# P2P und Komponenten –

# Eigenschaften, Unterschiede, Ergänzungsmöglichkeiten

*Markus Völter, voelter@acm.org, www.voelter.de*
*Thomas Quas*

**Peer-to-Peer (P2P) ist in aller Munde. Neben Webservices ist es ganz klar der neueste Hype (es gibt sogar schon eine Konferenz dazu!). Da fragt sich der etwas traditioneller veranlagte Softwareentwickler, der gerade erst den Komponentenhype verdaut hat: "Muss ich jetzt alles wegwerfen und wieder was neues lernen?" Keine Sorge, ganz so schlimm kommt es wohl nicht. Der vorliegende Artikel beleuchtet Komponententechnologien und P2P Systeme, zeigt ihre Unterschiede und Anwendungsgebiete, und versucht, Anwendungsfälle zu erläutern, wo sich beide Technologien ergänzen können.**

## Kompontenbasierte Systeme

Mit Komponentenarchitekturen hat die Entwicklergemeinde zwischenzeitlich einige Erfahrung, neben vielen proprietären Modellen existieren einige recht weit verbreitete: EJB, CCM, COM+ und .NET. All diese Komponentenmodelle beruhen auf den gleichen Mustern und lassen sich mit diesen recht gut beschreiben (siehe Kasten). Komponentenbasierte Systeme wie sie heute Verwendung finden zeichnen sich durch die folgenden Merkmale aus:

- **Zentralistisch:** Komponenten sind letztendlich eine Fortführung des Client/Server-Paradigmas. Komponenten liegen auf zentralen Maschinen und erbringen Dienste (im Gegensatz zu C/S nicht nur Datenhaltung) für "dünne" Clients; Komponenten sind also üblicherweise passiv. Diese Struktur minimiert die Ansprüche an Clients, sowohl bzgl. Rechenleistung und Resourcen als auch bezüglich Installation und Konfiguration.

- **Wohldefiniert:** Die Dienste die von einer Komponente erbracht werden sind über entsprechende Schnittstellen genau definiert (idealerweise Signatur *und* Semantik). Die Schnittstellen können - aufgrund der potentiell großen Anzahl Clients - nicht ohne weiteres verändert werden.

- **Vorkonfiguriert:** Ein komponentenbasiertes System ist üblicherweise recht starr konfiguriert. Die entsprechenden Komponenten sind vordefinierten Containern und Maschinen (bzw. Clustern) zugeordnet. Ein Client weiß genau, an wen er sich wenden muss, um eine bestimmte Dienstleistung zu erhalten.

- **Zuverlässigkeit:** Eine Komponente bzw. das darunterliegende System ist stabil. Eine definierte Verfügbarkeit und Service-Qualität muss eingehalten werden (Quality of Service, QoS).
- **Geschlossen:** Die Sicherheitsinfrastruktur eines komponentenbasierten Systems ist geschlossen. Nur registrierte Clients können, nach erfolgreicher Authentifizierung, die Dienste des Systems in Anspruch nehmen.

Die genannten Eigenschaften lassen Komponenten vor allem für abgeschlossene, wohlorganisierte Einsatzzwecke nützlich erscheinen. Natürlich ist dabei Raum für Weiterentwicklung, z.B. durch die Tatsache, dass sich die Implementierung einer Komponente ändern kann, ohne dass sich das Interface ändert - Clients müssen also nicht angepasst werden.

## Peer-To-Peer Systeme

Unserer Meinung nach ist es noch zu früh, für P2P Systeme eine kurze, prägnante und allgemeingültige Definition zu geben, dafür ist die Technologie noch zu wenig etabliert. Dasselbe gilt für Muster: etablierte Architektur- und Entwurfsmuster für P2P Systeme wurden noch nicht gefunden. Wir möchten trotzdem versuchen, einige Charakterisika von P2P Systemen anzugeben (Über Meinungen zu dem Thema würden wir uns sehr freuen - siehe Email-Adressen):

- **Dezentral:** In reinen P2P Systemen gibt es keine dedizierten Server oder Clients. Jedes teilnehmende System kann Dienste erbringen, Daten speichern oder Dienste in Anspruch nehmen. Jedes System stellt Dienste in dem Umfang zur Verfügung, wie es seine Resourcen erlauben. Der Zustand des Gesamtsystems ist (idealerweise redundant) auf alle teilnehmenden Systeme verteilt.
- **Flexibel:** Die Dienste die in P2P Systemen erbracht und verwendet werden, werden nirgends zentral verwaltet. Dienste können ihre Features und Interfaces jederzeit ändern - Clients müssen sich dann eben anpassen. Dies erlaubt eine schnellere, und vielschichtigere Anpassung der Dienste an sich verändernde Anforderungen. Spezialisierte Dienste können sich leicht entwickeln.
- **Dynamisch:** Dienste können jederzeit von einer anderen Maschine erbracht werden, oder eine Maschine kann aufhören einen Dienst anzubieten. Es gibt keine "definierte" Konfiguration des Systems, zu jedem Zeitpunkt hat das System potentiell eine andere Konfiguration. Clients müssen also jedesmal neu herausfinden, wer gerade einen bestimmten Dienst erbringt.
- **Keine QoS Garantien:** In P2P Systemen kann nicht garantiert werden, dass eine bestimmte Instanz eines Dienstes jederzeit verfügbar ist - das gilt auch für Datenspeicher. Auch die Performance oder andere QoS-Aspekte sind undefiniert und können sich jederzeit ändern. Die Nützlichkeit des Systems entsteht aus dem

(unkoordinierten) Zusammenspiel: Das System als Ganzes hat natürlich bestimmten Grad an Zuverlässigkeit, auch wenn die einzelnen Teilsysteme dies nicht haben.

- **Offen/Anonym:** In P2P Systemen gibt es üblicherweise keine geschlossene, zentral verwaltete Benutzergemeinde. Oft sind die teilnehmenden Systeme anonym, oder sind aufgrund von Vertrauensstellungen zu bereits am Gesamtsystem teilnehmenden Systemen teilnahmeberechtigt.

Aufgrund dieser Eigenschaften eignen sich P2P Systeme vor allem dann, wenn die genauen Anforderungen an das Gesamtsystem nicht definiert sind, wenn die Resourcen aller Teilnehmer besser genutzt werden sollen, oder kein zentrales System verfügbar ist. Ein wichtiges Kriterium ist auch das folgende: P2P Systeme eignen sich dann besonders gut, wenn das System für einen Nutzer auch dann noch einen Nutzen erbringt, wenn nicht alle Dienste verfügbar sind (Beispiel Napster: Wenn ein Host, der meinen Lieblingstitel enthält gerade nicht verfügbar ist, probiere ich es später nochmal).

## Anwendungsszenario

Das folgende, fiktive Beispielszenario soll demonstrieren, wie man sich einen Suchdienst auf Basis von klassischen Client/Server Technologien vorstellen kann, und wie dieser sich von einer P2P Implementierung unterscheiden würde. Nehmen wir zum einen *Yahoo!* als Anbieter eines Services, der dem Benutzer anhand Kategorisierung die Suche nach Informationen in diversen Themengruppen vereinfacht, zum anderen eine fiktive Anwendung namens *InfoPeer* auf Basis eines P2P Netzwerks. Der Einfachheit halber soll hier von Authentifizierung oder Namensdiensten abgesehen werden. Diese Einschränkung ist zwar eine starke Vereinfachung gegenüber realen Serviceanforderungen, wir werden jedoch noch darauf zurückkommen, wie diese Mechanismen im Vergleich zu traditionellen Diensten für P2P Netzwerke realisiert werden können.

Um den Service von *Yahoo!* benutzen zu können muss ein Benutzer die entsprechende Webseite im Browser ansteuern. Dies setzt bereits voraus, dass der Service **identifizierbar** sein muss, nämlich über seinen Namen, also *http://www.yahoo.com*, der in diesem Fall eine Internetdomäne wiederspiegelt. Ohne diese Indirektion bleibt der Service für den Benutzer unerreichbar. Dem Benutzer von *InfoPeer* dagegen werden nach dem Start des Clients sofort alle Services zur Verfügung stehen. Der Client selbst übernimmt die Kontaktaufnahme zu den Peers und präsentiert initiale Informationen, von der aus weiter navigiert werden kann.

Eine wichtige Eigenschaft, die wir üblichweise als gegeben annehmen, ist nach diesem ersten Schritt bereits sichtbar geworden: die Verfügbarkeit eines Dienstes. Während *Yahoo!* einen immensen Aufwand betreiben muss, um den Service jederzeit für alle Benutzer erreichbar zu halten, wird *InfoPeer* lediglich die Peers ansprechen, die zum Zeitpunkt der Benutzung ebenfalls aktiv sind. Damit wird deutlich, dass der klassische Dienstleister

unter Rahmenbedingungen operiert, die eine gewisse Organisation erfordern. Der Serviceanbieter ist nur innerhalb seiner Domain sichtbar, und die statische Struktur von Namensdiensten erfordert die Verfügbarkeit der Webseite als Kontakt zum Benutzer. Nur durch die Erfüllung dieser Bedingungen kann *Yahoo!* seinen Kunden einen zuverlässigen Dienst anbieten. *InfoPeer*-Benutzer hingegen sehen eben nur Informationen, die im Augenblick verfügbar sind. Das geht sogar so weit, dass eine bestimmte Information von einem Moment auf den nächsten unerreichbar werden kann, z.B. wenn sich ein Peer aus dem Netz ausloggt. Diese Dynamik beeinflusst natürlich grundlegend die **Qualität** des Dienstes. Um diese Auswirkungen zu minimieren, kann man versuchen, mittels Datenreplikation auf mehreren Peers die Qualität möglichst konstant zu halten.

Lassen wir für einen Moment die erwähnten technischen Faktoren aussen vor: *Yahoo!* beschäftigt ein Vielzahl von Redakteuren, die für die Kategorisierung von Informationen verantwortlich sind. Diese Redakteure bestimmen am Ende die Qualität der vorhandenen Informationen, die für den Benutzer sichtbar sind. *Yahoo!* kann allerdings nur eine begrenzte Anzahl von Redakteuren beschäftigen, und diese Redakteure befinden sich sicherlich nicht alle auf dem selben Qualitätsniveau. Was bedeutet das für den Benutzer von *Yahoo!*? Die Fülle der Information weist eine maximale Wachstumsrate auf, da es der begrenzten Anzahl von Redakteuren nicht möglich ist, beliebig schnell zu recherchieren und Webseiten zu generieren. Das Qualitätsniveau der vorhandenen Informationen wird für den Benutzer relativ hoch sein, da wir einfach unterstellen, dass die Redakteure wissen, was sie tun. Sieht man sich im Vergleich dazu *InfoPeer* an, dann wird jeder Benutzer zum Redakteur, sobald er den Client startet. Die lokal vorhandene Information steht **spontan** im gesamten P2P Netz zur Verfügung und ist von anderen Teilnehmern erreichbar. Was bedeutet das für den Benutzer?

Je mehr Teilnehmer sich im *InfoPeer* Netzwerk tummeln, desto größer sind die Chancen, die gesuchte Information zu finden. Durch eine Vielzahl von Teilnehmern kann es zu einer wahren Explosion an vorhandener Information kommen. Damit steht und fällt die Brauchbarkeit eines P2P Netzwerks mit der Anzahl der Peers, die sich daran beteiligen. *Yahoo!* hingegen wird aus oben genannten Gründen nicht die Möglichkeit haben, entsprechende Datenmengen anzubieten.

In welchem Verhältnis stehen nun Datenmenge und Qualität? Auf der einen Seite steht *Yahoo!* mit seiner komplexen semantischen Analyse, Metabeschreibung und Indizierung von Dokumenten. Auf der anderen Seite bewegt sich eine chaotische Masse an Information, die zur Zeit nur durch Dateinamen beschrieben ist. Der Schluss liegt nahe, dass klassisches Serviceanbieter hier heute noch die Nase vorn haben. Würde man jedoch eine Analyse von Dokumenten auf Seite des Clients verwirklichen, wären die Möglichkeiten für P2P praktisch unbegrenzt.

## Einflüsse auf die Softwareentwicklung

Bis jetzt hatten wir nur von dem Service aus Benutzersicht gesprochen. Was ändert sich jedoch bezogen auf die Entwicklung der Software?

Sieht man sich Componentware und P2P von der Entwicklerseite an, dann fällt zunächst auf, dass sich beide wesentlich in ihrer Komplexität unterscheiden. Erstere legt sehr viel Gewicht auf sogenannte **Quality of Service** (QoS) Parameter. Dazu gehören u.a.

- Verfügbarkeit, die üblicherweise über 99% liegen muss, um finanzielle und Image-Einbussen zu vermeiden,
- Skalierbarkeit, damit im Erfolgsfall auch große Benutzermengen bedient werden können ohne dass Benutzer durch zu hohe Antwortzeiten vergrault werden,
- Wartbarkeit, damit die Applikation ohne Ausfall auf den neuesten Stand gebracht werden kann (Zero Downtime)
- Operabilität, damit ein dediziertes Team die Möglichkeit hat, den Service rund um die Uhr zu überwachen und gegebenenfalls sofort zu reagieren, falls unvorhergesehene Zwischenfälle die Serviceleistung dezimieren.

Um dies zu realisieren sind umfangreiche, meist komplexe Architekturen erforderlich. Heutige Applikationsserver vereinfachen zwar einen Teil dieser Arbeit im Bereich der Softwareentwicklung, es gilt jedoch noch weitere Faktoren zu berücksichtigen. Beispiele dafür sind Netzwerktopologie, Sicherheitsmechanismen wie etwa Firewalls, Spezialistentum der Serviceteams u.a. Das grundsätzliche Problem ändert sich nicht: ohne Verständnis für die Problematik der QoS Parameter sind die Chancen sehr groß, ein Projekt (spätestens nachdem es live geschaltet wurde) an die Wand zu fahren.

Damit nicht genug. Um entsprechendes Wissen aufzubauen, und die Strategien der Produkte am Markt vergleichen zu können, bilden sich Communities, Foren und Newsgroups, und Mailinglists zusätzlich zu den Informationen, die der Hersteller bereitstellt. Prinzipiell eine gute Sache, aber ist man als Softwareentwickler dieser Informationsflut noch gewachsen? Und das alles, weil man ein Produkt ausgewählt hat, mit dem man einen Service realisieren will? Kann der durchschnittliche Entwickler heute überhaupt noch Schritt halten mit den massiven Technologiewechseln, ohne das eigentliche Ziel, nämlich möglichst einfach und schnell Applikationen zu entwickeln, aus den Augen zu verlieren?

Vergleicht man die o.g. Eigenschaften beider Modelle, erscheint dann die Entwicklung von P2P Client Software nicht trivial? Unterstützen wir Quality of Service? Naja, wenn sich der Teilnehmer, von dem wir gerade Informationen beziehen, just in diesem Moment ausloggt, dann können wir wohl nichts machen. Vielleicht den Teil der Information sichern, denn wir bereits haben, als nicht komplett markieren, und ab und an mal nachsehen, ob er entsprechende Teilnehmer wieder eingeloggt ist, so dass wir den Rest später holen

können? Das erhöht die Benutzbarkeit und ist relativ einfach zu implementieren. Verteilte Objekte? Server uptime? Transaktions-Timeouts? Nicht in der Welt von P2P. Wir kommunizieren mit dem, der gerade da ist, und wenn der keine Lust mehr hat... na wenn schon.

Dies ist sicherlich eine extreme Betrachtungsweise des Problems. Auch P2P Software ist keine Trivialität, und auch dort wird es Standards und Technologien geben, in die sich der Entwickler einzuarbeiten hat. Allerdings sind die Modelle, von denen wir hier sprechen, schon grundsätzlich verschieden . Was für Einflüsse auf die Softwareentwicklung gibt es noch?

Einen enorm wichtigen Punkt könnte die **Evolution** von Softwaresystemen darstellen. Betrachten wir noch einmal unserer klassischen Service *Yahoo!* dann stellen wir fest, dass ein geschlossenes Entwicklerteam die Funktionalität des Dienstes bereitstellt. Aehnlich dem Problem der festen Anzahl von Redakteuren, kann auch hier nur eine begrenzte Anzahl Personen an der Entwicklung teilhaben, muss dafür aber aus Millionen von Kundenwünschen die meistgefragten herausfiltern und anbieten. Der Rest wird nicht berücksichtigt. Dagegen könnte *InfoPeer* durch die Offenlegung der wichtigsten Schnittstellen eine wesentlich größere Entwicklergemeinde aktivieren, die sich ihre Features selbst implementiert. Genauso wie *JXTA* als P2P Plattform offen gelegt wurde, könnten applikationsspezifische Details verfügbar gemacht werden, und so zusätzlich die Popularität eines Dienstes steigern. Nach diesem Prinzip wurden bereits andere Projekte erfolgreich "popularisiert". Der bekannteste Vertreter dürfte wohl Napster sein. Durch Offenlegung des P2P Kommunikationsprotokolls *OpenNap* wurde die Entwicklung weiterer Clients im OpenSource Bereich gefördert. Das bedeutete im konkreten Fall von Napster natürlich auch, dass man selbst keinen Linux Client entwickeln musste, sondern dies voll und ganz der OpenSource Gemeinde überlassen und gleichzeitig die Zahl der Benutzer wesentlich steigern konnte.

Kommen wir noch einmal auf den Punkt Entwicklung neuer Features zurück. Wir haben bereits erwähnt, dass ein geschlossenes Team, dass für die Wartung und Weiterentwicklung eines Systems verantwortlich ist, meist wesentlich langsamer vorankommt als eine offene Gemeinde, die von allen Seiten zusätzliches Knowhow einbringen kann. Das könnte abhängig von den beteiligten Leuten zu einer Steigerung der Entwicklungsgeschwindigkeit führen. Betont sei hier das Wort *könnte*, da es natürlich auch Gegenbeispiele gibt, wie etwa Mozilla, den Nachfolger des populären Netscape Browsers. Immerhin besteht durch das offen Modell die Chance, Personen in ein Projekt zu involvieren, die wertvolles Wissen einbringen können, und damit direkten Einfluss auf den Dienst nehmen können.

Führen wir uns also die Eigenschaften beider Modelle, klassischer Service und P2P, nochmal vor Augen, dann haben beide ihre Vor- und Nachteile. Wie können wir diese zwei Welten also derart miteinander verbinden, dass wir dem Benutzer eine nächste

Generation von Services bieten können, die die Möglichkeiten verbessert? Unsere Antwort sind Hybridsysteme, die sich die Vorteile beider Modelle zu Nutze machen.

## Hybride: P2P und Komponenten

Üblicherweise wird in der Praxis nicht die reine Lehre umgesetzt, das gilt für fast alle Technologien. Erfolgreiche Systeme sind meist Hybriden die verschiedene Ansätze sinnvoll miteinander kombinieren. Statt also nun zu untersuchen wo und wie genau P2P oder Komponentensysteme eingesetzt werden können, wollen wir überlegen, wie sich die beiden Systemarten ergänzen können. Wir wollen dabei verschiedene Kombinationsmöglichkeiten aufzeigen. Dabei gilt, dass immer dann wenn zentrale Komponenten im Spiel sind, für diese die typischen Anforderungen an Komponentensysteme gelten: Skalierbarkeit, Performance, Verfügbarkeit, etc.

Einer der Nachteile von reinen P2P Systemen ist die Tatsache, dass nicht immer alle Peers verfügbar sind, und daher bestimmte Daten (oder Dienste) zu bestimmten Zeiten nicht verfügbar sind. Die naheliegendste Möglichkeit dies zu ändern ist sicherlich, Daten zentral zu halten, und die Verarbeitung der Daten P2P-mäßig verteilt durchzuführen. Dies stellt sicher dass wichtige Daten jederzeit für alle verfügbar sind, und ermöglicht trotzdem, die Vorteile von P2P Systemen zu nutzen: Also z.B. einfache und schnelle Evolution von Funktionalität.

Eine weitere Möglichkeit besteht darin, eine zentrale Stelle als "Cache" zu verwenden, für den Fall, dass bestimmte Clients nicht verfügbar sind. Dieser Cache kann die gesamten Daten enthalten, oder alternativ auch nur eine Art Index. Damit ist sichergestellt, dass ein Client bei einer Suche auch einen Verweis auf die Daten bekommt, die gerade wegen nicht verfügbarer Peers nicht abrufbar sind.

Die andere Möglichkeit ist, Daten verteilt zu halten und einige zentrale Dienste darauf anzubieten. Zum Beispiel ist eine Suchmaschine oder der Zugriff auf einen Gesamtindex sicherlich schneller, wenn er auf einer zentralen Datenbank liegt. (Übrigens: Dies ist das Prinzip wie das WWW und die darauf operierenden Suchmaschinen funktionieren).

Eine weitere Variante ist, Daten und Verarbeitung verteilt durchzuführen, aber einige Infrastrukturdienste zentral abzuhandeln. Typische Beispiele hierfür wären Security (unter Verwendung einer zentralen Benutzerdatenbank) oder Accounting, wo für jeden Benutzer ein Konto geführt wird, um ihn für die erbrachten Dienste zu entlohnen, bzw. ihm Gebühren für in Anspruch genommene Dienste in Rechnung zu stellen. Ein zentraler Bestandteil kann auch dazu dienen, Duplikate zu vermeiden oder eine bestimmte Strukturierung der Information zu erzwingen. Das ist zentral deutlich einfacher, als in einem reinen P2P System.

Eine ganz andere Funktion eines zentralisierten Elements in einem P2P System kann sein, Aufgaben zu koordinieren. Bei verteilten Berechnungen muss ja eine Instanz koordinieren

wer wann welche Teilaufgabe übernimmt, bzw. die Teilergebnisse müssen wieder zusammengesetzt werden.

## P2P Komponenten

Unter P2P Komponenten verstehen wir die Anwendung von (einigen) P2P Konzepten auf Komponentensysteme. Anstatt nur einige wenige Server und viele Clients zu verwenden, sollten alle an einem Netz beteiligten Systeme potentiell als Server für Komponenten dienen. Eine zentrale Stelle koordiniert, welche Komponenten auf welchem Server laufen können. Ein Beispiel: Ein großes Enterprise System verwendet in einem Intranet leistungsfähige PCs als Endgeräte für die Benutzer. Nachts arbeitet niemand an diesen Rechnern, man könnte sie für andere Aufgaben verwenden. Man könnte dazu, je nach anfallender Aufgabe, bestimmte Komponenten auf diesen Maschinen installieren und von zentralen Koordinationsprozessen aus verwenden. Dies wäre eine Anwendung von P2P Komponenten.

Was ist nun nötig, um P2P Komponenten zu realisieren? Ausgehen von den Mustern im Kasten soll nun kurz angerissen werden, was zusätzlich notwendig wäre:

Die **Komponenteninstallation** müsste vereinfacht werden. Da regelmäßig andere Komponenten auf bestimmten Maschinen laufen (aufgrund geänderter Anforderungen, oder zwecks Lastverteilung) muss die Installation mittels einer gemeinsamen API oder ähnlichem passieren. Vorher muss möglicherweise automatisch ein entsprechender Application Server auf den betreffenden Maschinen automatisch installiert oder upgedatet werden können. Dazu können allerdings die bereits vorhandenen IT-Management-Systeme (OpenView, etc.) verwendet werden. Um Komponenten die nicht mehr verfügbar sind auch wieder deinstallieren zu können, muss es möglich sein, dass man Clients dazu zwingen kann, ihre Referenzen auf eine Komponente freizugeben. **Leases** sind eine Möglichkeit, so etwas zu erzielen, und ein derartiger Mechanismus müsste in die Komponentenarchitektur integriert werden. Um den Vorteil der potentiell vielen vorhandenen Peers nutzen zu können, muss die Voraussetzung zur Lastverteilung geschaffen werden. Eine zentrale Stelle zur **Lastdiagnose** der einzelnen Peers ist nötig, diese zentrale Stelle wird bei Überlastung eines Peers die darauf laufenden Komponenten auf einer anderen Maschine zusätzlich installieren. Um Clients von diesen vorhandenen Komponenten in Kenntnis zu setzen, ist ein zentraler, ggf. replizierter **Trader** notwendig. Das Installationstool meldet die Komponenten bei dem Trader an oder ab, Clients können diesen verwenden um herausfinden, wo welche Komponente läuft. Um das ganze wirklich sinnvoll nutzen zu können, wäre dann noch ein grafisches **Konfigurationstool** nützlich, wo definiert werden kann, welche Komponente wann auf welcher Maschine läuft.

# Kasten: Grundlegende Muster für Komponentenarchitekturen

Komponentenarchitekturen wie Enterprise JavaBeans, CORBA Components oder COM+ basieren alle auf denselben grundlegenden Prinzipien und Mustern. In [1] sind diese ausführlich besprochen. Dieser Überblick erläutert die Konzepte kurz.

## Prinzipien

Kernprinzip bei Komponentenarchitekturen ist die **Trennung von Belangen:** es werden technische und funktionale Belangen getrennt. Technische Belange sind z.B. Sicherheit, Transaktionen, Nebelhäufigkeit, Ressourcenmanagement - funktionale Anforderungen sind die fachlichen Anforderungen des Systems. Eine Trennung dieser Anliegen ermöglicht, dass die beiden Anliegen unabhängig voneinenander - und von verschiedenen Personen - weiter entwickelt werden können.

Desweiteren sollten Komponentenarchitekturen aus Gründen der Ausfallsicherheit, Lastverteilung und Skalierbarkeit als M**ehrschichtige Systeme** aufgebaut werden. Eine spätere Verteilung der einzelnen Schichten auf verschiedene Maschinen wird damit erleichtert.

Als drittes Prinzip ist die **Variabilität** zu nennen. Um die Wiederverwendung der einzelnen Komponenten (s.u.) zu fördern, sollten Komponenten einen gewissen Grad an Flexibilität aufweisen.

## Grundlegende Muster

Im Mittelpunkt steht die **Komponente**, die einen genau definierten Verantwortungsbereich hat. Die Gesamtfunktionalität wird also in einzelnen Komponenten aufgeteilt. Damit diese miteinander arbeiten können, ohne auf die konkreten Implementationen angewiesen zu sein, wird ein **Komponenteninterface** eingeführt. Dieses definiert die Signatur und (idealerweise) die Semantik der Operationen, die die entsprechende Komponente anbietet. Abhängigkeiten bestehen nur zwischen den Interfaces der einzelnen Komponenten, die Implementierungen können beliebig weiterentwickelt werden.

Komponenten implementieren, wie erläutert, nur die funktionalen Belange der Anwendung. Damit diese aber richtig funktioniert, müssen auch die technischen Belange irgendwie realisiert werden: Dazu dient der **Container**, eine Ablaufumgebung für Komponenten. Er kümmert sich um Transaktionen, Sicherheit, Threading, etc. Der Komponentenprogrammierer braucht sich nicht (sehr) darum kümmern. Ganz im Sinne von Wiederverwendung werden Container überlicherweise als fertiges Produkt gekauft. Um nun mehrschichtige Anwendungen realisieren zu können, müssen die Komponenten im Container über's Netz (von Clients oder anderen Komponenten aus) erreichbar sein. Ein **Komponentenbus** abstrahiert die Kommunikationsinfrastruktur (Netzwerk, Protokoll,

etc.). Er transportiert Operationsaufrufe über's Netz und bedient sich dabei üblicherweise OO-RPC Mechanismen wie CORBA/IIOP, RMI/JRMP oder COM+/RPC.

Nun kann nicht zu jeder Zeit jede Komponente aktiv im Speicher des Containers sein, da dies zu Resourcenproblemen führen würde. Daher muss ein Kompontenmodell verschiedene **Identitäten unterscheiden**: Die logische Identität einer Komponente, und die physikalische Komponenteninstanz. Der Container kann einer physikalischen Instanz beliebig viele logische Instanzen zuweisen, oder nicht gebrauchte Instanzen temporär aus dem Speicher entfernen. Ein **Komponentenproxy** dient als Statthalter für gerade nicht aktive logische Instanzen. Der Client kommuniziert grundsätzlich nur mit dem Proxy, was dem Container erlaubt, nach eigenem Gutdünken die Identitäten zu wechseln oder Komponenten zu reaktivieren. Damit eine Komponente vom Container in ihrem Lebenszyklus gesteuert werden kann, muss sie über **Lebenszyklusoperationen** verfügen, die der Container (oder der Proxy) aufruft, um bspw. einen Wechsel der logischen Identität einer Instanz herbeizuführen. Der Implementierer der Komponente muss sich nicht um das Management des Lebenszyklus kümmern, er muss lediglich die Lebenszyklusoperationen korrekt implementieren, also beispielsweise den Zustand aus einer Datenbank laden.

Clients müssen in der Lage sein, mit bestimmten Komponenten zu kommunizieren. Da sie ja nicht im Container leben, muss ihnen irgendwie Zugang zum Komponentenbus gewährt werden, und üblicherweise werden auch irgendwelche Interface-Dateien benötigt. Diese stelle die sogenannte **Client-Bibliothek** bereit.

Desweiteren müssen Komponenteninstanzen erzeugt, zerstört, oder - bei Komponenten mit logischer Identität - wiedergefunden werden. Diese Vorgänge können beliebig komplex sein und hängen üblicherweise eng mit der Implementation des Containers zusammen: eine Sache, die den Client oder den Komponentenimplementierer nicht interessieren soll. Daher benötigt man ein **Komponenten-Home**, welches die entsprechenden Operationen zur Verfügung stellt, und welches (zum Grossteil) vom Container implementiert wird. Nun muss der Client (oder eine Clientkomponente) allerdings irgendwie in den Besitz eines solchen Homes kommen: Dazu wird üblicherweise ein **Namensdienst,** wie man ihn auch aus CORBA oder RMI kennt, verwendet.

Leider kann das Ideal, dass sich der Komponentenentwickler nur um die funktionalen Belange kümmern muss, nicht ganz erreicht werden, denn zumindest muss der Entwickler sagen, *was* der Container für ihn tun soll: also beispielsweise Angeben, wo welcher Benutzer zugreifen darf, oder wo eine Transaktion notwendig ist. Er muss dies zwar nicht selbst programmieren, aber er muss **Anmerkungen** schreiben, in denen deklarativ vermerkt ist, welches Verhalten er sich vom Container erwartet. Der Container muss dies dann umsetzen - damit die Performance stimmt, wird daher üblicherweise eine **Glue-Code Schicht** generiert. Teil dieser Schicht ist auch der Komponentenproxy.

Dieser Code muss natürlich irgendwann erzeugt werden, allerdings erst nachdem die Konsistenz und Korrektheit der Anmerkungen überprüft wurde. Daher ist ein expliziter **Installationsschritt** notwendig. Hier wird die Komponente dem Container "vorgesetzt", dieser prüft alles und führt, wenn alles in Ordnung ist, alle notwendigen Schritte durch, um die Komponente im Container laufen lassen zu können. Dabei wird sie auch mit dem Komponentenbus verbunden, und ihr Homeinterface wird im Namensdienst registriert.

Wenn die Komponente dann im Container läuft kann sie dies üblicherweise nicht ohne einen gewissen Kontakt zum Container tun. Sie muss zum Beispiel auf Resourcen zugreifen können, oder muss den Container über den Status der aktuellen Transaktion unterrichten können. Dazu wird ihr durch Lebenszyklusoperationen ein **Komponentenkontext** zur Verfügung gestellt. Resourcen sollten aufgrund der Trennung der Identitäten und anderer Kriterien nicht von der Komponente verwalten werden. Vielmehr ist die **Verwaltung von Resourcen**, also z.B. das Anlegen von entspr. Pools Aufgabe des Containers.

Um dem Anspruch an Variabilität gerecht zu werden, muss eine Komponente Zugriff auf **Konfigurationsparameter** haben. Diese können bei der Installation der Komponente mit den passenden Werten versehen werden, und der Komponentenkontext bietet dann zur Laufzeit Zugriff auf diese Parameter.

Zuguterletzt muss noch sichergestellt werden, dass bei der Ausführung einer Operation einer Komponenten Zugriff auf die aktuelle Transaktion oder Sicherheitsinformationen besteht. Beim Aufruf von Operationen durch den Client, oder durch einen andere Komponente, muss also mehr übertragen werden als nur die reine Operation: dazu wird ein **Aufrufkontext** benötigt.

## Weitere Muster

Die oben genannten Muster sind grundlegend für eine Komponentenarchitektur. Sowohl EJB als auch CCM und COM+ wenden diese Muster an. Es gibt nun aber noch eine ganze Reihe weiterer Muster, die angewendet werden können - aber nicht zwingend sind. Dazu zählen verschiedene Arten von Komponenten, ein Packaging-Mechanismus, Interception, Pooling und Passivierung, etc.

Die optionalen Muster, als auch die grundlegenden sind ausführlich beschrieben in [1].

## Referenzen

[1] Völter, Schmid, Wolff: *Server Component Patterns - Architecture and Applications with EJB*, Wiley, geplant für Frühjahr 2002.