Java-Komponenteninfrastrukturen: Grundlagen und Beispiele

Markus Völter, voelter@acm.org, www.voelter.de

Diesers Artikel soll die grundlegenden Prinzipien aufzeigen, die Komponenteninfrastrukturen zugrunde liegen, mit speziellem Fokus auf Java. Dazu werden zunächst einige der Grundlagen erläutert, und dann zwei verschiedene Realisierungen betrachtet: Zum einen Enterprise JavaBeans, und zum anderen ein Prototyp eines Komponentenmodells für Embedded Systeme – einer Domäne, wo kompontenbasierte Ansätze nicht alltäglich sind. Der Artikel führt nicht in EJB ein, Kenntnisse der Funktionsweise werden vorausgesetzt.

Kompontenbasierte Systeme

Komponentenarchitekturen wie Enterprise JavaBeans, CORBA Components oder COM+ basieren alle auf denselben grundlegenden Prinzipien und Mustern. In [VSW02] sind diese ausführlich besprochen. Ziel dieser abstrakten Muster ist es, die grundlegenden Charakteristika von Komponenteninfrastrukturen herauszuarbeiten, um dann die gleichen Konzepte in anderen Domains anwenden zu können.

Hier im Rahmen des Artikels sollen die Konzepte kurz erläutert werden. Es folgt dann die Implementierung der Konzepte in EJB um zu zeigen, dass die Muster wirklich angewandt werden (man könnte das auch mit CORBA Components oder COM+ zeugen. Im dritten Teil des Artikels folgt die Umsetzung der Konzepte im Rahmen eines Komponentenmodells für Embedded-Systeme. Ziel des Artikels ist es, die allgemeine Anwendbarkeit der Komponentenorientierung zu zeigen – auch in einer Domäne (embedded), wo dieser Ansatz nicht alltäglich ist.

Grundlegende Prinzipien

Kernprinzip bei Komponentenarchitekturen ist die **Trennung von Belangen (engl. Separation of Concerns):** es werden technische und funktionale Belangen getrennt und in verschiedenen Software-Modulen gekapselt. Technische Belange sind z.B. Sicherheit, Transaktionen, Nebenläufigkeit, Ressourcenmanagement, Persistenz - funktionale Anforderungen sind die fachlichen Anforderungen des Systems. Eine Trennung dieser Belange ermöglicht, dass die beiden Belange unabhängig voneinander - und durch verschiedenen Personen - weiter entwickelt werden können.

Desweiteren sollten Komponentenarchitekturen aus Gründen der Ausfallsicherheit, Lastverteilung und Skalierbarkeit als **Mehrschichtige Systeme** aufgebaut werden. Eine spätere Verteilung der einzelnen Teile 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 Bestandteile einer Komponenteninfrastruktur

Im Mittelpunkt steht die **Komponente**, die einen genau definierten Verantwortungsbereich hat. Die Gesamtfunktionalität einer Anwendung wird also in einzelne Komponenten aufgeteilt. Damit diese miteinander arbeiten können, ohne von den konkreten Implementationen abhängig zu sein, wird ein **Komponenteninterface** eingeführt. Dieses definiert die Signatur (und idealerweise auch 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 die Gesamtanwendung 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., was auch immer als „technische Belange" in der entsprechenden Domäne identifiziert wird. 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 können nicht zu jeder Zeit alle Instanzen einer Komponente aktiv im Speicher des Containers sein, da dies zu Resourcenproblemen führen würde. Daher arbeitet ein Container üblicherweise mit v**irtuellen Instanzen:** Die logische Identität einer Komponente und die physikalische Komponenteninstanz werden konzeptionell getrennt. Der Container kann einer physikalischen Instanz nacheinander beliebig viele logische Instanzen zuweisen (Pooling), oder nicht gebrauchte Instanzen temporär aus dem Speicher entfernen (Passivierung). Ein **Komponentenproxy** dient als Statthalter für eine oder mehrere 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 der aktuellen Identität aus einer Datenbank laden.

Potentiell remote liegende Clients müssen in der Lage sein, mit 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 eine **Komponenten-Home**, welche 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 diese Anmerkungen dann umsetzen - damit die Performance stimmt, wird dann daraus üblicherweise eine **Glue-Code Schicht** generiert. Teil dieser Schicht ist auch der oben erwähnte Komponentenproxy.

Dieser Glue-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 Home wird im Namensdienst registriert.

Wenn die Komponente dann im Container läuft kann sie dies 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 auch nicht von der Komponente selbst 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 der Komponente dann zur Laufzeit Zugriff auf diese Parameter.

Zuguterletzt muss noch sichergestellt werden, dass bei der Ausführung einer Operation einer Komponenten der Container Zugriff auf alle für die technischen Aspekte nötigen Informationen besitzt, z.B. die aktuelle Transaktion oder Sicherheitsinformationen. 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.

Projektion auf EJB

Die Projektion dieser Konzepte auf EIB ist relativ knapp gehalten, da EJB ja relativ bekannt ist. Im Rahmen von EJB unterscheidet man zwischen vier Arten von **Komponenten**: Stateless Session Beans, Stateful Session Beans, Entity Beans, und Message Driven Beans. Auf einer Erklärung der Unterschiede sei hier aus Platzgründen verzichtet.

Auch in EJB hat eine Komponente ein **Interface**, genauer gesagt, sogar zwei: Das Remote Interface und das Local Interface. Das Remote Interface definiert die Operationen, die auf einer Bean remote aufgerufen werden können, das Local Interface definiert die Operationen, die nur innerhalb des Application Servers, also von anderen Beans, aufgerufen werden können. Die Interfaces definieren nur die Syntax, semantische Informationen können nicht spezifiziert werden.

Um die technischen Belange kümmern sich bei EJB Bean-Typ spezifische **Container**, die in der Regel innerhalb eines J2EE Application Servers laufen. Die Container kümmern sich um Security, Transaktionen, Virtuelle Instanzen, Persistenz und einiges mehr. Zusätzliche Funktionalität, wie z.B. Naming, sind aufgrund des umgebenden J2EE Servers für Beans zugänglich. Der **Komponentenbus** basiert auf RMI, welches wiederum auf verschiedenen darunterliegenden Kommunikationsprotokollen aufbauen kann – üblicherweise IIOP, teils verwenden Applikationsserver aber auch proprietäre Protokolle.

Um Resourcen zu schonen, verwendet auch EJB das Prinzip, dass logische und physikalische Identitäten einer Komponenteninstanz unterschieden werden (**Virtuelle Instanzen**), und zwar folgendermaßen:

- Bei Stateless Session Beans und Message Driven Beans wird der Container einfach einen Pool von Instanzen anlegen, und jeden Request einfach an eine der Instanzen im Pool weiterleiten – da sie ja stateless sind, sind alle gleich.
- Bei Stateful Session Beans geht der Container anders vor: Wann immer der Client einen neue Instanz erzeugt, so wird dadurch auch wirklich physikalisch einen neue angelegt. Allerdings werden diese Instanzen passiviert (also auf Platte

ausgelagert), wenn sie für einen bestimmten Zeitraum nicht mehr benötigt wurden – wenn wieder ein Aufruf für die Instanz ankommt, wird sie wieder aktiviert.

- Bei Entity Beans geht der Container nochmals anders vor. Wie bei Stateless Session Beans legt er einen Pool von Instanzen an. Wenn ein Aufruf für eine bestimmte logische Instanz (also eine bestimmte Entität) beim Server ankommt, ruft er bestimmte Lebenszyklusoperationen auf (z.B. *ejbLoad()),* sodass die Instanz auch wirklich die benötigte Identität bekommt.

Damit all dies funktioniert, wird beim Deployment ein **Komponentenproxy** generiert, der, oft per Reflection, die Operationen auf der Instanz aufruft, nachdem er sich vorher um Transaktionen und Security gekümmert, und die nötigen Lebenszyklusoperationen auf der Instanz aufgerufen hat. Diese **Lebenszyklusoperationen** sind in den *SessionBean* und *EntityBean* Interfaces spezifiziert. Jede Bean-Implementierungsklasse muss dieses Operationen korrekt implementieren. Das **Komponentenhome** wird durch das Home Interface (bzw. durch das Local Home, im Lokalfall) realisiert. Dies wird vom Entwickler formal nie implementiert, das macht der Container automatisch. Allerdings muss der Entwickler die Operationen in der Beanimplementierungsklasse bereitstellen – der Grad, bis zu dem er sie wirklich mit Inhalt füllen muss hängt davon ab, ob Container- oder Bean-Managed Persistent verwendet wird.

Als **Namensdienst** kommt bei EJB das Java Naming and Directory Interface zum Einsatz, mittels dem man CORBANaming, LDAP oder andere Namensdienste ansprechen kann. Üblicherweise bieten Application Server ihre eigene Implementierung.

Auch in EJB benötigt man **Anmerkungen** um den Container bei der Realisierung seiner technischen Aspekte zu steuern. Diese Anmerkungen sind in EJB ein XML File und heißen Deployment Deskriptor. Er enthält Angaben darüber, welcher Attribute persistent sein sollen (bei CMP), für welche Operationen Transaktionen notwendig sind, wer welche Operationen aufrufen darf, sowie Einstellungen betreffend dem Namensdienst und weitere, Herstellerkennzeichen Einstellungen. Die meisten Applikationsserver generieren daraus eine **Glue-Code Schicht**, alternativ wird auch Reflection eingesetzt um zur Laufzeit die technischen Aspekte zu realisieren. Der Installationsschritt heißt bei EJB Deployment, und er wird üblicherweise durch Application-Server spezifische Tools unterstützt.

Beans können ihre Umgebung ansprechen indem sie entweder den SessionContext/*EntityContext* verwenden, ein **Komponentenkontext** den ihnen der Application Server zu Beginn ihres Lebenszyklusses zur Verfügung stellt, oder, sie können direkt Lookups im JNDI Kontext durchführen. Dort können sie zum Beispiel *DataSources* finden, welche die Rolle der **Managed Resources** für Datenbanken spielen. **Konfigurationsparameter** sind auch im Deployment Descriptor angegeben, sie sind zur Laufzeit von der Bean aus einem speziellen Teil des JNDI-Kontextes auslesbar.

Zu guter Letzt sei erwähnt, dass auch EJB einen **Aufrufkontext** unterstütz, der entweder, nativ mit IIOP transportiert wird oder manuell per RMI. Er enthält Transaktions- und Security Tokens.

Warum sind Komponentenmodelle erfolgreich?

Im Enterprise Umfeld sind Komponentenmodelle unbestritten das derzeit erfolgreichste Anwendungsentwicklungsparadigma. Aber warum ist das so? Meines Erachtens gibt es dafür mehrere Gründe:

- Die meisten Applikationen in diesem Umfeld haben eine ähnliche Struktur: Eine Datenbank die Business-Entitäten enthält, irgendwelche Prozesse, die diese verändern, und das ganze üblicherweise transaktional gesichert, mit einem schönen GUI. Es lassen sich da leicht technische und funktionale Aspekte trennen.
- Es liegen Standards vor, und dafür gibt es kommerzielle Container-Implementierungen die man einfach zukaufen kann.
- Als Folge kann man den Programmierern ein relativ einfaches Programmiermodell an die Hand geben (ok, EJB ist nicht einfach. Aber wenn man sich überlegt, wieviel Aufwand es wäre, das alles von Hand zu machen...)
- Die Programmierer können sich entweder auf die fachlichen Dinge konzentrieren und müssen sich nicht um Lastverteilung, Transaktionen, etc. kümmern, oder, wenn sie beim Toolhersteller arbeiten, die fachlichen Dinge ignorieren und nur technische Aspekte implementieren.

Es wäre ja sicher vorteilhaft, wenn man in der Embedded-Welt ähnliche Vorteile haben könnte, um die Entwicklung effizienter zu gestalten. Dabei müssen einige Randbedingungen eingehalten werden.

Komponenten für kleine Geräte

Wenn man die oben genannten Vorteile für Komponenten im Bereich kleiner, embedded oder mobiler Geräte nutzen möchte, so muss man zunächst mal einige Randbedingungen festhalten:

- Man darf sich lange nicht so viel Overhead leisten, wie man das in Enterprise-Systemen kann, die Resourcen (Speicher, Persistenter Speicher, Rechenleistung, opt. Batterieleisting) sind ja begrenzt
- Man muss die spezifischen Eigenschaften der Devices effizient nutzen können
- Man darf keinen „Ballast“ mit sich herumschleppen, wenn ein Device eine bestimmte Funktionalität nicht benötigt

Diese Anforderungen legen einen Ansatz auf der Generative Programmierung basierend nahe (siehe Kasten). Man muss dann analysieren, was denn die technischen Aspekte bei solchen Geräten ausmacht. Die folgende Liste macht einen ersten Vorschlag, mehr ist zu finden unter [SMC].

- Threading und Scheduling
- Interrupt Handling
- Datenspeicherung und Session Handling
- Events und Timer
- Generisches Driver Interface
- Austauschbare, optimierende (Remote-)Transportschicht
- Sicherheit
- Überprüfen von Timing-Randbedingungen
- Zustandsverwaltung
- Überprüfung auf Lauffähigkeit des Systems zur Build-Zeit

Einige dieser Funktionalitäten werden derzeit bereits von (Echtzeit-)Betriebssystemen zur Verfügung gestellt – die Übergänge zu einem Komponentenmodell sind hier fließend. Ein konsequenter, komponenten-basierter Ansatz kommt jedoch nicht zum Einsatz. Dies liegt meines Erachtens vor allem daran, dass die klassischen Implementierungen von Komponentenmodelle viel zu viel Overhead mitbringen, der im embedded-Bereich nicht akzeptabel ist. Die Möglichkeiten der Generativen Programmierung (siehe Kasten), insbesondere der Mechanismen zur Anpassung von Systemen auf Quellcode-Level und zur Compilezeit sind hier sehr interessant.

Im folgenden soll nun beschrieben werden, wie die oben dargestellten Konzepte im Rahmen eines Prototyps für ein Komponentenmodell im Embedded-Bereich realisiert wurden.

Projektion auf den SmallComponents Prototyp

Der aktuelle Prototyp für SmallComponents ist in Java geschrieben, und auch im Rahmen einer J2ME Implementierung z.B. auf einem Palm lauffähig (nähere Infos zum Small Components-Projekt unter [SMC]). Das Prinzip und das Kompoentenmodell lassen sich aber auch mit andere Programmiersprachen implementieren.

Auch im Rahmen des SMC-Prototyps steht die **Komponente** im Mittelpunkt. Sie ist zunächst mal eine gewöhnliche Java-Klasse. Sie muss allerdings drei verschiedene **Interfaces** zur Verfügung stellen. Das Lifecycle Interface (siehe unten) benötigt sie, um in

Container ausgeführt werden zu können. Desweiteren bietet sie ein Service-Interface (SI) an, welches die Operationen definiert, die von Clients auf der Komponente aufgerufen werden können. Als drittes definiert jede Komponente ein sogenanntes Resourceninterfaces. Dieses spezifiziert, welche Resourcen (per Definition auch Komponenten) die Komponente benötigt, um erfolgreich arbeiten zu können. Der Container erkennt spätestens beim Hochfahren, üblicherweise schon bei der Konfiguration, wenn etwas nicht passt. Für jede Resource, die die Komponente benötigt, definiert das Resourceninterface eine Operation der Form set*Resourcenname(Resources-Serviceinterfacetyp* resource). Um z.B. zu spezifizieren, dass die Komponente einen Threadpool benötigt, würde die Operation folgende Signatur haben *setMyThreadPool(ThreadPoolSI resource).* Komponenten sind, zumindest derzeit, grundsätzlich stateless. Dies macht verschiedene Verfahren einfacher und vereinfacht den Container.

Der **Container** bei Small Components wird komplett generiert. Er enthält (durch Generierung fest verdrahtet) den Code um die Komponenten zu instanziieren, sie hochzufahren, und um deren Resourcenanforderungen zu entsprechen. Daher muss weder dynamisches Class-Loading noch sonst ein Reflection-Mechanismus verwendet werden – das ist wichtig, weil diese Mechanismen auf J2ME Devices (und in anderen Sprachen, wie C) üblicherweise nicht vorhanden sind. Ausserdem können viele Dinge schon zu „Generierungszeit" gecheckt werden, unter anderem auch die Resourcenanforderungen. Weitere technische Aspekte, wie asynchrone Aufrufe, das „Zwischenschalten" von Interceptoren (für Security, Logging, Locking, etc.) oder Remote-Zugriff werden über einen automatisch generierten **Proxy** abgehandelt. Dieser Proxy wird allerdings nur generiert und verwendet, wenn er wirklich benötigt wird. Andernfalls werden Aufrufe von einer Komponente zur anderen als direkter Methodenaufruf, ohne jeden Overhead, ausgeführt: wiederum ein wichtiges Merkmal für Embedded Systeme.

Der **Komponentenbus**, also der Remote-Zugriff ist auch komplett generiert. Bei der Konfiguration des Systems muss angegeben werden, welche Komponenten remote zugänglich sein sollen. Es wird dann automatisch (im Rahmen des oben erwähnten Proxy) Code generiert, der die Methoden auf der Komponente im Zielcontainer aufruft, und es wird eine Proxy-Komponente generiert, die im Client-Container verwendet werden kann, um die Aufrufe an den Zielcontainer weiterzuleiten. Auch der Aufruf der Komponente im Zielcontainer passiert ohne Reflection – der Code ist ja komplett generiert. Für den eigentlichen Remote-Transport der Aufrufinformationen können prinzipiell beliebige Protokolle verwendet werden, derzeit existiert eine Implementierung mittels RMI sowie Sockets. Die Serialisierung der Daten kann entweder mittels Java Serialisierung oder auch mit manuellen (auch generierten) Mechanismen passieren, da Java's Serialisierung nicht auf allen Devices verfügbar ist. Operationen die keine Rückgabewerte besitzen können per Konfiguration auch asynchron ausgeführt werden.

Die explizite Verwendung von **Virtuellen Instanzen** ist bis jetzt nicht implementiert, da dieser Aspekt bei reinen Stateless-Komponenten nicht so entscheidend ist. Man kann

allerdings nicht nur einzelne Instanzen von Komponenten definieren, sondern auch einen Pool anlegen, aus dem dann für jeden parallelen Aufruf einer Operation (also pro Thread) eine eigene Instanz verwendet wird.

Der Lebenszyklus einer Komponente, und damit auch ihre **Lebenszyklusoperationen**, weisen eine Besonderheit auf: Es wird explizit zwischen drei Phasen unterschieden: der Konfigurationsphase, der Startphase und der Operationsphase. Wenn bei der Konfiguration des Systems Fehler auftreten (z.B. nicht vorhandene Resourcen, oder fehlende Konfig-Parameters) so wird der Container beendet. Jede Komponente muss mittels der Operation *configComplete()* melden, dass sie komplett und korrekt konfiguriert wurde. Dann folgt die Startphase: Hier kann sich die Komponente initialisieren, z.B. Serversockets anlegen, Treiber initialisieren, etc. Erst danach folgt die eigentliche operationelle Phase, wo die Komponenten verwendet (also Methoden aufgerufen) werden können. Für alle diese Phasen in ihrem Leben bietet das Lifecycle-Interface die nötigen Operationen.

Client-Bibliotheken im eigentlichen Sinne sind nicht nötig, da auch der Client einfach als Container implementiert wird, eben mit möglicherweise einer einzigen Client-spezifischen Komponente. Überhaupt erstreckt sich bei SmallComponents die Konfiguration nicht auf einen Container, sondern immer auf ein gesamtes System aus potentiell vielen Containern die miteinander arbeiten. Einen **Namensdienst** benötigt man hier nicht, weil zur Konfigurationszeit alle IDs von Komponenten und Containern bekannt sind und diese im generierten Code fest verdrahtet werden können. Natürlich könnte, um dem System einen höheren Grad an Flexibilität zu geben, auch ein Namensdienst eingeführt werden, jedoch ist diese Flexibilität in den meisten Embedded Systemen eher kontraproduktiv.

Anmerkungen werden hier nicht pro Komponente geschrieben, sondern für das gesamte System. Sie werden in Form einer XML Datei angegeben. Daraus wird dann eben der Code für den Container und die ggfs. zu generierenden Komponenten generiert. Eine **Glue-Code *Schicht*** im eigentlichen Sinne existiert nicht – der ganze Container ist ja sozusagen Glue-Code, da er komplett generiert ist. Der Installationsschritt entspricht der Generierung, zumal hier ja auch Konsistenzchecks durchgeführt werden. Daraus ergibt sich dann der Abbildung 1 gezeigte generelle Ansatz:

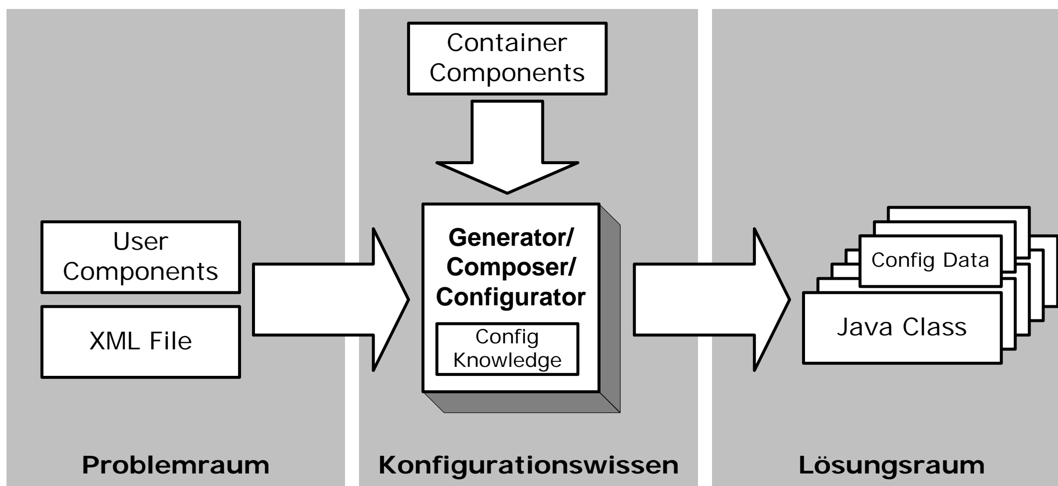

Abbildung 1: SmallComponents: Prinzipielle Funktionsweise

Das System unterscheidet weiterhin zwischen zwei Arten von Komponenten: regulären User-Komponenten die die eigentliche Anwendungsfunktionalität erbringen, und Container-Komponenten, die praktisch „modularisierte Container-Funktionalität" darstellen. User-Komponenten besitzen keinen Zugriff auf den Container oder ihre Umgebung, sie spezifizieren ja alle benötigten Resourcen im Resourcen-Interface. Daher haben sie auch keinen **Komponentenkontext**. Container-Komponenten hingegen haben über einen Kontext die Möglichkeit, auf beliebige andere Komponenten zur Laufzeit zuzugreifen, oder auch den Container zu steuern, zum Beispiel herunterzufahren.

Resourcen gibt es in diesem System im engsten Sinne nicht. Datenbankverbindungen werden z.B. als entsprechende (Container-)Komponenten abgebildet. Da diese Komponenten gepoolt werden können, ergibt sich daraus automatisch auch ein Pooling von Resourcen.

Konfigurationsparameter die die Komponente während ihrer Initialisierung benötigt werden direkt in dem Kofigurations-XML File angegeben und der Komponente während der Initialisierung über ein *java.util.Properties* zugänglich gemacht. Der **Aufrufkontext** wird bei Remote-Aufrufen in das über's Netz gesendete Datenpaket mit eingebaut.

Der Container bietet über das erläuterte Hinaus noch weitere Dienste:

- Signale: Eine Komponente kann sich für bestimmte Signaltypen registrieren (zur Konfigurationszeit). Es wird dann Code generiert, der zur Laufzeit erzeugte Signale an diese Komponenten weiterleitet, und zwar synchron oder Asynchron.
- Timer: Basierend auf den Signalen kann ein Timer verwendet werden, der in konfigurierbaren Abständen bestimmte Signale erzeugt. Der Signal-Service transportiert diese dann zu ihren Empfängern.

- Session Management: Nachdem derzeit alle Komponenten zustandslos (stateless) sind, muss Client-abhängiger Zustand in Sessions verwaltet werden. Dazu bietet der Container entsprechende Dienste an: den Transport von Session-IDs mit Remote-Aufrufen, die Verwaltung von Sessions, und, optional, deren Passivierung.

Das Featurediagramm [EC00] in Abbildung 2 zeigt den derzeitigen Konfigurationsraum des SmallComponents Containers.

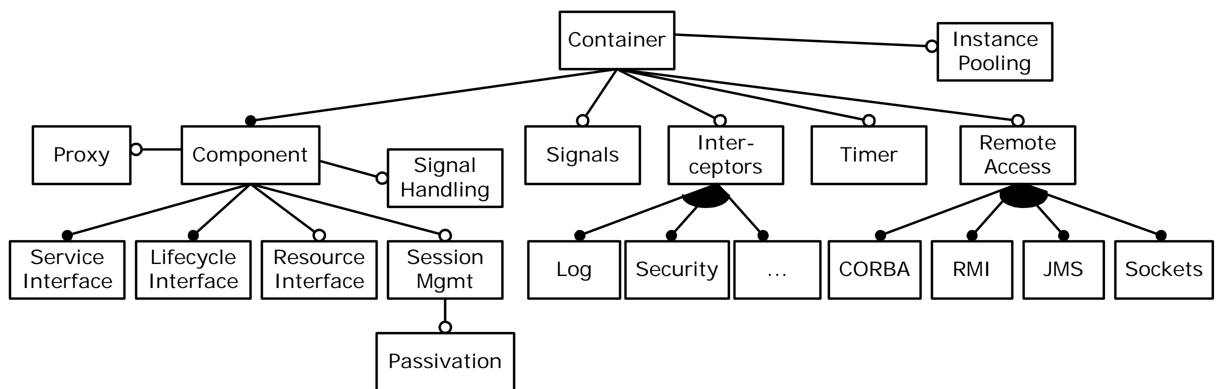

Abbildung 2: Feature-Diagramm des Small Components Container. Erklärung der Notation in [EC00]

Zusammenfassung

Der vorliegende Artikel führt zunächst in die grundlegenden Prinzipien von Komponentenarchitekturen ein und belegt dies am Beispiel von EJB. Danach wird aufgezeigt, wie mit Hilfe dieser Prinzipien eine Komponentenarchitektur für einen Anwendungsbereich implementiert wurde, in dem dieser Ansatz bisher wegen (angeblich) zu hohen Overheads nicht verwendet wurde. Ich hoffe, damit die Möglichkeiten von Komponentenarchitekturen aufgezeigt zu haben, und Sie von der Mächtigkeit abstrakter Betrachtungen (in Form vom Patterns, siehe [VSW02] überzeugt zu haben. Übrigens ist SmallComponents eine Open-Source-Projekt, und Sie sind erzlich eingeladen, sich daran zu beteiligen (siehe [SMC]).

Referenzen

[AOP] *Aspect-Oriented Software Development*, http://aosd.net

[VSW02] Völter, Schmid, Wolff: *Server Component Patterns – Component Infrastructures illustrated with EJB*, Wiley, geplant für Sommer 2002.

[EC00] Eisenecker, Czarnecki: *Generative Programming*, Addison-Wesley 2000

[SMC] *Small Components Project*, www.voelter.de/smallComponents

Kasten: Generative Programmierung

Zunächst die „offizielle" Definition von GP aus [EC00]:

> *Die generative Programmierung (GP) modelliert Softwaresystemfamilien so, daß ausgehend von einer Anforderungsspezifikation mittels Konfigurationswissen aus elementaren, wiederverwendbaren Implementierungskomponenten ein hochangepaßtes und optimiertes Zwischen- oder Endprodukt nach Bedarf automatisch erzeugt werden kann.*

Dazu wird zunächst mit Mitteln des Domain Engineering [EC00] versucht, die Domäne einzugrenzen und zu analysieren, für welche Domäne die Systemfamilie erstellt werden soll. Dabei entsteht üblicherweise ein Feature-Modell, oft als Feature-Diagramm dargestellt. Es zeigt den Raum der möglichen Kombinationen von Features für Produkte der Familie. Es muss nun analysiert werden, welches Feature zu welcher Zeit (also Codierungszeit, Compilezeit, Linkzeit, Startzeit oder Laufzeit) gebunden werden soll. Die generative Programmierung versucht dabei insbesondere, für Dinge die zur Codier- und Compilezeit festgelegt werden mächtige Verfahren anzubieten. Dazu zählen Codegeneratoren, Konfigurationsrepositories, (Template-)Metaprogrammierung und XML/XSLT basierte Verfahren. Auch Aspektorientierung [AOP], ein Verfahren zur Kapselung von Dingen, die „quer" zur primären Modul- oder Klassenstruktur eines Systems gehören, fällt in diesen Bereich.