Mobil Orientiert

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

Komponentenbasierte Systeme sind in Enterprise- und eBusiness-Systemen zwischenzeitlich zur de-facto Architektur avanciert. Für diesen Erfolg gibt es verschiedene Gründe die in der prinzipiellen Funktionsweise von komponentenbasierten Infrastrukturen liegen. Als wichtigste Eigenschaft kann man da die "Separation of Concerns" angeben, auf deutsch ungefähr die "Trennung von Belangen". Konkret bedeutet dies, dass bestimmte technische Aspekte die in allen Systemen einer bestimmten Domäne in der einen oder anderen Art vorkommen, in ein separates Stück Software, den sogenannten Container, ausgelagert werden. Anstatt also diese technischen Aspekte und die Businesslogik in einem Stück Code zu vermischen, werden die technischen Aspekte vom Container standardisiert behandelt, und der Programmierer spezifiziert nur noch *was* er gerne vom Container hätte, er muss aber nicht es aber nicht ausprogrammieren. Da man sich innerhalb einer Domäne üblicherweise auf eine sinnvolle Trennung zwischen technischen und funktionalen Aspekten einigen kann, kann man für Container einen Standard festschreiben. Dies erlaubt die Wiederverwendung und letztendlich das zukaufen des Containers, und damit eine Wiederverwendung der technischen Aspekte einer Domäne.

Am Beispiel von Enterprise JavaBeans kann man sehr schön sehen, daß dieser Ansatz auch funktioniert. Die technischen Aspekte sind hier Dinge wie Transaktionen, Security, Failover, Lastverteilung, oder Naming.

Zusätzlich zur Auftrennung in technische und funktionale Aspekteversucht man durch Verteilung der funktionalen Teile in Komponenten die Wiederverwendung zu erhöhen. Eine Komponente hat daher eine definierte, abgeschlossene Funktionalität, die sie durch ein oder mehrere Interfaces anderen Komponenten zur Verfügung stellt.

Eine Anwendung ist dann also eine Ansammlung von Komponenten die sich gegenseitig über ihr Interface benutzen und in einem Container laufen, der sich um die technischen Aspekte kümmert (für Details zum Thema Komponenteninfrastrukturen siehe [VSW02]).

Komponeten in der Embedded Welt

Nun möchte man von diesen offensichtlichen Vorteilen auch in anderen Domänen als der Enterprise-Welt profitieren. Eine solche Domäne ist die Welt der kleineren, eigebetteten, oder mobilen Geräte: also PDAs, Handies, Embedded Systems generell. Es ist dabei selbstverständlich, dass man da nicht ganz genauso vorgehen kann wie bei Enterprise Systemen, wo man sich den Overhead durch Indirektion und generische Frameworks mehr oder weniger problemlos leisten kann. Dies ist in der Welt der "kleinen Geräte" natürlich nicht so. Der Container, welcher die Komponenten beheimatet muss dort ein recht kleines und optimiertes System darstellen.

Generative Programmierung als Lösung

Die Generative Programmierung (GP, siehe [GP]) kann hier Hilfe bieten und das Dilemma zwischen Flexibilität und Effizienz lösen. Die Ein-Satz-Definition der GP lautet:

> *"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".*

Dies passt ziemlich gut auf das oben geschilderte Problem. Das Ziel ist also, mit Mitteln der generativen Programmierung eine Systemfamilie von Containern zu realisieren. Wichtig ist es hier zu verstehen, dass es nicht darum geht, mit Hilfe der GP und Komponenten eine Familie von Anwendungssystemen zu entwickeln die alle in demselben Container laufen - vielmehr ist das Ziel, eine Familie von angepassten, hochperformanten Containern bereit zu stellen.

Dies ist ein völlig anderer Ansatz als der, der bei EJB (bzw. J2EE) Applikationsservern gewählt wird, dort ist der Applikationsserver immer "aus einem Guss". Die Container-Systemfamilie im Embedded-Fall wird nötig, denn

- Embedded/Kleingeräte sind um einiges inhomogener. Dies betrifft sowohl ihre Aussstattung mit Speicher, Rechenpower und Batteriekapazität, als auch Netwerkinterfaces, die Verfügbarkeit der Netzwerkverbindung usw.
- Die Resourcen sind generell beschränkter. Man kann es sich eben hier nicht leisten, Code und Rechenzeit für Dinge mitzuschleppen, die das aktuelle Gerät nicht bietet oder die man in der aktuellen Anwendung nicht benötigt.

Technische Aspekte in embedded Systemen

Wenn man sich nun also über eine Komponenteninfrastruktur für embedded Systeme Gedanken macht, dann muss man zunächst festlegen, welches denn die technischen Aspekte sind, um die sich ein Container hier kümmern soll. Sicherlich sind dies nicht dieselben, wie im Enterprise-Umfeld. Einige Beispiele für technische Aspekte wären:

Konfiguration und Abhängigkeiten: Ein embedded System muss zuverlässig laufen. Das bedeutet, es muss beim Hochfahren des Systems klar sein, dass das System eine gültige, lauffähige Konfiguration hat. Hat es diese nicht, darf es nicht hochfahren. Dazu muss eine Komponente genau spezifizieren, welche Resourcen sie benötigt.

- Threading: Multithreading sowie asynchrone Nachrichtenübermittlung sollte vom Container erledigt werden können.
- Events: Eine Komponente sollte in der Lage sein, Events zu erzeugen und auch auf im System auftretende Events zu reagieren.

- Timer: Der Container sollte in konfigurierbaren Intervallen oder zu bestimmten Zeiten Events an Komponenten schicken können.
- Remoting: Komponenten sollten mit entfernten Komponenten kommunizieren können, und zwar mittels konfigurierbarer Kommunikationsprotokolle. Mindestens genauso wichtig ist es aber, dass im Falle von lokalen Aufrufen wirklich kein Overhead durch "das im Wege stehen" von Remoting-Infrastrukturen entstehen.
- Interface-Semantik überprüfen: Ein Teil der Semantik von Interfaces läßt sich als Zustandsmaschine mit Zeitangaben angeben. Der Container kann dann überprüfen, wenn solche Timing- oder Zustands-Constraints nicht eingehalten werden und entsprechend aussagekräftige Fehlermeldungen geben.
- Interrupt Handling: Low-Level Tätigkeiten wie Interrupthandling können für die Komponenten leicht verwendbar gemacht werden.

Umsetzung mittels GP

Wie oben bereits erwähnt, stellen diese Anforderungen eine Produktfamilie bezüglich des Containers dar. Im Rahmen eines derzeit in Entwicklung befindlichen Prototypen wurde daher ein auf GP basierender Ansatz gewählt. Abbildung 1 zeigt das entsprechende Feature-Diagramm.

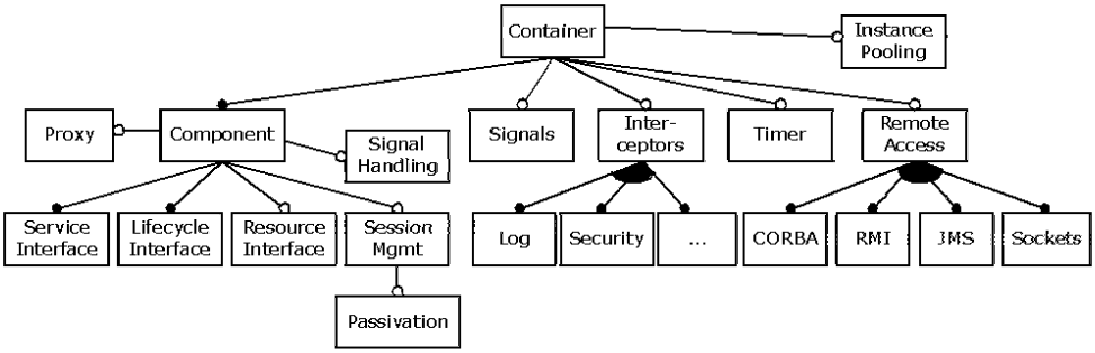

Abbildung 1: Feature Diagramm des Prototypen

Das generative Domänenmodell (in Abb. 2) besagt nun, das man die im Feature-Diagramm dargestellten abstrakten Eigenschaften des Systems mittels Konfigurationswissen auf die Bausteine eines Lösungsraumes abbildet.

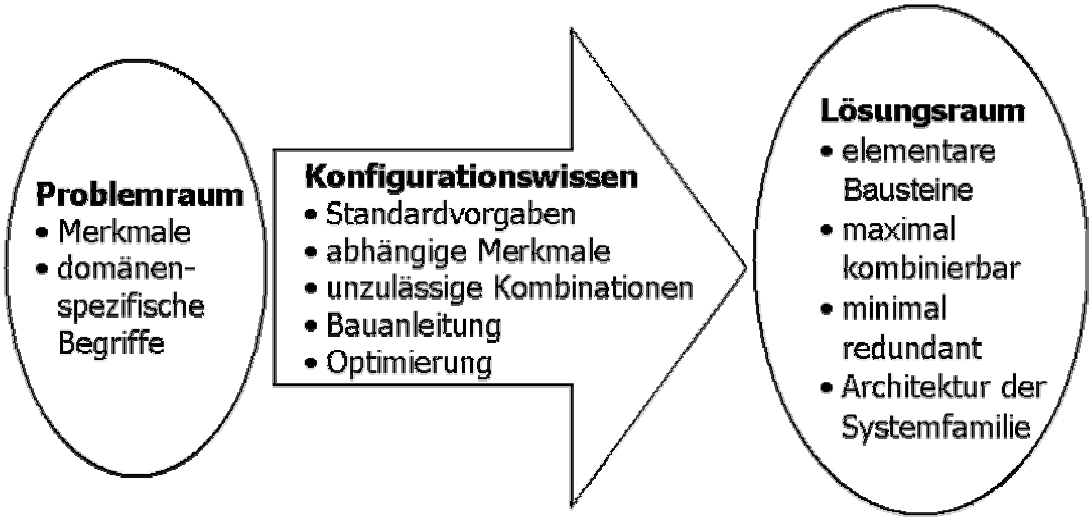

Abbildung 2: Das generative Domänenmodell (nach [GP])

Diese Bausteine sollten sich möglichst einfach kombinieren lassen um die Produkte der Familie zusammenzubauen. Ein Teil der Bausteine stellt die Architektur der gesamten Systemfamilie dar - sie werden also in allen Produkten der Familie verwendet. Die Abbildung der Features des Problemraumes auf die Bausteine des Lösungsraumes sollte im Idealfall vollautomatisch erfolgen, das Konfigurationswissen stellt dazu eine Art Bauplan und einen Produktionsprozess zur Verfügung. Dabei wird zunächst abgeprüft, ob eine bestimmte erwünschte Konfiguration überhaupt baubar ist (die sog. Baubarkeitsprüfung). Danach werden eventuelle Standardvorgaben automatisch gesetzt, und im letzten Schritt vor der konkreten Montage der Bausteine werden eventuell mögliche Optimierungen durchgeführt - wir wollen ja ein "hochangepasstes und optimiertes Produkt" erzeugen. Abbildung 3 zeigt nun, wie das dieser Ansatz beim Prototypen realisiert ist.

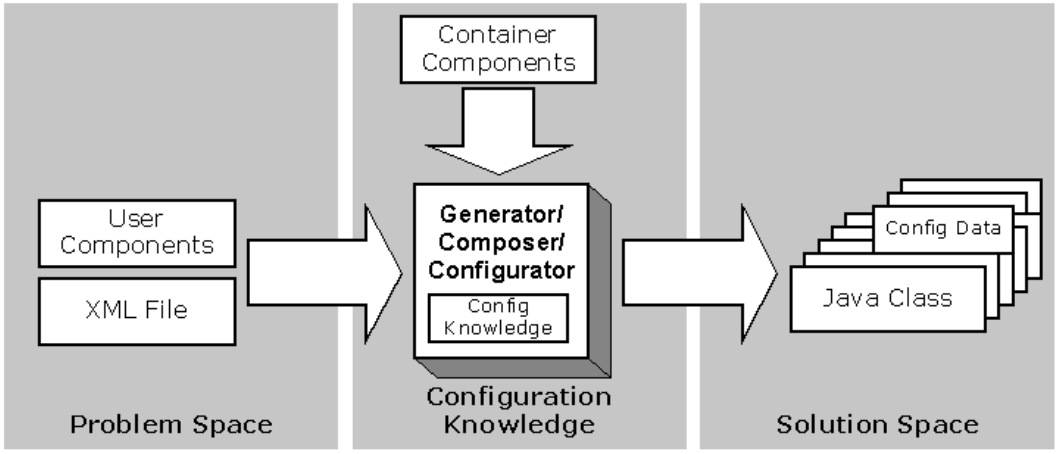

Abbildung 3: Die Generator-Infrastruktur des Prototypen

Realisierung des Prototypen

Der Prototyp ist in Java implementiert. Wir verwenden allerdings keine Java-spezifischen Features, er könnte auch mit C implementiert werden. Generell setzen wir bei dem Prototyp sehr stark auf Codegenerierung - und das aus den folgenden Gründen:

- Generierter Code ist üblicherweise effizienter als generischer Code
- Bestimmte Features sind mit der Sprache Java ohne Generierung nicht realisierbar, vor allem Aufgrund des Fehlens von Generizität (Templates)
- Bestimmte Features (Reflection, zum Teil dynamisches Klassenladen) sind auf kleinen Java Plattformen (J2ME, Personal Java, kJava) nicht verfügbar.

Wenn der Entwickler eine auf dem Container basierende Applikation entwickeln will, so beginnt er zunächst mit der Entwicklung der Komponenten. Solche Komponenten sind manuell programmierte, nicht generierte Java-Klassen, die drei verschiedene Interfaces besitzen:

- Zunächst beschreibt das Service Interface die Operationen, die eine Komponente ihren Clients anbietet (derzeit noch ohne Semantik).
- Das Lifecycle Interface bietet Operationen wie init(), start() oder stop() mit denen der Container die Komponenteninstanz in ihrem Lebenszyklus kontrollieren kann.
- Das Resource Interface beschreibt alle Resourcen (per Definition, andere Komponenten) die eine Komponente benötigt, um vernünftig arbeiten zu können.

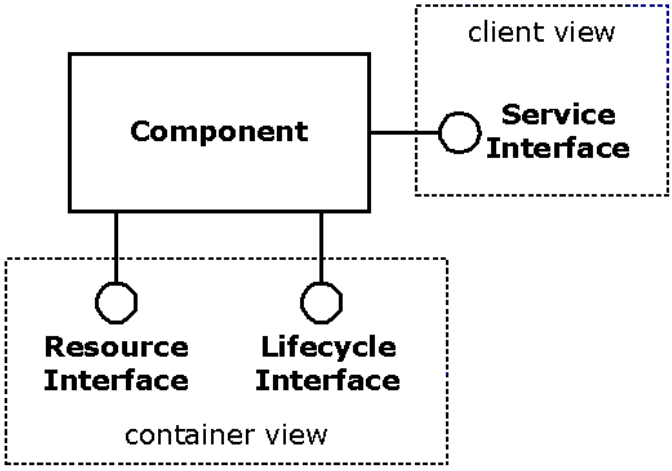

Abbildung 4: Die Schnittstellen einer Komponente

Der Container selbst wird, wie aus Abbildung 3 ersichtlich, komplett generiert. Dabei wird in einem XML-Konfigurations-File angegeben, welche Komponenten im Container laufen sollen. Diese geben, mit Hilfe ihres Resource Interface, an, welche Anforderungen sie an den Container stellen. Der Container-Generator kann nun schon bei der Generierung erkennen, ob das System lauffähig sein wird. Wenn eine Komponente Resourcen benötigt die im entsprechenden System nicht verfügbar sein werden, wird die Generierung mit einem Fehler abgebrochen. Anders gesagt: Wenn der Container erfolgreich generiert werden kann, so ist die Wahrscheinlichkeit dass er dann auch läuft, sehr hoch.

Komponenten als Resourcen

Bestimmte Funktionalitäten bietet der Container natürlich schon von Haus aus. Diese sind allerdings nicht fest im Container eingebaut, sondern stehen als sogenannte Container-Komponenten zur Verfügung. (Dies ist der Grund für die Aussage weiter oben, dass alle Resourcen auch Komponenten sind). Diese Container-Komponenten können vom Entwickler als Resourcen angegeben werden, und das Konfigurationswissen im Generator sorgt dafür, dass diese Container-Komponenten dann auch wirklich im Container vorhanden sind. Ein Beispiel: Wenn eine Komponente ihre Operationen asynchron ausführen soll (was derzeit bedeutet, dass es eine void-Operation sein muß und sie keine Exceptions werfen darf) dann muss dies nur bei der Komponente im Konfgurationsfile angegeben werden. Der Containergenerator erzeugt dann automatisch einen Proxy für die Komponente, der bei Aufruf einer Operation zunächst einen Thread startet, bevor er die Ausführung der Operation an die eigentliche Komponenteninstanz delegiert. Da die Erzeugung von Thread potentiell teuer ist, wird dazu ein Threadpool verwendet. Dieser ist eine Containerkomponente. Der Proxy spezifiziert also in seinem Resource Interface, dass er einen Threadpool benötigt. Dieser wird dann vom Containergenerator automatisch mit

eingebunden (mit Default-Parametrierung; wenn diese nicht passt, muss der Entwickler manuell einen angepaßten Threadpool im Konfigurationsfile definieren).

Status des Prototypen und Zusammenfassung

Derzeit befindet sich der Prototyp noch in der Entwicklung - das Wesentliche funktioniert schon, aber es sind noch nicht alle wünschenswerten Features vorhanden. Er läuft derzeit komplett in J2SE, eine Portierung auf J2ME ist in Arbeit. Der Prototyp wird dann mit Hilfe von esmertec's jBed JVM [JBED] auf einem Palm laufen, später auch auf anderen Geräten.

Resourcen

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

[JBED] JBed JVM, www.esmertec.com

[VSW02] Voelter, Schmid, Wolff: Server Component Patterns, Component Infrastructures illustrated with EJB