. Filter transformieren, ergänzen oder verfeinern ihre Eingabedaten. Der Datenfluss zwischen den Filtern erfolgt nach dem „push-“ oder „pull-“ Prinzip. Während aktive Filter ihre Eingabedaten selbständig aus der Pipeline zur Verarbeitung entnehmen und die Ergebnisse an die Pipeline übergeben, stellen passive Filter ihre Ergebnisse nachfolgenden Elementen zur Verfügung und bekommen zu bearbeitende Eingabedaten übergeben. Ein passiver Filter wird als Funktion (pull) oder Prozedur (push) aufgerufen, während aktive Filter selbständig als separates Programm starten. Ein typisches, dynamisches Verhalten eines Pipes-and-Filters Systems wird mit nachfolgender Abbildung illustriert. Hier sind alle Filter (Filter 1 und 2) aktiv. Die beiden aktiven Filter werden durch einen puffernden „First-in-First-out“ Kanal miteinander verbunden und synchronisiert.

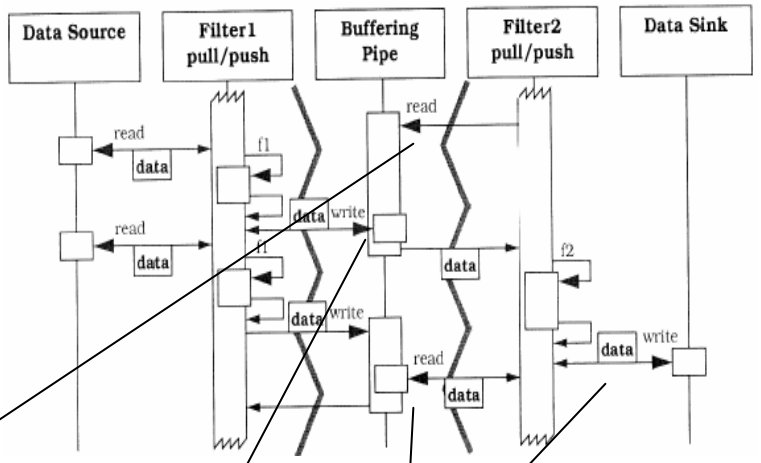

Abb. 5: *dynamisches Szenario* in einem Pipes-and-Filters System

Zunächst versucht der zweite Filter an Eingabedaten zu gelangen und liest hierfür auf der Pipe. Da noch keine Daten vorhanden sind, wird die Anfrage gestoppt. Der erste Filter entnimmt der Datenquelle eine Eingabe und verarbeitet diese (f1). Das Ergebnis wird an den Kanal übergeben. Nun kann Filter 2 Daten einlesen und verarbeiten (f2) und Filter 1 holt sich ebenfalls neue Eingabedaten, während Filter 2 *parallel* seine Ergebnisse in die Datensenke schreibt. Nachdem Filter 2 die bereits verfügbaren Daten aus dem Kanal gelesen hat, kann Filter 1 auch wieder mit der Verarbeitung fortfahren.

Bei der *Variante* Tee-and-Join-Pipeline-System wird die Regel, dass ein Filter nicht mehr als eine Ein- bzw. Ausgabe erlaubt gelockert und so die Verarbeitung entlang gerichteter Graphen ermöglicht. *Anwendung* findet das Muster in UNIX-Systemen wo mit dem Pipe-Befehl „|“ Kommandos beliebig angeordnet werden können. In Abb. 5 wird die Verbindung der beiden UNIX-Kommandos ls/bin und wc –w durch eine Pipe dargestellt. [He99, S. 17]

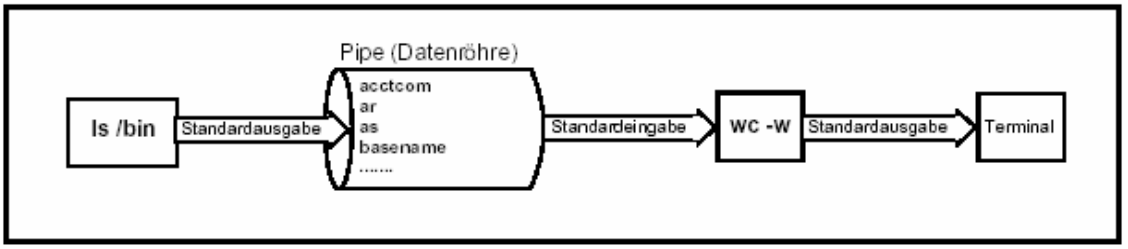

Abb. 6: Pipeline für den UNIX Befehl ls/ bin | wc –w

Die Abspeicherung von Zwischendateien wird durch die Verwendung von (puffernden) Kanälen und Filtern unnötig. Ein weiterer *Vorteil* liegt in der Austauschbarkeit und beliebigen Neuanordnung der Komponenten. So lassen sich nicht effektive Implementierungen eines Filters leicht durch bessere Lösungen ersetzen. Aktive Filter erlauben die Konstruktion von Pipelines durch den Endanwender. Parallelverarbeitung, wie in Abb. 5 dargestellt steigert die Effizienz. Bei UNIX schreibt das Kommando ls /bin beispielsweise gleichzeitig in die Pipe, während gleichzeitig wc –w aus dem Puffer liest (Abb. 6).

Ein *Nachteil* stellt die Synchronisation und Weitergabe zwischen Filtern dar. Sie verursachen nicht zu unterschätzende Kosten und somit Effizienzverluste. Beim Einsatz des Pipes-and-Filters Muster gibt es keinen globalen Zustand im System Zustände was eine Fehleridentifizierung bzw. -lokalisierung und -behandlung schwierig gestaltet. Hohe Aufwendungen können auf Grund von Datenkonvertierungen in Filtern entstehen, wenn man sich auf ein einheitliches Datenformat für die gesamte Verarbeitungspipeline geeinigt hat.

3.1.3 Blackboard

Das Blackboard Muster wird angewendet bei Problemen, die nicht auf eine eindeutige Lösungsstrategie hindeuten. Den *Kontext* für „Blackboard“ bilden somit Problembereiche, für die es noch keine festgelegten Lösungsstrategien gibt. Beispiele hierfür sind Spracherkennungs-, Bildverarbeitungs-, sowie Überwachungssysteme. Die

Probleme lassen sich in mehrere spezifische Teilprobleme zerlegen, deren Zusammenspiel zur Zielerreichung näherungsweise durch ausprobieren bestimmt werden kann. Es gilt jedoch hierbei mehrere *Kräfte* auszugleichen. Die große Anzahl an möglichen Lösungen (bsplw. Permutation von Buchstaben zur Erzeugung von Wörtern) kann zeitlich gesehen nicht komplett zur Lösungsfindung durchlaufen werden. Da die „perfekte“ Anordnung der heterogenen Teilprobleme und somit die beste Reihenfolge der Aufgabenbewältigung einzelner Komponenten nicht von vornherein bekannt ist, müssen die Komponenten zum experimentieren leicht austauschbar sein. Die Algorithmen müssen heterogene Eingabedaten verwenden und geben ihrerseits unterschiedliche Repräsentationsformen aus, welche wiederum als Eingabe für andere Algorithmen dienen. Bei Ausgabedaten treten oftmals mehrere mögliche Lösungen auf. Dies kann zum Beispiel bei gleicher Aussprache von Wörtern vorkommen, die somit ohne syntaktische Aspekte schwierig auseinander zu halten sind.

Die *Lösung* liegt in einem System von spezialisierten Programmen, die über eine Steuerungskomponente, abhängig vom Gesamtfortschritt eingesetzt und so zentral gesteuert werden. Entlang des Lösungsprozesses werden so Teillösungen erarbeitet, kombiniert, geändert oder aber verworfen. Man kann sich eine Tafel namens „Blackboard“ vorstellen mit einem notwendigerweise einheitlichem Vokabular, auf der die Lösungshypothesen der „Spezialisten“ bewertet (abhängig von vermutetem Wahrheitsgrad, Abstraktionsebene, etc.) und entsprechend verworfen oder beibehalten werden. Je höher die Abstraktionsebene einer Teillösung ist, desto näher ist diese an der möglichen Gesamtlösung. Auf der niedrigsten Ebene befindet sich die interne Repräsentation der Eingabedaten. Jeder „Spezialist“ entscheidet anhand des jeweiligen Fortschritts, ob er zur Lösung beitragen kann und tut dies gegebenenfalls mit anschließender Übermittlung des Ergebnisses an das Blackboard. Nachfolgend wird der *dynamische* Ablauf und das Zusammenspiel der besprochenen drei Komponenten (Steuerung, Blackboard, Wissensquellen bzw. Spezialisten) unter Verwendung des Spracherkennungsbeispiels geschildert. An Hand der Abbildung 6 wird das Vorgehen deutlich. Die Control-Komponente stößt die Wissensquellen an, nachdem festgestellt wurde, dass „Segmentation“, „Syllable Creation“ und „Word Creation“ zur Lösung beitragen können. Diese drei Programme bewerten nun (execCondition), nach Betrachten (inspect) des Blackboards, inwiefern sie einen Beitrag zur Lösung machen können. Auf Grund dieser Ergebnisse entscheidet sich die Steuerungskomponente für ein auszuführendes Programm und startet in unserem Beispiel den Aktionsteil der

„Silbenerzeugung“ (engl. Syllable Creation) durch „execAction“. Das Ergebnis wird auf das Blackboard geschrieben (update).

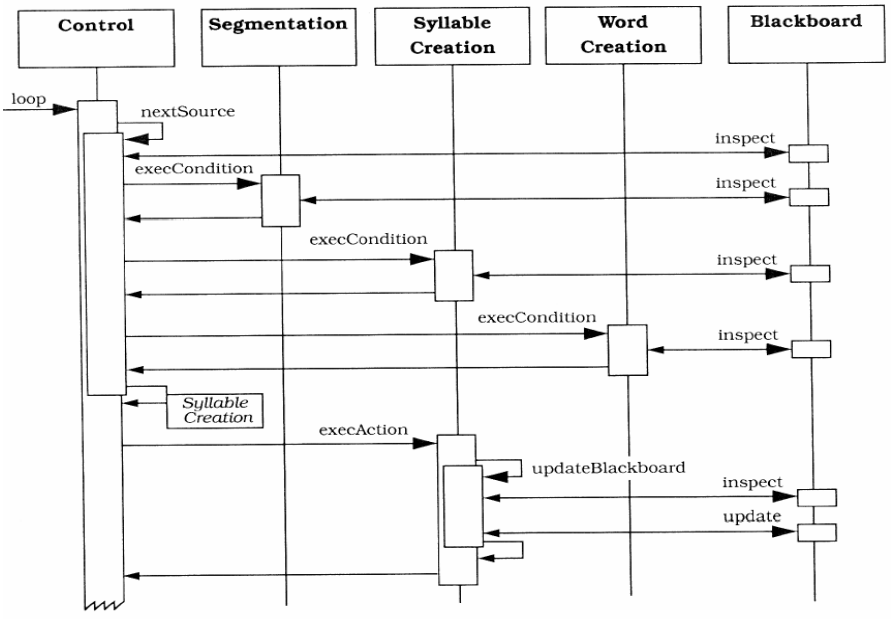

Abb. 7: Blackboarddynamik

Erste Anwendung fand das Blackboard Muster beim HEARSAY-II-Spracherkennungssystem. Mit Hilfe von Spracherkennung sollte dieses System Dokumente aus einer Datenbank heraussuchen. Ein weiteres Einsatzgebiet fand die Blackboard Architektur beim HASP-System, welches durch die Auswertung von Sonarsignalen U-Boote orten sollte.

Das Blackboard Muster bietet genügend Raum zum Experimentieren mit Algorithmen und Heuristiken. Durch die voneinander getrennten Wissensquellen, die über das Blackboard indirekt miteinander kommunizieren wird Wartbar- und Änderbarkeit unterstützt. Es werden nur Lösungsbeiträge übernommen, die auf Unterstützung der anderen Komponenten treffen. *Nachteile* entstehen auf Grund der schlechten Vorhersagbarkeit von Lösungsqualitäten. Zum anderen stellen schlechte oder falsche Teillösungen einen Bestandteil des Lösungsweges dar, auch wenn sie mit Hilfe von aufwendigen Bewertungsregeln wieder verworfen werden. Desweiteren trägt das Ausgangsproblem auf Grund seiner schlechten Strukturiertheit zu einem hohen Entwicklungsaufwand bei.

3.2 Architekturmuster für verteilte Systeme

Ein Trend zur Abkehr von dem zentralen Systemansatz ist seit einiger Zeit zu erkennen. Ausgelöst wurde dies durch die rasant wachsende Leistung der Hardware zu schrumpfenden Preisen. Desweiteren verbesserte sich die Infrastruktur lokaler Datennetze. So entstand „gebündelte Rechenleistung“, die zu gesteigerter Leistungsfähigkeit und Zuverlässigkeit der Systeme geführt hat. Ein Nachteil von verteilten Systemen ist, dass sie eine grundlegend andere Software, als zentralisierte Systeme benötigen. Drei Muster sind im Zusammenhang mit „Verteilten Systemen“ zu nennen. Das Pipes-and-Filters- sowie das Microkernel Muster fallen auch in diese Kategorie. Sie haben jedoch andere Schwerpunkte, weswegen an anderer Stelle auf sie eingegangen wird. Bei Pipes and Filters (siehe Kap. 3.1.2) geht es in erster Linie um die Strukturierung der grundlegenden Systemfunktionalität, während das Microkernel Muster (siehe Kap. 3.3.1) Anpassungsfähigkeit in den Vordergrund rückt und deshalb im Kapitel „Adaptierbare Systeme“ beschrieben wird. Das Broker Muster hingegen dient der Strukturierung verteilter Systeme mit entkoppelten Komponenten und wird nun näher erläutert.

3.2.1 Broker

Typische Beispiele für verteilte Informationssysteme stellen Veranstaltungskalender, Touristeninformationen, sowie Hotel- oder Reiseführer dar. Die Daten und Dienste sind hier dezentral verteilt, die Aufenthaltsorte können sich ändern und die Systeme werden ständig erweitert oder geändert. Eine Umgebung mit verteilten, eventuell heterogenen Systemen und ihren voneinander unabhängigen Komponenten, die in einem Netzwerk kommunizieren, stellt den *Kontext* des Broker Musters dar.

Die Kommunikation sollte jedoch nicht zu Abhängigkeiten führen, die entstehen wenn die Komponenten selbst für die Interprozesskommunikation (IPC) verantwortlich sind. Die Clients müssen dann beispielsweise wissen, wo sich die Server mit der gesuchten Information befinden. Nachfolgend werden weitere auszugleichende *Kräfte* aufgeführt:

Die Komponenten sollten über ein Netzwerk auf entfernte Dienste anderer Komponenten zugreifen können. Für den Zugriff sollte es nicht notwendig sein, implementierungsspezifische Details der Dienste, oder ihre Lokalisierung zu kennen. Außerdem ist der Aufenthaltsort flexibel und somit nicht zwingend bekannt. Einzelne

Komponenten sollten zur Laufzeit ausgewechselt, hinzugefügt, oder entfernt werden können. Dadurch wird gewährleistet, dass Anwender und Anbieter von Diensten zur Laufzeit verschwinden oder hinzukommen können. Zur *Lösung* des *Problem*s wird eine Vermittlungskomponente zwischen Clients (Anwender) mit ihren Dienstanforderungen und den entsprechenden Servern (Anbieter) eingeführt. Von daher muss nicht die anfragende Komponente wissen, wo der richtige Anbieter lokalisiert ist, sondern der Broker. Die Server melden sich selbständig beim Broker an und stellen so über Schnittstellen ihre Dienste zur Verfügung. Zu den Aufgaben des Brokers zählen die Weiterleitung von Dienstanforderungen an den richtigen Server, sowie die Übermittlung der Ergebnisse oder Fehlermeldungen zurück zum Client.

Sechs Komponenten beinhaltet das Broker Muster. Server, clients, brokers, bridges, client-side proxies und server-side proxies. Server stellen über Schnittstellen ihre Dienste über die flexible IDL (engl. Interface Definition Language), oder einen Binärstandard zur Verfügung. Entweder sie enthalten spezielle Funktionalitäten für eine Anwendung, oder aber allgemein gehaltene Dienste für mehrere Anwendungsgebiete.

Clients sind Anwendungen, die Dienste von mindestens einem Server aufrufen. Nachdem die Operation ausgeführt wurde, erhalten sie das Ergebnis über den Broker. Server können ebenfalls als Clients agieren und heben so eine feste Rollenzuweisung zugunsten von dynamischen *Strukturen* auf. Da die Clients nicht den Standort des angesprochenen Servers kennen müssen, ergibt sich der *Vorteil* die Komponenten zur Laufzeit auszutauschen, oder neue Dienste hinzuzufügen. Die „Vermittler“ bilden die Schnittstelle zwischen Client und Server. Falls der richtige Server allerdings nicht von dem Broker direkt angesteuert werden kann, muss er mit weiteren Brokern kommunizieren, bei denen der gesuchte Server angemeldet ist. Dies geschieht über so genannte „bridges“. Zwischen Server/Client und Broker stehen bei der indirekten Kommunikation über die Vermittlungskomponente proxy. Sie dienen zum Beispiel der Internalisierung von Daten (Client) bzw. Aufruf der relevanten Dienste (serverseitig) und berücksichtigen so jeweils die implementierungsspezifischen Details von Client und Server. An Hand der nachfolgenden Abbildung soll die indirekte Kommunikation über server- und client-side proxies illustriert werden. Alternativ dazu gibt es die *Variante „Direct-Communication“*, in der Client und Server nach erstem Verbindungsaufbau über den Vermittler direkt miteinander kommunizieren.

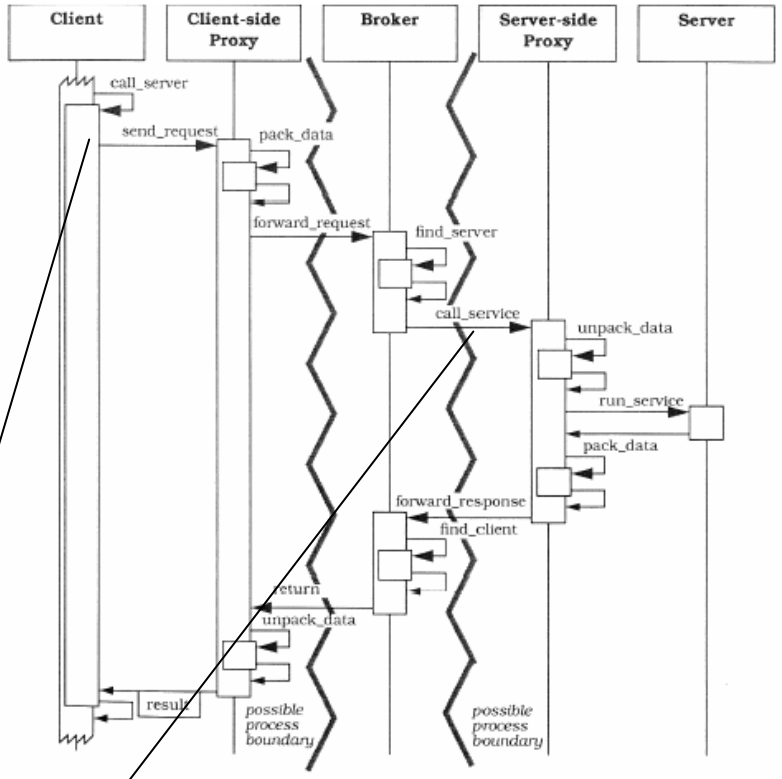

Abb. 8: Indirekte Kommunikation beim Broker Muster

Zunächst fordert die Client Anwendung den Dienst eines lokalen Servers an. Die Nachricht wird vom proxy an den Broker weitergeleitet, welcher in seinem Verzeichnis nach dem richtigen und angemeldeten Server sucht, um die Dienstanforderung dann entsprechend weiterzuleiten. Vom server-side proxy wird der geforderte Dienst dann gestartet und das Ergebnis über den gleichen Weg wieder zurück bis zur Client Anwendung geleitet.

Das Broker Muster findet in der Common Object Request Broker Architecture (CORBA) und der OLE-2.x-Technologie von Microsoft *Anwendung*. Zum anderen stellt das World Wide Web ein Broker System dar. Hier dienen www-Server als Anbieter und Browser als Clients.

Durch Broker können sowohl Clients, als auch Server unabhängig vom jeweiligen Standort miteinander kommunizieren. Mit den Zwischenschichten proxies und bridges wird Änderbarkeit und Erweiterbarkeit gewährleistet und gleichzeitig vermieden, dass Netzsystemdetails an Client oder Server weitergeleitet werden. Wenn sie das gleiche Nachrichtenprotokoll verstehen, können verschiedene Broker-Systeme miteinander kommunizieren. Ein weiterer *Vorteil* besteht in der Wiederverwendbarkeit der Dienste für weitere Anwendungen. Einen Nachteil bringt die Komponentenverteilung mit sich, da Vermittler zur Verbindung von Client und Server zwischengeschaltet werden

müssen, um eine Kommunikation zu gewährleisten. Dies bringt Effizienzverluste im Gegensatz zur direkten Kommunikation mit sich.

3.3 Architekturmuster für anpassbare Systeme

Softwaresysteme sehen sich heute in kürzester Zeit wechselnden Standards und Anforderungen gegenübergestellt. Um auf diese Veränderungen schnell und ohne großen Aufwand reagieren zu können benötigt man anpassbare, flexible Systeme. Das System sollte von vornherein mögliche Erweiterungen unterstützen, während die Kernfunktionalität davon unberührt bleiben kann. Nachfolgend werden die beiden Muster Microkernel und Reflection beschrieben. Ein funktioneller spartanischer Grundkern wird beim Microkernel Muster von der Fülle an möglichen Erweiterungen getrennt. So bietet es die Möglichkeit, gewünschte Funktionalität ohne großen Aufwand hinzuzufügen, wobei der Mikrokern die Interaktion der Funktionen koordiniert. Das Reflection Muster beschreibt auf einer Metaebene Systemeigenschaften, Verhaltensweisen und ihre dynamischen Veränderungen. Demgegenüber steht die Basisebene des Systems, welche seine veränderten Eigenschaften implementiert.

3.3.1 Microkernel

Software Systeme, die wechselnden Systemanorderungen genügen müssen, benutzen das Microkernel Muster. Der Mikrokern stellt hierbei die Basis für mögliche Erweiterungen dar und koordiniert gleichzeitig ihre Zusammenarbeit. Eine solche Architektur soll es verschiedenen Anwendungen ermöglichen auf einer Maschine und dessen funktionalen Kern zu laufen, obwohl die zugrunde liegenden Betriebssysteme unterschiedlich sind. In diesem *Kontext* sind mehrere *Probleme* zu lösen. Damit nicht nur die heutige Technik integriert werden kann, muss das System auch zukünftigen Hard- und Software-Entwicklungen standhalten können und sich dementsprechend erweitern und anpassen lassen, um verschiedene Anwendungsplattformen emulieren zu können.

Um die Probleme zu lösen und die damit zusammenhängenden *Kräfte* auszugleichen, werden Grundfunktionalitäten in einem „Mikrokern“ gekapselt, der möglichst wenig Speicherkapazität und Rechenleistung vereinnahmt. Funktionen, die auf Grund ihrer geringen Aufruffrequenz eine weniger Große Rolle spielen, werden in „interne Server“ verlagert, um die Performanz des Kerns zu bewahren und seinen Funktionsumfang so

gering wie möglich zu halten. Ausgelagerte speicherplatzintensive Funktionen in internen Servern, werden vom Mikrokern bei Bedarf über Dienstanforderungen aufgerufen ([1]). Andere Betriebssysteme können durch die Implementierung einer „Sicht“ ausgehend von den atomaren Diensten des Kerns von den externen Servern realisiert werden. Hierbei stellt der Mikrokern die von ihm verwalteten Systemressourcen und Kommunikationsdienste über Schnittstellen zur Verfügung ([2]). Verschiedene Sichten, stehen für unterschiedliche Anwendungsplattformen. Dienstanforderungen werden so von externen Servern und ihrer eigenen Sicht interpretiert und durch den Aufruf der entsprechenden Dienste bearbeitet. Das Ergebnis wird an den Client übertragen. Clients repräsentieren die Anwendungen eines externen Servers, der eine bestimmte Sicht beinhaltet. Mit Hilfe von Adaptern und den Kommunikationsdiensten des „Microkernel“ ([4]), werden die Anforderungen der Clients über den Adapter an die Server weitergeleitet ([3]). Unflexible Abhängigkeiten zwischen Server und Client werden so vermieden. Auf Grund der Adapterschnittstelle können Clients nicht zwischen emuliertem (Sicht vom externen Server) oder echten System unterscheiden. Die *Struktur* der Mikrokern-Architektur lässt sich somit in fünf Komponenten teilen: Microkernel, interne Server, externe Server, Adapter und Clients. Nachfolgendes Klassendiagramm visualisiert die statische Struktur der Komponenten. Hieraus lassen sich dynamische Szenarien beispielsweise der Dienstaufruf eines Client ableiten. Dafür stellt der Adapter mit Hilfe des Microkernel eine Kommunikationsverbindung zum Server her und übermittelt ihm danach die Dienstanforderung bspw. mit Hilfe eines RPC. Der Dienst wird ausgeführt (executeService) und das Ergebnis zurückgegeben.

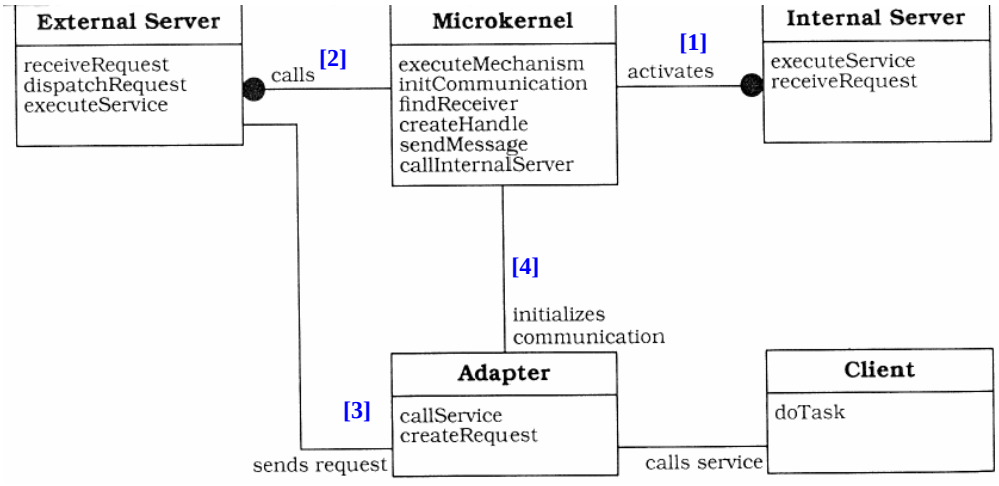

Abb. 9: Statische Struktur des Microkernel Muster

Eine Variante der besprochenen Architektur ergibt sich aus der indirekten Kommunikation von Client und Server über den Microkernel, der alle Anfragen weiterleitet. Eine weitere Variante ist im Sinne eines verteilten Systems möglich, in dem der Mikrokern ebenfalls als Nachrichtenübermittler agiert. Server und Clients können so Rechnernetzförmig verteilt werden.

Das Microkernel Muster wird beispielsweise im „Mach"-, „Amoeba"- und „Windows NT" Betriebssystem angewendet [BMRSS96, S. 188f.].

Auf Grund des flexiblen Mikrokerns, bedarf es bei der Portierung in andere Software- oder Hardwareumgebungen keiner großen Anpassungen. Ein weiterer *Vorteil* liegt in der Erweiterbarkeit; zusätzliche Funktionalität kann durch Hinzufügen oder Erweitern von Servern erreicht werden. Im Vergleich zu monolithischen Systemen ist das Microkernel Muster bezüglich der Performanz schlechter. Des weiteren ist der Entwicklungsaufwand sehr hoch, da genaue Kenntnisse über den Problembereich vorliegen müssen, um eine scharfe Trennung zwischen Mechanismen (Mikrokern) und Strategien, im Sinne von funktionalen Erweiterungen gewährleisten zu können.

3.3.2 Reflection

Systeme verfügen über spezifische Eigenschaften, die sich in ihrem Verhalten und ihrer Struktur wiederspiegeln. Das Reflection Muster benutzt für die Beschreibung der aktuellen Eigenschaften eine Metaebene. Die Anwendungslogik befindet sich auf der so genannten Basisebene und ändert sich abhängig von den beschriebenen Systemeigenschaften auf der Metaebene. Den *Kontext* bei Reflection bilden Systeme, die sich ihrer selbst bewusst sind (Metaebene) und eine Anpassbarkeit dadurch und im Allgemeinen unterstützen.

Bei der Veränderung und Wartung von Systemen tauchen häufig *Probleme* auf. Um die Änderungen sicher durchzuführen zu können, muss man über die meist komplexen Strukturen des Systems Bescheid wissen, damit keine Abhängigkeiten, die von veränderten Komponenten ausgehen übersehen werden. Häufig ist eine große Zahl Komponenten von Änderungen betroffen und steigern so den Gesamtaufwand. Die Komplexität nach außen sollte so gering wie möglich gehalten werden, um eine schnelle Wartung zu unterstützen. Die Anzahl der Veränderungsmechanismen muss reduziert und vereinheitlicht werden. Systemänderungen können von kleinen Anpassungen bis hin zu grundlegenden Strukturänderungen reichen. Um die genannten *Kräfte*

auszugleichen, ist ein System notwendig, welches über sich und somit seine Struktur und Verhalten Bescheid weiß. Bei der *Lösung* des Problems mit Hilfe des Reflection Musters enthält eine so genannte Metaebene die Selbstrepräsentation der Software. Änderungen auf der Metaebene, verändern gleichzeitig die Basisebene, welche die Anwendungslogik enthält. Dies geschieht, ohne den Code verändern zu müssen. Über die Schnittstelle „Metaobjektprotokoll“ (MOP) wird auf die „Metaobjekte“ (bspw. Typstruktur) in der Metaebene zugegriffen. Das MOP ist zudem verantwortlich für die Korrektheit und Umsetzung der Veränderungen in den Metaobjekten.

Die *Strukturierung* erfolgt also unter Verwendung der Meta- und Basisebene, sowie eines Metaobjektprotokolls. Auf der Metaebene befinden sich die Metaobjekte, welche Aspekte hinsichtlich Struktur, Verhalten oder Zustand des Systems kapseln. Sie bekommen ihre Informationen entweder über die Laufzeitumgebung, die Zustandsanalyse der Basisebene, oder durch Benutzervorgaben. Mit der Basisebene werden die Anwendungslogik, das Datenmodell und alle grundlegenden Beziehungen zwischen den Komponenten implementiert. Sie benutzt Typinformationen und andere veränderbaren Aspekte von den Objekten der Metaebene. So bleibt der Code auf Basisebene unabhängig von Veränderungen in den Metaobjekten. Das Metaobjektprotokoll stellt die externe Schnittstelle und somit die Zugriffsmöglichkeit auf die Metaobjekte dar. Veränderungen der Metaobjekte können von menschlichen Benutzern, oder Komponenten der Basisebene eingebracht werden. Zwischen Basis- und Metaebene existiert eine gegenseitige Abhängigkeit. Während die Metaobjekte ihre Informationen teilweise aus aktuellen Zuständen der Basisebene beziehen, benutzen („uses“/ siehe Abbildung 10) Komponenten der Basisebene die Informationen der Metaobjekte, wie zum Beispiel Funktionsaufrufmechanismen.

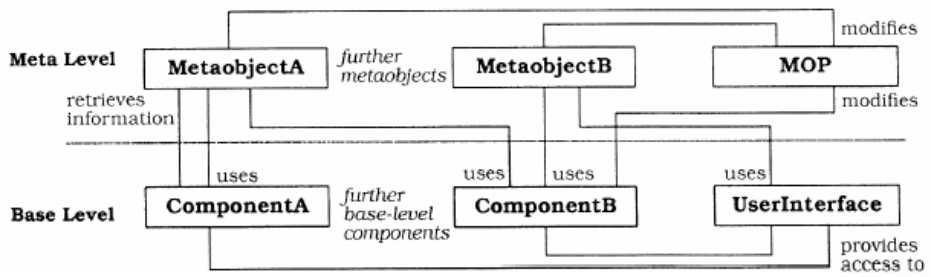

Abb. 10: Zusammenspiel der Ebenen und Schnittstellen beim Reflection Muster

Anwendung findet das Reflection Muster beispielsweise in dem Autohändlersystem „NEDIS“. Die reflexive Architektur soll hier der Anpassbarkeit an landes- und kundenspezifische Aspekte dienen. Die Zulässigkeit von Eingabewerten und

Funktionen, sowie das „look and feel“ der Software werden bei NEDIS mit Hilfe eines „run-time dictionary“ verändert. Ein großer Vorteil des Reflection Musters ist die Möglichkeit der einfachen Anpassung und Erweiterung, ohne den Code manuell ändern zu müssen. Darum kümmert sich das Metaobjektprotokoll, welchem die Änderungsspezifikation übergeben wird. Gleichzeitig sorgt das MOP für die Korrektheit der Änderungen. Auf Grund der beliebigen Anzahl an Metaobjekten und ihren gespeicherten Systeminformationen ist eine gute Skalierung möglich. Wenn jedoch fehlerhafte Änderungen vom MOP nicht erkannt werden, kann ihre Umsetzung zu empfindlichen Schäden im System führen. Einen weiteren Nachteil bringt die komplexe Beziehung zwischen den Ebenen mit sich. Sie existiert auf Kosten der Performanz.

4 Zusammenfassung und Fazit

Architekturmuster beschreiben erprobte Lösungen für häufig wiederkehrende Entwurfs- und Implementierungsprobleme bei der Entwicklung von Softwaresystemen auf höchster Abstraktionsstufe. Sie helfen dem Entwickler eine Auswahlentscheidung bei der Vielzahl an Entwurfsalternativen zu treffen und geben ihm mit Hilfe der Muster Entwurfsrichtlinien für eine Architektur vor, welche von Experten erarbeitet worden sind. Durch die standardisierte Beschreibungstechnik, bieten die Muster ein Standardvokabular, welches allgemein kommuniziert werden kann. So wissen Entwickler auf Grund des verwendeten Architekturmusters schon, wie das System strukturiert ist und wie mit ihm umzugehen ist. In der vorliegenden Arbeit wurden Architekturmuster aus drei Kategorien beschrieben. In der ersten Kategorie wurde auf Strukturierungsmuster eingegangen. Das Layers Muster hilft die Komplexität großer Systemen zu reduzieren, indem die Komponenten auf Schichten gleicher Abstraktionsebene verteilt und so strukturiert werden. Bei dem Pipes und Filters Muster steht die Koordination von Datenströmen im Mittelpunkt und das Blackboard Muster befasst sich mit Problemfeldern, für die noch keine feste Lösungsstrategie entworfen wurde. Bei dem Broker Muster handelt es sich um ein Muster aus dem Bereich „Verteilte Systeme“. Hier wird ein Vermittler verwendet um als Schnittstelle zwischen den verteilten Systemen zu agieren und so die Kommunikation zu steuern. In der letzten Kategorie „Adaptierbare Systeme“ werden die beiden Muster Microkernel und Reflection beschrieben. Beim Microkernel bietet ein möglichst kleiner Kern Grundfunktionalität und sorgt andererseits für die Koordination von Erweiterungskomponenten, die für die Umsetzung von sich ändernden Systemanforderungen zuständig sein können. Eine reflexive Architektur liefert das Reflection Muster mit dem Ziel grundlegende strukturelle und verhaltensmäßige Systemeigenschaften von der Anwendungslogik zu kapseln und so veränderbar zu halten. Die Vorteile und Schwerpunkte der einzelnen Architekturmuster führen dazu, dass sie zum einen in abgewandelten Formen und zum anderen gemeinsam eingesetzt werden können. Sie lassen sich der vorliegenden Aufgaben- und Problemstellung durch Kombination oder Variation einzelner Muster anpassen und bieten damit einen guten und hilfreichen Lösungsansatz bei der Softwareentwicklung.

5 Literaturverzeichnis

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern – Oriented Software Architecture, A System of Patterns,* Wiley & Sons, 1996.

[SW04] Uwe Schneider, Dieter Werner: *Taschenbuch der Informatik*, Fachbuchverlag Leipzig, 2004.

[Si03] Johannes Siedersleben: *Softwaretechnik, Praxiswissen für Softwareingenieure*, Hanser, 2003.

[He99] Helmut Herold: *Linux-Unix-Shells, Bourne-Shell, Korn-Shell, C-Shell, bash, tcsh,* Addison-Wesley, 1999.