4 Architekturelle Prinzipien, Bausteine und Muster

Jedes Software-System hat eine Architektur, welche die aktuelle Organisation des Software-Systems darstellt, unabhängig davon, wie sorgfältig diese Organisation ausgewählt worden ist. Trotzdem kann man eine „gute“ Architektur intuitiv von einer „schlechten“ Architektur unterscheiden. Da dieses Spektrum zwischen einer guten und einer schlechten recht breit ist, präsentieren wir in diesem Abschnitt zuerst einige Qualitätsattribute von Software-Architekturen, bevor wir danach auf die einzelnen architekturellen Prinzipien und Bausteine näher eingehen.

4.1 Qualitätsattribute einer Software-Architektur

In der Literatur werden einige Qualitätsattribute zur Bewertung von architekturellen Entwürfen vorgeschlagen. Wir wollen hier jene von Bass, Clements und Kazman speziell herausgreifen, da diese Attribute breiten Anklang in der Beurteilung von Software-Architekturen gefunden haben (Bass et al. 1998).

Auf der Ebene eines Software-Systems können zwei große Kategorien von Qualitätsattributen unterschieden werden, an denen ein Software-System gemessen werden kann:

- *Beobachtbar durch Ausführung*: Wie gut erfüllt ein Software-System seine Verhaltensanforderungen während der Ausführungszeit? Liefert es die geforderten Resultate in der vorgesehenen Zeit? Sind die Resultate korrekt oder innerhalb der vorgesehenen Genauigkeit? Funktioniert das System in Zusammenarbeit mit anderen weiteren Software-Systemen?
- *Nicht beobachtbar durch Ausführung*: Wie leicht ist es, das System zu integrieren, zu testen und zu modifizieren? Wie aufwendig (in Form von Zeit und Kosten) war es zu entwickeln? Wie lange war die Produkteinführungszeit (*time-to-market*)?

Information aus dem beobachtbaren Bereich liefert keinerlei Antworten zu Fragen aus dem nicht beobachtbaren Bereich. So können beispielsweise Software-Systeme, die ihre Laufzeit-Anforderungen erfüllen, unterschiedlich lange (und kostenintensiv) entwickelt worden sein. Andererseits müssen leicht modifizierbare Systeme nicht notwendigerweise ihre funktionalen Anforderungen erfüllen. Auch innerhalb dieser Kategorien können keine sicheren Schlüsse gezogen werden: Ein

stabiles Software-System muss nicht automatisch auch korrekte Ergebnisse liefern. Genauso können Systeme, die in wenigen Wochen entwickelt worden sind, Monate für bestimmte Modifikationen erfordern.

Qualität muss daher in allen Phasen von der Analyse, über das Design, die Implementierung bis hin zur Inbetriebnahme (*deployment*) berücksichtigt und mittels verschiedener Qualitätsmaße ausgedrückt und überprüft werden. Nicht jedes Qualitätskriterium hat einen Einfluss auf die Architektur des Software-Systems: Benutzerfreundlichkeit ist mehr ein Kriterium für den Anwender der Software und nicht der Architektur selbst. Die Ausprägung einer Benutzerschnittstelle und deren Realisierung mittels diverser Software-Komponenten oder Programmbibliotheken hat andererseits wiederum sehr wohl Einfluss auf die Modifizier- und Änderbarkeit eines Software-Systems.

Modifizierbarkeit ist ein an sich stark architekturelles Attribut. Ein System ist modifizierbar, wenn Änderungen nicht viele und nicht große Teile des Systems betreffen. Performanz hat demgegenüber sowohl architekturelle als auch nicht-architekturelle Bedeutung: Einerseits hängt diese zum Teil von der erforderlichen Kommunikation zwischen Komponenten und der den Komponenten jeweils zugeordneten Funktionalitäten ab (architekturell); andererseits ist die Wahl und die Kodierung des Algorithmus für die Performanz wesentlich (nicht-architekturell).

Bass et al. fassen zusammen, was hinsichtlich Architektur und Qualitätsattributen wesentlich ist:

- Architektur ist kritisch für die Realisierung von vielen Qualitätsattributen eines Software-Systems. Diese Qualitätsattribute sollten auch auf der Ebene der Architektur evaluiert werden.
- Manche Qualitätsattribute sind nicht architekturabhängig und ein Erreichen dieser durch architekturelle Mittel ist keineswegs zielführend.

Jede Diskussion von Qualitätsattributen bedingt darüber hinaus, dass kein Qualitätsmerkmal alleine und unabhängig von anderen betrachtet und realisiert werden kann. Qualitätsattribute sind nicht orthogonal und können nur im gegenseitigen Wechselspiel unter Berücksichtigung von sinnvollen und erreichbaren Ausprägungen anderer Charakteristiken erreicht werden.

Zu den zur Laufzeit beobachtbaren Qualitätsattributen zählt man u.a. Performanz, Sicherheit, Verfügbarkeit, Funktionalität oder Benutzerfreundlichkeit. Nicht zur Laufzeit beobachtbar sind vielmehr Modifizier- und Änderbarkeit, Portabilität, Wiederverwendbarkeit, Integrierbarkeit oder Testbarkeit. Darüber hinaus gibt es noch so genannte Geschäfts-Qualitätsattribute, die ein Software-System prägen: Produkteinführungszeit, Kosten, projektierte Einsatzzeit des fertigen Systems, Marktausrichtung, Markteinführungsplan oder auch Integration mit anderen, existierenden Systemen.

Tabelle 4.1 gibt einen Überblick über die soeben diskutierten Qualitätsattribute und deren architekturellen Einfluss auf die Architektur.

Tabelle 4.1. Überblick über Qualitätsattribute (nach Bass et al. 1998)

Qualitätsattribut	architekturell?	architekturelle Aspekte
Erkennbar durch Beobachtung während der Ausführung		
Performanz	ja	Kommunikation zwischen Komponenten; Aufteilung zur Ausnützung von Parallelität
Sicherheit	ja	Spezialisierte Komponenten; z.B. Authentifizierungsserver
Verfügbarkeit	ja	Fehlertoleranz mittels redundanter Komponenten
Funktionalität	nein	Interaktion mit anderen Qualitätsattributen
Verwendbarkeit (*usability*)	ja	Modifizierbarkeit unterstützt dieses; Effizienz in Beziehung mit Performanz
Nicht erkennbar durch Beobachtung während der Ausführung		
Modifizierbarkeit	ja	Modularisierte, gekapselte Komponenten
Portabilität	ja	Portabilitätsschicht
Wiederverwendbarkeit	ja	Lose Kopplung von Komponenten
Integrierbarkeit	ja	Kompatible Verbindungen; konsistente Komponenten Schnittstellen
Testbarkeit	ja	Modularisierte, gekapselte Komponenten

Zusätzlich zu den Qualitäten eines Systems und jenen aus dem Geschäftsbereich gibt es einige Qualitätsattribute, die direkt einer Software-Architektur zugeordnet werden können und als solche durch einen Entwurf erreicht werden sollten.

Für Software-Design gelten einige Axiome, die teilweise auch aus anderen Disziplinen stammen (Witt et al. 1994; Mills et al. 1986):

- *Separation of Concerns*: Dieses Konzept als ein Kernteil im Software-Engineering hat das Ziel, jene Teile eines Software-Systems zu identifizieren, zu kapseln und zu manipulieren, die relevant für eine bestimmte Angelegenheit oder Sache konkret zuständig sind. Solche *Concerns* sind eine primäre Motivation für die Zerlegung von Software in handhabbare und verständliche Teile (i.e. Dekomposition). Beispiele von *Concerns* sind Features, Aspekte, Rollen, Variation, Konfiguration etc.
- *Comprehension*: Die intellektuelle Beherrschung von vielen auf den Entwickler oder Designer einwirkenden Informationen und deren Unterschiedlichkeit ist ebenso eine Grundlage für ein gutes Design. Die experimentell definierte Regel „sieben plus oder minus zwei“ von (Miller 1956) ist gut als Maß für eine notwendige Gruppierung, Abstraktion oder Zusammenfassung von Informationen im Rahmen des architekturellen Entwurfs verwendbar.

- *Korrektheit und Vollständigkeit*: Ein korrektes und vollständiges Design (i.e. ein Design, das die Spezifikation von Schnittstelle(n) und Verhalten erfüllt) bleibt unbeeinflusst durch den Wechsel äquivalenter Kontexte. Plattformen, die gleiche Services anbieten, ändern somit nicht die Korrektheit von Software-Systemen.
- *Ersetzbarkeit*: Wird in einem korrekten und vollständigen Design eine Komponente ausgetauscht, so bleibt das gesamte Design korrekt, sofern die neu integrierte Komponente Schnittstellen und Verhaltensspezifikation der getauschten Komponente erfüllt. Mills et al. 1986 bezeichnen diese Eigenschaft auch als referenzielle Transparenz. Design-Elemente höherer Ebenen können Referenzen auf Elemente niedrigerer Ebenen aufweisen, aber wie diese Spezifikationen erfüllt werden, ist transparent (unsichtbar) für die höheren Ebenen.

Darauf aufbauend haben sich einige wichtige Design-Prinzipien herauskristallisiert, die für die Beurteilung eines guten oder weniger guten Designs herangezogen werden sollten:

Ein Design sollte **modular** sein. Entwürfe sollten aus leicht austauschbaren, eigenständigen und in sich abgeschlossenen Teilen bestehen, sodass sowohl die Entwicklung des Systems als auch die spätere Wartung gut unterstützt wird. Ein modulares Design kann wesentlich zur Eingrenzung von Design-Änderungen beitragen: Teile können geändert werden, ohne dass andere Teile dadurch berührt werden und ebenfalls geändert werden müssen. Die Modularität bezieht sich auf das Design von Komponenten, aber nicht auf Änderungen von Schnittstellen oder gar Verhalten. Dadurch können Software-Entwickler Änderungen auch lokal vornehmen, ohne das gesamte Software-System im Detail zu kennen. Modularität ist umso bedeutender, je größer das zu entwickelnde Software-System ist: Solche Projekte werden von vielen Entwicklern parallel durchgeführt und es ist daher absolut notwendig, eine geeignete und effektive Arbeitsaufteilung mit wenig Abhängigkeiten zu definieren. Neben dem Kommunikations-Overhead ist auch das notwendige Verständnis von allzu vielen (wenn nicht allen) Teilen des Software-Systems zeitintensiv und gar hinderlich. Sowohl während der Entwicklung als auch nach Auslieferung sollten die Teile leicht austauschbar und somit ein Redesign oder eine Weiterentwicklung von kleinen Teilen leicht möglich sein.

Designs sollten **portabel** sein. Ein Entwurf oder Teile daraus sollte in anderen Umgebungen wieder verwendbar sein. Das Ziel hierbei ist, jene Teile zusammenzufassen, die von der Ausführungsumgebung oder der Plattform abhängig sind, um somit bei einer Portierung so wenig wie möglich ändern oder ersetzen zu müssen. Wenngleich heutige Ansätze im Bereich virtueller Maschinen (z.B. Java VM) wesentlich dazu beitragen, ist dennoch stets darauf zu achten, dass beispielsweise Hardware-Abhängigkeiten gekapselt werden (z.B. in einem *Hardware Hiding Module*).

Designs sollten **formbar** sein. Die Anpassung an sich ändernde Anforderungen des Benutzers ist ein weiteres wesentliches Design-Prinzip. Änderungen, die aus der Welt des Endbenutzers stammen, oder zusätzliche zu verarbeitende Informationen sollten kein komplettes Redesign des ursprünglichen Entwurfes nach sich

ziehen. Die Formbarkeit des Designs ist gleich zu Beginn eines Software-Projektes ins Auge zu fassen. Auch hier gilt es wieder, Details zu verbergen, Repräsentationen von Daten zu kapseln, Algorithmen zu generalisieren und schmale Schnittstellen anzubieten sowie algorithmische Optimierungen nach außen zu verbergen.

Ein Design sollte **konzeptuelle Integrität** aufweisen. Diese ist die zugrunde liegende Vision, die das Design des Systems auf allen Abstraktionsebenen vereinheitlicht. Der Entwurf sollte ähnliche Dinge auf ähnliche Art und Weise durchführen. Für Brooks ist die konzeptuelle Integrität überhaupt die wichtigste Überlegung und Eigenschaft für ein System-Design (Brooks 1975; Brooks 1995).

Ein Design sollte unter **intellektueller Kontrolle** sein. Dies bedeutet, dass alle für die Korrektheit Zuständigen die Komplexität in Form und Inhalt detailliert verstehen. Haben die Designer ein hohes Vertrauen in die Korrektheit ihres Designs vor der Implementierung, so kann man sagen, dass der Entwurf unter intellektueller Kontrolle ist. Wenn die Designer erst durch Testen der Implementierung die Korrektheit feststellen wollen, kann man dieses Design-Prinzip nicht attestieren.

Die *Buildability* (**Herstellbarkeit**) ist ein Prinzip, das dem verfügbaren Team unter gegebenen zeitlichen Rahmenbedingungen die Herstellung des Systems erlaubt und dabei auch die Möglichkeit von Anpassungen während des Entwicklungsprozesses einschließt. Ein weiterer Aspekt der Herstellbarkeit ist auch die Tiefe des Wissens über eine Problemstellung. Ein Design, das bekannte Konzepte und Technologien einsetzt, ist leichter umzusetzen als jenes, das mit lauter neuen Konzepten aufwartet.

Lose Kopplung und hohe Kohäsion sind weitere wichtige Design-Prinzipien. Kopplung (*coupling*) und Kohäsion (*cohesion*) sind qualitative Maße zur Evaluierung der Gruppierung von Design-Teilen basierend auf ihren Abhängigkeiten untereinander. Kopplung ist ein abstraktes Konzept, das die Wahrscheinlichkeit betrachtet, dass ein Entwickler, wenn er ein Modul entwickelt oder ändert, auch andere, weitere Module betrachten, verstehen oder ändern muss (Yourdon u. Constantine 1978). Faktoren, welche die Kopplung beeinflussen, sind die Art der Verbindung zwischen Modulen, die Komplexität der Schnittstelle, die Art des Informationsflusses über die Schnittstelle sowie die Bindungszeit der Module (z.B. zur Übersetzungs- oder Laufzeit etc.). Kohäsion andererseits betrachtet jedes Modul in Isolation und wie dessen interne Elemente zueinander verbunden oder in Relation stehen. Kopplung und Kohäsion stehen miteinander in Wechselbeziehung: Je höher die Kohäsion individueller Module eines Systems, desto geringer ist die Kopplung zwischen den Modulen.

Eine der wichtigsten Eigenschaften eines guten Designs ist, dass jede Ebene der Dekomposition eines Systems durch lose Kopplung und hohe Kohäsion eines jeden Teils (oder Moduls) charakterisiert ist. Lose Modul-Kopplung ist anstrebenswert, um die Schnittstellen leichter zu spezifizieren; hohe interne Kohäsion erlaubt andererseits, Verhalten leicht zu abstrahieren und zu spezifizieren.

Lose Kopplung wird ausgezeichnet durch:

- Design-Unabhängigkeit: Jedes Design-Element kann unabhängig entworfen werden, und spätere Modifikationen erfordern keine Änderungen anderer Design-Teile. Das erfordert, dass die vereinbarten Schnittstellen beibehalten werden und jeder Teil genau das tut, was als seine Aufgabe spezifiziert wurde.
- Schmale Schnittstellen: Die Anzahl verschiedener Datenelemente, die über die Schnittstelle ausgetauscht werden müssen, ist klein.
- Wenig Schnittstellenverkehr: Die Häufigkeit von Informationsaustausch zwischen Design-Teilen via Schnittstelle ist gering.

Hohe Kohäsion zeichnet sich aus durch:

- Einheit: Verwandte Problemstellungen und Anforderungen werden in dieselbe Design-Abstraktion aufgenommen. Die Anforderungen können dadurch einfacher ausgedrückt werden und damit nimmt der Grad der willkürlichen Zuordnung ab.
- Kapselung: Alle voneinander abhängigen Design-Teile werden zusammengefasst.

Zu guter Letzt wollen wir noch das **Design-for-change**-Prinzip diskutieren. Dieses Prinzip, u.a. von Parnas im Kontext von Software-Evolution beschrieben (Parnas 1994), umfasst viele Konzepte: Abstraktion, *information hiding*, *separation of concerns*, *data hiding* oder auch Objektorientierung. Um dieses Prinzip erfolgreich anzuwenden, sind die wahrscheinlichen Änderungen im Software-System, über die Lebenszeit des Produkts gerechnet, vorab zu charakterisieren:

- Erwartete Änderungen für das zu entwickelnde System oder Änderungen, die in ähnlichen Systemen bereits aufgetreten sind. Die Erfahrungen aus anderen Projekten und System-Entwicklungen sind dafür hilfreich.
- Erwartete Änderungen, die bei einer Analyse der Anforderungsspezifikation aufgrund von Unschärfen oder Mehrdeutigkeiten auftauchen. Dies kann beispielsweise auf zu erwartende Funktionalität hindeuten.
- Einschränkungen der Funktionalität aufgrund budgetärer Zwänge. Dies deutet auf ausstehende Features und Funktionen hin, die zwar derzeit nicht implementiert werden sollen, aber ursprünglich durchaus geplant waren.

Eine Architektur sollte daher so gewählt werden, dass diese an die erwarteten oder prognostizierten Änderungen gut anpassbar ist.

4.2 Architekturelle Stile

Ein architektureller Stil ist ähnlich zu einem Stil aus der Architektur. Er wird durch bestimmte Eigenschaften charakterisiert und bildet die Vorlage (den Bauplan) für konkrete Software-Systeme dieser Art. Ein architektureller Stil be-

schreibt somit eine Familie von Software-Systemen, die aufgrund struktureller und semantischer Eigenschaften verwandt sind. Ein architektureller Stil ist eine spezielle Design-Sprache für eine Klasse von Systemen und stellt folgende vier Teile zur Verfügung (Monroe et al. 1997):

- Ein Vokabular von Design-Elementen: Typen von Komponenten (z.B. Pipes, Filter, Server, Parser oder Datenbanken) oder Konnektoren (z.B. Unterprogramm-Aufrufe, *remote procedure calls*, *data streams*, *sockets* etc.).
- Design-Regeln und Einschränkungen (*constraints*) legen fest, welche Kompositionen der Elemente zulässig sind. Beispielsweise könnten diese Regeln es verbieten, dass es Zyklen in einem bestimmten Pipe-and-Filter-Stil gibt, oder spezifizieren, dass es in einer Client-Server-Organisation stets n:1 Beziehungen gibt.
- Eine semantische Interpretation, wobei die Kompositionen von Design-Elementen (unter Berücksichtigung der *constraints*) eine klar definierte Semantik aufweisen.
- Analysen, um zu überprüfen, ob ein System einem bestimmten architekturellen Stil entspricht. Beispiele dafür wären Deadlock-Erkennung für Client-Server-Nachrichtenaustausch oder Schedulability-Analyse für Echtzeit-Verarbeitung.

Perry und Wolf haben den Begriff des architekturellen Stils wie folgt definiert (Perry u. Wolf 1992):

> Ein architektureller Stil definiert eine Familie von Software-Systemen bezüglich ihrer strukturellen Organisation. Ein architektureller Stil drückt die Komponenten und Relationen zwischen diesen gemeinsam mit den Einschränkungen ihrer Anwendung, der assoziierten Komposition und den Design-Regeln für deren Konstruktion aus.

Daraus folgt, dass ein architektureller Stil eine bestimmte Art von fundamentaler Struktur für ein Software-System gemeinsam mit den assoziierten Methoden für ihre Konstruktion festlegt. Ein architektureller Stil beinhaltet ebenso Informationen darüber, wann man den Stil oder seine Varianten bzw. Spezialisierungen am besten einsetzt und welche Konsequenzen der Einsatz für das zu entwickelnde Software-System mit sich bringt.

Shaw und Garlan haben einige weit verbreitete architekturelle Stile katalogisiert (Shaw u Garlan 1996). Architekturelle Stile treten nicht nur regelmäßig in System-Entwürfen auf, sie kommen oft in leichten Abwandlungen vor. Stil-Kataloge wie in Abb. 4.1 aus (Shaw u. Garlan 1996) helfen somit herauszufinden, ob zwei Stile verwandt sind.

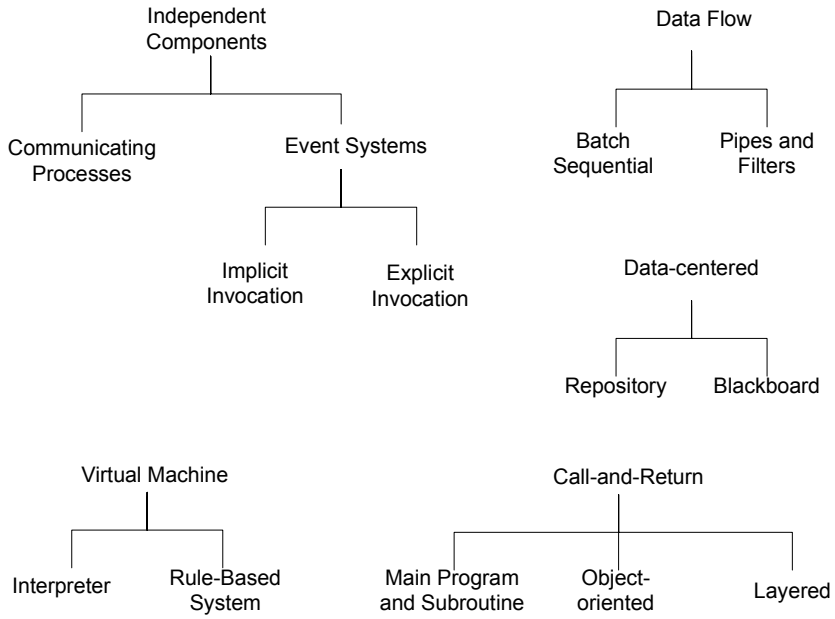

Abb. 4.1. Katalog von architekturellen Stilen

4.2.1 Datenzentrierte Software-Architekturen

Datenzentrierte Architekturen (*data centered*) beschreiben Systeme, bei denen der Zugriff und die Aktualisierung von Daten eines Repository vorrangiges Ziel sind. Dieses Repository kann nicht nur als Datenbank (passiv), sondern auch als aktiver Datenbehälter (*blackboard*) realisiert sein. Es handelt sich dabei um einen zentralisierten Datenspeicher, der mit Clients kommuniziert. Die Kommunikation bzw. Koordination teilt sich in zwei Arten: *Repository* und *Blackboard*. Das Blackboard sendet Benachrichtigungen an Interessierte (*subscriber*), wenn sich die entsprechenden Daten, für die sich subskribierte Komponenten interessieren, ändern. Dieser Stil gewinnt an Bedeutung, da jene Systeme, die bereits existierende Komponenten integrieren, Datenintegration durch die Verwendung von Blackboard-Mechanismen erreichen. Die Clients sind somit weitgehend unabhängig voneinander und der Datenspeicher ist ebenfalls unabhängig von seinen Clients. Dieser architekturelle Stil ist demzufolge gut skalierbar: Neue Clients können einfach hinzugefügt werden und Änderungen in Clients beeinflussen keine anderen Clients. Wichtig ist, dass, wenn die Clients als unabhängige Prozesse realisiert werden, es sich um einen anderen Stil handelt: unabhängige Komponenten (*independent components*).

4.2.2 Datenfluss-Architekturen

Datenfluss-Architekturen (*dataflow architectures*) zielen speziell auf Wiederverwendung und Modifizierbarkeit ab: Im Datenfluss-Stil wird das System als eine Reihe von Transformationen von aufeinander folgenden Eingabeströmen gesehen. Daten gelangen ins System, wandern Komponente für Komponente durch das System, werden dabei manipuliert und am Ende auf einem Ausgabestrom (oder ein anderes Medium) bereitgestellt. Dieser Stil hat zwei Ausprägungen: Pipe-and-Filter und Batch Sequential. Im Batch-Sequential-Stil sind die Komponenten unabhängig agierende Programme und diese werden nacheinander abgearbeitet. Eine Komponente beendet ihre Aufgabe, und erst dann beginnt die nächste in der Abarbeitungsreihenfolge. Dieser Stil stammt aus dem klassischen Datenverarbeitungsbereich der EDV, realisiert vor allem in Host-basierten Systemen.

In einem Pipe-and-Filter(P-F)-Stil transformiert jede Komponente einen Eingabestrom in einen Ausgabestrom von Daten durch lokale Transformationen (so genannte Filter). Der Ausgabestrom wird dabei bereits auf die Pipe geschrieben, noch bevor der gesamte Eingabestrom konsumiert worden ist. Beispiele dafür kommen aus dem Unix-Bereich: auflisten (`cat`) → Strings finden (`grep`) → sortieren (`sort`) → ausgeben (`more`) etc. Die Pipes sind in diesem Stil als Konnektoren, die Programme als Komponenten anzusehen. Häufige Spezialisierungen dieses Stils sind so genannte Pipelines – eine Verknüpfung zahlreicher Filter in einer konkreten Topologie (i.e. ein Zusammenspiel spezieller Filter und Pipes in einer wohl definierten Reihenfolge) –, *bounded pipes* (mit limitierten Speicherfähigkeiten der Pipes) oder *typed pipes*, die den Typ der verarbeitbaren Daten festlegen. Abb. 4.2 zeigt einen P-F-Stil für das noch folgende KWIC-Beispiel.

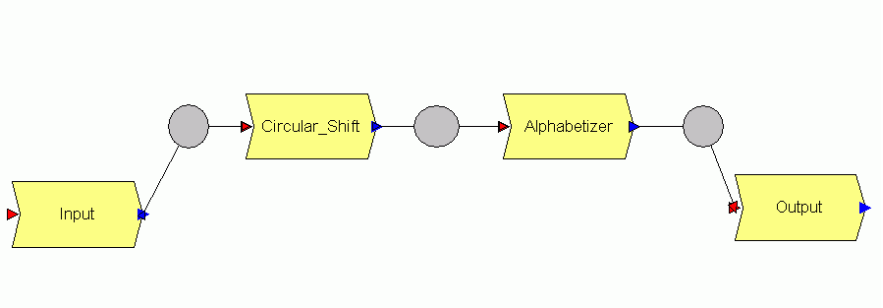

Abb. 4.2. KWIC-System im Pipe-and-Filter-Stil

Ein Vorteil von P-F-Architekturen ist in deren Einfachheit: Die Funktionalität wird durch das Zusammenwirken der einzelnen Filter über die durch die Pipes definierten Verbindungen festgelegt. Es gibt darüber hinaus keine komplexen Komponenten-Interaktionen zu verwalten. Filter können weiters als Black Box gesehen werden und somit leicht durch andere Filter ersetzt werden. Das System ist leicht modifizierbar und erweiterbar. Hierarchische Kombinationen sind ebenfalls leicht durchführbar, wobei an der Schnittstelle eines äußeren Filters wiederum nur eine

Filtereigenschaft gegeben ist (und somit eine weitere Kombinier- und Ersetzbarkeit).

Andererseits wird durch das „Zusammenstecken“ der Filter der Verarbeitungsvorgang definiert. Filter können kaum kooperativ eine Problemstellung lösen und Interaktivität ist ebenfalls nicht umsetzbar, da der Stil dafür ungeeignet ist. Die Performanz von P-F-Systemen ist wegen zahlreicher Eigenschaften dieses Stils in vielen Fällen nicht optimal (Bass et al. 1998):

- Filter verlangen nach dem kleinsten gemeinsamen Nenner der Datenrepräsentation (d.h. meist nach ASCII). Wird der Datenstrom in Tokens analysiert, so hat jeder Filter für sich das Parsen und Unparsen extra (und damit vielfach im System) zu erledigen.
- Kann ein Filter seine Aufgabe nicht ohne vollständiges Einlesen des Eingabestroms erledigen, benötigt er theoretisch einen Puffer unlimitierter Größe. Beispielsweise weist ein Sortier-Filter genau diese Problematik auf. Werden darüber hinaus noch *bounded pipes* verwendet, können auch Deadlocks auftreten.
- Operiert jeder Filter auch als separater Prozess oder Prozeduraufruf, so ist mit den jeweiligen Aufrufen zusätzlicher Overhead verbunden.

4.2.3 Virtuelle Maschinen-Architekturen

Virtuelle Maschinen-Architekturen (*virtual machines*) zielen mitunter auf Portabilität des Software-Systems ab. Virtuelle Maschinen simulieren Funktionalität, die in einer eigenen Schicht über Hardware- oder Software-Plattformen ablaufen und damit unabhängig von der aktuellen Hard- oder Software-Plattform sein sollen. Dies erlaubt auch neu zu bauende Plattformen (z.B. Hardware) zu simulieren und zu testen (wie von Flugsimulatoren oder anderen sicherheitskritischen Systemen bekannt).

Bekannte Beispiele für diesen architekturellen Stil sind Interpreter und regelbasierte Systeme oder Kommandosprachen-Verarbeitung. So ist beispielsweise die Programmiersprache Java in einer Java-Virtuellen Maschine ablauffähig, um die Sprache somit plattformunabhängig zu realisieren. Eine typische VM-Architektur ist in Abb. 4.3 dargestellt und besteht aus folgenden Teilen: das interpretierte Programm, die Programm-Daten (z.B. die Werte von Variablen, die während der Programmausführung zugewiesen werden), der interne Zustand des Interpreters (z.B. die Werte von Registern der gerade ausgeführten Anweisung).

Die Interpretationsmaschine wählt eine Anweisung des Programms, aktualisiert ihren inneren Zustand und aktualisiert abhängig von der konkreten Anweisung auch den Zustand der Programm-Daten. Eine solche Programmausführung erlaubt Flexibilität, sodass das interpretierte Programm angehalten und sein jeweiliger Zustand abgefragt werden kann. Die Interpretation eines Programms benötigt andererseits zusätzliche Rechenressourcen und ist daher als ein Nachteil solcher VM-Architekturen anzusehen.

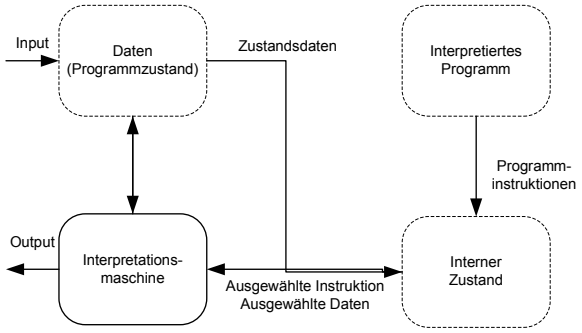

Abb. 4.3. Virtuelle Maschinen-Architektur

4.2.4 Call-and-Return-Architekturen

Diese Architekturen sind lange Zeit der dominierende architekturelle Stil großer Software-Systeme gewesen. Innerhalb der Call-and-Return(CR)-Architekturen haben sich besondere Substile ausgeprägt.

Die **Hauptprogramm-Unterprogramm-Architektur** (*main program and subroutine architecture*) repräsentiert das klassische Programmier-Paradigma. Es gibt einen singulären Kontrollfluss und jede Komponente in der Hierarchie bekommt sukzessive die Kontrolle von ihrer übergeordneten Komponente, gibt diese jeweils an untergeordnete Komponenten weiter, und anschließend wird diese Kontrolle wieder an die ursprünglich aufrufende Komponente nach oben in der Hierarchie zurückgegeben.

Die **Remote-Procedure-Call(RPC)-Architektur** ist eine CR-Architektur, wobei die Komponenten auf unterschiedlichen Rechnern beheimatet und über ein Netzwerk miteinander verbunden sind. Berechnungen werden auf mehrere Rechner verteilt und nützen somit Parallelität und mögliche Lastverteilung aus.

Objektorientierte oder abstrakte Datentyp-Architekturen sind die moderne Variante von CR-Architekturen und fokussieren auf die Kapselung von Daten und das *information hiding*. Zugriffe auf Objekte erfolgen nur über deren Schnittstellen. Der Zugriff folgt den Prinzipien eines CR, stellt aber eine eingeschränkte Form von Prozeduraufrufen dar. Dieser Stil zielt vor allem auf die *separation of concerns* ab, die wir als Design-Prinzip bereits diskutiert haben. Handelt es sich bei den Abstraktionen um Komponenten, die als Black Box ihre Services anbieten und als solche von anderen Komponenten aufgerufen werden, so spricht man von einem *aufrufbasierten Client-Server-Stil* (im Gegensatz zu einem prozessbasierten Client-Server-Stil, der nachfolgend beschrieben wird).

In **Schichten-System-Architekturen (*layered architectures*)** werden die Komponenten eigenen Schichten zugeordnet. Diese Schichten sind hierarchisch organisiert und erfüllen eine klar definierte Aufgabe für die Schicht darüber unter Verwendung der Dienste der darunter liegenden Schicht. Jede Schicht für sich rep-

räsentiert und implementiert eine abstrakte virtuelle Maschine, auf der die darüber liegende Schicht operiert. Die Konnektoren sind durch die Protokolle für die Schichten-Interaktion definiert. Topologische Einschränkungen limitieren die Interaktion auf unmittelbar benachbarte Schichten. Die unterste Schicht realisiert Kernfunktionalität wie z.B. diverse Betriebssysteme. Die nächste Schicht baut auf dieser untersten Schicht und deren Funktionalität auf; so wird Schicht auf Schicht gesetzt, wobei jede Schicht ihre Dienste der jeweils unmittelbar darüber liegenden Schicht anbietet. Bekannte Beispiele für Schichten-Architekturen stammen aus dem Bereich der Kommunikation: Das OSI-7-Schichten-Modell oder auch das TCP/IP-4-Schichten-Modell sind Systeme, die diesem architekturellen Stil folgen (siehe u.a. Tanenbaum u. van Steen 2002).

Schichten-Architekturen erlauben zunehmende Abstraktionsebenen im Design. Komplexe Probleme können so in eine Sequenz kleinerer zerlegt werden. Weiters unterstützt dieser Stil die Weiterentwicklung eines Systems: Da jede Schicht mit höchstens zwei benachbarten Schichten kommuniziert, betreffen Änderungen der Funktionalität einer Schicht maximal die benachbarten Schichten. Änderungen, die keine Modifikation der Schnittstellen nach sich ziehen, betreffen ohnedies nur eine Schicht. Auch die Wiederverwendung wird durch diesen Stil gefördert, da unterschiedliche Implementierungen derselben Schicht auswechselbar sind (sofern damit keine anderen Schnittstellen einhergehen). Dies erlaubt es, Standard-Schnittstellen zu definieren, die sogar von unterschiedlichen Herstellern gebaut werden können.

Schichten-Architekturen haben aber auch Nachteile: Nicht jedes System lässt sich sinnvollerweise in Schichten abbilden. Selbst wenn man ein System logisch in Schichten einteilen kann, so können Performanz-Anforderungen eine enge Kopplung von Komponenten über Schichten hinweg erfordern. Oftmals braucht man eine Überbrückung zwischen Schichten (*layer bridging*), da eine Schicht u.U. mit einer anderen als der direkt benachbarten kommunizieren muss. Dies wiederum bricht die ansonsten klare Struktur und kann zu Portabilitätsproblemen führen und die obigen Vorteile aufweichen bzw. sogar aufheben. Wird das Prinzip der Schichtung andererseits konsequent eingehalten, erfordert eine Portierung nur die Re-Implementierung einer Schicht. Eine Java VM wird beispielsweise für verschiedene Plattformen geschrieben, die Java-Programme selbst, die auf dieser Schicht aufbauen, benötigen selbst keine Änderungen.

4.2.5 Unabhängige Komponenten-Architekturen

Diese Architekturen bestehen aus einer Anzahl unabhängiger Prozesse oder Objekte, die über Nachrichten miteinander kommunizieren. Solche Architekturen haben das Ziel, Modifizierbarkeit durch die Entkopplung von Berechnungen in diversen Rechnern zu erreichen. Diese Systeme tauschen Daten nicht durch direkte Kontrolle aus, sondern Nachrichten werden als Datenmedium verwendet, um Informationen zu bekannten oder unbekannten Komponenten zu vermitteln.

Ereignisgesteuerte (Event-basierte) Systeme stellen einen besonders interessanten Typ dieses architekturellen Stils dar. Einzelne Komponenten geben Informationen bekannt (*publish*), die sie mit interessierten, aber weitgehend unbekannten Komponenten ihrer Umgebung teilen wollen. Die interessierten Komponenten registrieren sich für eine bestimmte Art von Information (*subscribe*). Publisher- und Subscriber-Komponenten kennen einander nicht und werden nur indirekt über ein Event-System miteinander gekoppelt: Das Event-System ermöglicht die Kommunikation zwischen den Komponenten, indem es bei Auftreten einer entsprechenden Nachricht die interessierte Komponente (den *subscriber*) davon unterrichtet.

Im *Publish-Subsribe*-Paradigma registrieren sich Komponenten für bestimmte Arten von Informationen, an denen sie interessiert sind, und geben andere Arten von Informationen bekannt, die sie zur Verfügung stellen können. Information wird publiziert, indem diese an das Event-System weitergeleitet wird. Das Event-System leitet die Nachrichten (Informationen) dann an alle dafür registrierten Komponenten weiter. Damit werden die einzelnen Komponenten voneinander entkoppelt: Für eine einzelne Komponente ist es nicht erforderlich, die Namen oder Orte anderer zu kennen (und zu verwalten). Das Event-System übernimmt genau diese Aufgabe und erlaubt somit eine Ebene der Indirektion. Dieses architekturelle Prinzip erlaubt auch die Entkopplung von Kontrolle: Komponenten arbeiten so unabhängig und parallel wie möglich und tauschen nur bei Bedarf asynchron Nachrichten (Informationen) aus. Solche Arten von Systemen erlauben es neuen Komponenten, sich beim Event-System zu registrieren und somit das gesamte System leicht erweiterbar, wartbar und integrierbar zu halten.

Neben den Event-basierten Systemen gibt es noch einen weiteren architekturellen Stil von unabhängigen Komponenten: die kommunizierenden *Prozesse (communicating processes*). Dies sind Multiprozess-Systeme wie beispielsweise Client-Server. Ein Server stellt Daten für einen oder mehrere Clients im Netzwerk zur Verfügung, die diese Informationen auf synchronem oder asynchronem Wege vom Server abholen. Die dabei verwendeten Kommunikationsprotokolle sind beispielsweise RPC (Remote Procedure Call im Falle von C/C++) oder RMI (Remote Method Invocation im Falle von Java).

Eine spezielle Art stellt der *Peer-to-Peer*-Stil dar, der in Kap. 7 noch detailliert beschrieben wird. In einem Peer-to-Peer(P2P)-Stil interagieren die Komponenten als Gleichrangige (i.e. Peers). Die Kommunikation ist eine Art von Anfrage/Antwort (*request/reply*) ohne die Symmetrie, die man in Client-Server-Systemen findet. D.h., eine Komponente kann mit anderen Komponenten durch Inanspruchnahme ihrer Services kommunizieren. Konnektoren dieses Stils können umfangreiche bidirektionale Interaktionsprotokolle umfassen, welche die Zwei-Weg-Kommunikation zwischen zwei oder mehreren Peers repräsentieren. P2P-Systeme basieren oft auch auf verteilten Objekt-Infrastrukturen wie CORBA, COM+ oder Java RMI. Peers können Objekte, verteilte Objekte oder auch Clients sein. Ungleich zu Client-Server kann die Kommunikation von jedem Peer initiiert werden (Peers agieren als Clients und als Server, je nach Bedarf und Services, die sie zur Verfügung stellen). Peers haben Schnittstellen, welche die Services beschreiben, die sie von anderen Peers anfordern bzw. selbst anbieten. Die Steuerung zwischen den einzelnen Peers ist symmetrisch: Peers initiieren Aktionen

durch Kooperation mit anderen Peers, um ihre Aufgaben zu erfüllen. Einschränkungen im Peer-to-Peer-Stil sind hinsichtlich der Anzahl der pro Peer zulässigen weiteren Peers oder aber auch im Wissen von Peers über andere im System existente Peers (Topologie-basiert) gegeben und im jeweiligen P2P-System und den darin verwendeten Protokollen umgesetzt (z.B. im Gnutella-Protokoll).

4.2.6 Heterogene architekturelle Stile

Software-Systeme werden selten mit nur einem architekturellen Stil gebaut. Vielmehr spielen mehrere der zuvor beschriebenen Stile für diverse Aufgaben und Verantwortlichkeiten im System zusammen. So kann eine Kommunikationsschicht als Dienst die zuverlässige Übertragung von Daten bereitstellen, dies aber beispielsweise auf Peer-to-Peer-Art und Weise erledigen. Komponenten, die auf der Kommunikationsschicht aufsetzen, können wiederum einem anderen architekturellen Stil zur Erledigung ihrer Aufgaben folgen. So sind zahlreiche industrielle Client-Server-Systeme, die mittels CORBA Middleware kommunizieren, als geschichtete objektbasierte Multiprozess-Systeme anzusehen und damit eine Kombination dreier architektureller Stile. Dieser heterogene Stil kann aufgrund seiner häufigen Anwendung durchaus auch als eigener architektureller Stil angesehen werden. So entstehen aus der Kombination und Integration diverser Stile manchmal neue heterogene Stile, beeinflusst auch durch neue Technologien wie beispielsweise Peer-to-Peer etc.

Heterogenität kann zumindest in drei Arten auftreten (Bass et al. 1998):

- Bereichsabhängige Heterogenität: Je nach Bereich oder Subsystem tritt der eine oder andere Stil hervor. Ein System kann im Wesentlichen einem architekturellen Stil folgen, dann aber in Teilbereichen andere Substile umsetzen. So können in einem unabhängigen Komponenten-Stil einzelne (oder auch nur eine) Komponenten einem Pipe-and-Filter-Stil folgen.
- Hierarchieabhängige Heterogenität: Komponenten, die sich hierarchisch in weitere Subkomponenten auffalten, können einem anderen Stil folgen; so kann beispielsweise eine Komponente eines Event-basierten Stils in sich einer Schichtung folgen.
- Simultane Heterogenität: Nicht jedes System kann nur mit einem Stil optimal beschrieben werden; manchmal können mehrere Beschreibungen und damit architekturelle Stile für ein System gleich passend sein.

Aus Obigem lässt sich folgern, dass architekturelle Stile die Software-Architekturen in orthogonale, nicht überlappende Kategorien einteilen lassen. Vielmehr stellen diese ideale Abstraktionen und Beschreibungen für bestimmte Eigenschaften oder Verantwortlichkeiten von Systemen oder System-Teilen dar.

4.2.7 Architekturelle Stile in ihrer Anwendung

Die Verwendung von geeigneten architekturellen Stilen für den Entwurf eines Software-Systems ist eine wesentliche Herausforderung für Software-Architekten und hängt von den zu erreichenden System-Qualitäten ab. Stile dienen als elementares Beschreibungsmittel für eine Software-Architektur und sind somit essenziell für die Darstellung systemkritischer und undeterminierter Teile. Regeln und Richtlinien für die effektive Anwendung sind dafür wichtig. Der Einsatz eines architekturellen Stils hängt stark von den angestrebten Qualitätsattributen des Software-Systems ab (z.B. Zuverlässigkeit, Performance, Fehlertoleranz, Sicherheit etc.) und sollte zuerst danach ausgerichtet werden. Eine geeignete Vorgangsweise ist es, mit der am besten geeigneten architekturellen Struktur zu beginnen und dann den Stil für diese Struktur auszuwählen, der die Qualitätsattribute am besten umsetzen lässt.

Bass et al. 1998 haben einige Faustregeln zur Auswahl des geeigneten architekturellen Stils für die jeweiligen Anforderungen und Rahmenbedingungen definiert, die wir in Tabelle 4.2 auszugsweise darlegen:

Tabelle 4.2. Faustregeln zur Auswahl des geeigneten architekturellen Stils

Architektureller Stil	anzuwenden, wenn …
Datenfluss	Das System produziert einen wohl definierten, einfach identifizierbaren Output, der das direkte Resultat von sequentiellen Transformationen aus wohl definierten Inputs in einer Zeit-unabhängigen Art und Weise ist. Die Integrierbarkeit (resultierend aus einfachen Schnittstellen der Komponenten) ist wesentlich.
• Batch sequential	• Eine singuläre Output-Operation ist das Resultat des Lesens einer Menge von Input-Daten und deren Transformationen sind sequenziell.
• Datenfluss	• Input und Output treten als wiederkehrende Serie auf, und es gibt einen direkten Zusammenhang zwischen entsprechenden Teilen jeder Serie.
– azyklisch	– … und die Transformationen beinhalten keine Feedback Loops.
– nur Fan-out-Komponenten	– … und die Transformationen beinhalten keine Feedback Loops und ein Input führt jeweils zu mehr als einem Output.
– Pipeline, Pipe-and-Filter	– Die Berechnung besteht aus Transformationen von kontinuierlichen Datenströmen. Die Transformationen sind inkrementell; eine Transformation kann beginnen, bevor der vorangehende Schritt abgeschlossen ist.
• Closed Loop Control	Das System überwacht andauernde Aktivität, ist in einem physikalischen System eingebettet und ist unvorhersehbaren externen Störungen ausgesetzt, sodass vordefinierte Algorithmen fehlschlagen können.
Call-and-Return	Die Reihenfolge der Berechnung ist fixiert und die Komponenten können ohne Ergebnisse von anderen Komponenten