Frameworks und Komponenten: Widerspruch oder Ergänzung

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

„Diese Technologie ist die Zukunft der Softwareentwicklung". Ein Satz den man dieser Tage regelmässig nicht nur bei den Marketingabteilungen der Toolhersteller hört, sondern auch auf konzeptionellem Niveau. Ein Kandidat ist zum Beispiel komponentenbasierte Softwareentwicklung. Ein anderer sind Frameworks. Auf den ersten Blick scheinen die beiden Bereiche nichts viel gemeinsam zu haben, vielleicht widersprechen sie sich sogar scheinbar. Bei näherer Betrachtung lässt sich diese These aber nicht halten. Der vorliegende Artikel soll die Gemeinsamkeiten und Ergänzungsmöglichkeiten von Komponenten und Frameworks näher beleuchten.

Historie der komponentenbasierten Softwareentwicklung

Anfang der 80er Jahre versprach die Objektorientierung die Lösung der sogenannten Softwarekrise, die sich durch stark steigende Kosten für die Entwicklung und vor allem die Wartung von Softwaresystemen abzeichnete. Der Lösungsvorschlag der Objektorientierung basierte auf Wiederverwendung von Code. Einer der Grundgedanken der Objektorientierung ist, dass Softwareentwickler Klassen entwickeln, die für genau eine Aufgabe verantwortlich sind, und diese eine Aufgabe sehr gut erledigen können. Die Erledigung anderer Aufgaben delegieren sie an Klassen die auf diese Aufgabe spezialisiert sind. Zu diesem Zweck besitzt jede Klasse eine genau definierte, öffentliche Schnittstelle. Die Implementierungsdetails sollen vor dem Benutzer einer Klasse verborgen bleiben. Wiederverwendung soll nun auf verschiedene Weise eintreten:

- Eine Klasse kann in verschiedenen Anwendungen verwendet werden, da die Verantwortlichkeiten so feingranular verteilt sind, dass dies problemlos möglich ist. Anwendungen sollen also (bei Vorhandensein einer entsprechenden Menge an Basisklassen) aus vorhandenen Klassen zusammengesetzt, statt jedesmal neu implementiert, werden.
- Wenn die Funktionalität einer Klasse geändert werden soll, so muss nicht der Code der entsprechenden Klasse geändert werden. Vielmehr kann durch Vererbung eine neue Klasse erzeugt werden, die alle Features der Basisklasse unterstützt, diese jedoch ändern oder durch neue Features erweitern kann.
- Clients können ohne Codeanpassung sowohl mit der Elternklasse als auch mit abgeleiteten Klassen zurechtkommen, da Unterklassen mindestens die Schnittstelle und Funktionalität der Elternklasse bieten müssen. Man nennt diese Eigenschaft Polymorphismus.

Diese Wiederverwendung soll insbesondere nicht nur für technische Klassen wie z.B. GUI-Widgets eintreten, sondern auch für Fachklassen, wie beispielsweise einer Klasse *Auto* innerhalb einer PKW-Fabrik. Dieses Ansinnen hat sich jedoch nicht so erfüllt, wie es wünschenswert wäre, gerade im Bereich der Fachklassen. Dies hängt mit vielerlei Gründen zusammen, einige folgen.

Um eine Klasse wiederverwenden zu können, muss sie allen potentiellen Anforderungen aller „Verwender" gerecht werden. Dies erfordert eine aufwendige Analyse. Desweiteren stellt eine Klasse ja immer eine Abstraktion der Wirklichkeit dar, und jeder Entwickler abstrahiert die Wirklichkeit anders – abhängig von der Anwendung, in der die Klasse verwendet werden soll. Bei Wiederverwendung durch Vererbung muss gegebenenfalls auf Implementierungsdetails zurückgegriffen werden, dadurch wird das Prinzip der Kapselung verletzt. Das Haupthindernis liegt aber darin, dass eine Klasse im üblichen objektorientierten Sinne mehrere Aspekte in sich vereint:

- Zunächst enthält eine Fachklasse natürlich die Business-Logik, ihre eigentliche Existenzberechtigung.
- Zusätzlich wird oftmals Persistenzcode in die Klasse integriert, d.h. die Klasse weiß (zumindest teilweise) wie sie in einem persistenten Behälter gespeichert werden kann, und wie sie mit Transaktionen umzugehen hat.
- Auch Sicherheitsaspekte, d.h. das Überprüfen von Zugriffsberechtigungen sind oft Teil einer Klassendefinition.
- Oftmals sind weitere Aspekte wie Thread-Sicherheit, Remote-Zugriff oder Log-Ausgaben vorhanden.

Das dahinter lauernde Problem ist eigentlich fast offensichtlich. Selbst wenn die Analyse der fachlichen Anforderungen für eine Klasse sorgfältig durchgeführt wurde und die Klasse somit allen fachlichen Anforderungen aller potentiellen Nutzer gerecht wird: Die Wahrscheinlichkeit dass die oben geschilderten zusätzlichen Aspekte alle identisch sind ist sehr gering, weil dann doch oft ein anderes Persistenzmedium oder ein anderes Zufgriffskonzept verwendet wird.

Ein vielversprechender Ansatz, um dieses Problem in den Griff zu bekommen ist die aspektorientierte Programmierung [AOP]. Dort werden die verschiedenen Aspekte einer Klasse oder eines Systems in separaten Entitäten notiert, den Aspekten nämlich. Durch entsprechende Tools werden die verschiedenen Aspekte dann „zusammengewoben", wobei einzelne Aspekte flexibel an- oder abgeschaltet bzw. ausgetauscht werden können.

Wie oben schon geschildert, sind die wichtigsten Aspekte bei Businesssoftware Persistenz und Transaktionen, Security, sowie entfernter und konkurrierender Zugriff. Damit eine Fachklasse wirklich wiederverwendet werden kann, sollte sie von diesen Aspekten unabhängig sein. D.h. eine Klasse sollte nur die eigentliche Businesslogik implementieren,

und zwar so flexibel, dass sie, wie oben erläutert, allen potentiellen Anwendungen gerecht wird. Alle anderen Aspekte sollten der Fachklasse von aussen „übergestülpt" werden können, abhängig von den technischen Anforderungen des Einsatzzwecks.

Genau diesen Ansatz verfolgen Komponentenarchitekturen wie Enterprise JavaBeans oder COM+. Der Komponentenentwickler erstellt nur die Klassen mit der Fachlogik und definiert deren öffentliches Interface, der Rest geschieht (mehr oder weniger) automatisch. Damit dieser Automatismus greift, existieren Komponenten immer innerhalb einer dafür vorgesehenen Ablaufumgebung – dem Container.

Historie von Frameworks

Die Wiederverwendung von Fachklassen (und damit der typischen Abstraktionen einer Institution oder einer Domäne) ist die eine Sache. Üblicherweise ist eine Klasse für sich alleine aber recht nutzlos, weil sie eben nur genau eine Aufgabe erfüllt. Die Leistungsfähigkeit des objektorientierten Ansatzes (und damit der objektorientierten Systeme) entsteht eigentlich erst dadurch, dass die einzelnen Klassen in genau definierter Art und Weise miteinander interagieren. Der Prozess, der die entsprechenden Klassen und ihre Interaktionen definiert, wird als Design (bzw. auf abstrakterer Ebene als Architektur) bezeichnet. Da ein gutes Design sehr aufwändig sein kann und üblicherweise eine vorhergehende Analyse der zu modellierenden Domäne verlangt, sollte die Wiederverwendung von Design und Architektur eigentlich mindestens genau so wichtig sein wie die Wiederverwendung von Fachklassen. Um dieser Anforderung gerecht zu werden, gibt es mehrere Möglichkeiten.

Beispielsweise liefern Design Patterns (Entwurfsmuster) wiederverwendbare Lösungen für typische Problemstellungen, wobei zusätzlich zu Problem und Lösung auch die (positiven und negativen) Konsequenzen sowie Beispiele und bekannte Verwendungen mit angegeben werden. Design Patterns bieten somit Wiederverwendung von Design-Erfahrung auf konzeptionellem Level, sie bündeln Expertenwissen in wiederverwendbarer Form.

Einen Schritt weiter gehen Frameworks. Sie versuchen, Architektur und Design von ganzen Anwendungsfamilien zu „konservieren" und damit wiederverwendbar zu machen. Dies steht im Gegensatz zu Klassenbibliotheken, wo einzelne Bausteine wiederverwendet werden sollen, und die Anwendungsprogrammierer daraus neue Anwendungen erstellen. Bei Frameworks wird das Anwendungsskelett wiederverwendet und der Programmierer erstellt bestimmte Bausteine, mit denen das Framework konfiguriert wird.

Um also allen potentiellen Anwendungen einer Familie (den Instanzen des Frameworks) gerecht werden zu können, muss dafür gesorgt werden, dass das Framework an allen nötigen Stellen genügend Anpassbarkeit aufweist. Man nennt solche Stellen auch Hot-Spots (nach Pree, [PREE]). Diese Hot-Spots lassen sich dann oft durch den Einsatz von

Design Patterns elegant umsetzen. Im Gegensatz zu Patterns, die Wiederverwendung auf konzeptioneller Ebene bieten, bestehen Frameworks wirklich aus konkretem Code, der die Architektur und das grundlegende Design einer Anwendungsfamilie definiert.

Bevor allerdings mit der Implementierung eines Framework begonnen werden kann, muss durch Analyse des Anwendungsgebietes herausgearbeitet werden, welche Bestandteile eines Frameworks fest implementiert werden können oder sollen, und welche variabel gehalten werden müssen. Eine derartige Analyse wird als Domain Analysis bezeichnet (siehe unten).

Frameworks und Komponenten

Frameworks als Mittel zur Implementierung vom Komponenten

Komponenten sollen ein vernünftiges Maß an Konfigurierbarkeit besitzen, um in möglichst vielen Anwendungen wiederverwendet werden zu können. Aufgrund der aspektuellen Trennung bei Komponenten bezieht sich diese Konfigurierbarkeit vor allem auf die Businesslogik. Insofern stellt eine Komponente eine „Anwendungsfamilie“ dar. D.h. es liegt nahe, eine Komponente als Framework zu implementieren und beim Einsatz der Komponente eben eine entsprechend konfigurierte Instanz des Frameworks zu verwenden.

Damit kommen auch Analysemethoden aus dem Framework-Umfeld zum Einsatz. Deren Ansatz ist es ja gerade, die Variabilität einer Anwendungsfamilie (hier dann einer Komponente) herauszuarbeiten.

Ein Ansatz ist die Feature-Oriented-Domain-Analysis (FODA, [SEI]). Diese ist folgendermaßen definiert:

> *Feature-Oriented-Domain-Analysis beschreibt die wichtigsten Eigenschaften von Anwendungen einer Domäne aus Sicht des Kunden oder Endbenutzers.*

Solche Features einer Systemfamilie können verschiedenen Ausprägungen haben:

- Pflicht-Features: Jede Anwendung der Familie muss das Pflicht-Feature aufweisen. Dies entspricht den fixen Teilen eines Frameworks (Jedes *Auto* hat einen *Motor*).
- Wahlpflicht-Features: Jede Anwendung muss ein Feature aus einer Menge alternativer Features haben (Ein *Auto* hat entweder ein *Schaltgetriebe* oder ein *Automatikgetriebe*).
- Optionale Features: Eine Anwendung kann ein Feature aufweisen oder auch nicht (Ein *Auto* kann eine *Klimaanlage* haben).

Features können beliebig hierarchisch verschachtelt werden. Desweiteren kann es Abhängigkeiten zwischen Features geben. Ein Beispiel könnte sein, dass alle Autos die das optionale Feature *Tempomat* aufweisen, ein *Automatikgetriebe* besitzen.

Diese Features und ihre Abhängigkeiten lassen sich in sogenannten Feature-Diagrammen darstellen. Von FODA macht übrigens auch die generative Programmierung ausführlich Gebrauch, im Buch von Eisenecker und Czarnecki [ECGP] findet sich eine sehr gute Beschreibung dieser Methode.

Das so aufgestellte Feature-Modell kann nun weiter verfeinert werden. Ein Feature entspricht üblicherweise einem (gegebenenfalls auch mehreren) Hot-Spot. Im Gegensatz zu Features geht die Modellierung eines Frameworks mittels Hot-Spots allerdings bereits darauf ein, wie ein Feature softwaretechnisch umgesetzt werden kann. Ein Hot-Spot besteht daher üblicherweise aus mehreren Hooks (siehe [PREE]), die zur Implementierung des Hot-Spots dienen. Hooks können verschiedene Typen haben:

- Schalter: Innerhalb des Frameworks muss ein Flag angepasst werden.
- Auswahl: Der Entwickler muss sich für eine Möglichkeit aus einer vorgegebenen Menge von Alternativen entscheiden.
- Programmierung: Der Entwickler muss selbst ein Stück Code erstellen, das bestimmten Anforderungen genügen muss; es muss also beispielsweise ein bestimmtes Interface implementieren oder von einer bestimmten Basisklasse erben, etc.

Auf diese Art und Weise kann die Variabilität eines Systems und damit auch einer Komponente definiert werden. Eine systematische Implementierung einer Komponente auf Basis von Framework-Technologie kann eine Komponente wartbarer, flexibler und damit effizienter machen.

Definition der Granularität von Komponenten

Komponenten sollen per Definition wiederverwendet werden, und zwar dadurch, dass Anwendungen aus fertigen Komponenten „assembliert“ werden, wobei einzelne Komponenten noch angepasst werden können oder müssen. Um dabei ein möglichst hohes Mass an Wiederverwendbarkeit zu erreichen, ist es erforderlich, die Granularität der Komponenten richtig zu definieren.

Verschiedene Aspekte spielen bei der Festlegung der Granularität eine Rolle; unter anderem sind dies

- Performanz und Skalierbarkeit: Lastverteilung findet üblicherweise pro Komponente statt.

- Installationsaufwand: Je mehr Komponenten ein System besitzt, desto aufwendiger ist dessen Installation und das Management der Abhängigkeiten bezüglich Versionen und Änderungen der Schnittstelle.

Der wichtigste Gesichtspunkt ist allerdings sicherlich der, dafür zu sorgen, dass eine Komponente nur Funktionalitäten in sich birgt, die immer gemeinsam verwendet werden. Ansonsten sind Clients von Funktionalitäten abhängig, die sie gar nicht benötigen.

Auch für diesen Zweck ist also eine systematische Analyse der Anwendungsdomäne notwendig. Eine Domänen-Analyse mittels FODA kann auch hier helfen. Nun wird allerdings nicht eine Komponente als Domäne betrachtet um deren variable Aspekte zu definieren, vielmehr wird hier ein komplettes System betrachtet um dessen variable und fixe Anteile herauszufinden. Auch ein solches System lässt sich üblicherweise als Feature-Model darstellen.

Wenn als Ergebnis der Modellierung bestimmte Features immer gemeinsam auftauchen (also abhängig sind), dann liegt es nahe, die Verantwortlichkeit für diese Features in einer Komponente zu sammeln. Andernfalls sind mehrere Komponenten sinnvoller.

Frameworks für die nicht-fachlichen Aspekte

Ziel der komponentenbasierten Softwareentwicklung ist die aspektuelle Trennung der ehemaligen Klassen in die eigentlichen Fachklassen und separate Aspekte für Security, Persistenz, Verteilung usw.

Wenn nicht auf „fertigen“ Containern aufgesetzt wird (wie z.B. Enterprise JavaBeans, siehe unten) so müssen die nicht-fachlichen Aspekte natürlich trotzdem implementiert werden. Typisch für die nicht-fachlichen Aspekte ist allerdings eine sehr starke Systematik bei der Implementierung. Das bedeutet, das üblicherweise einige (evtl. nicht-triviale) grundlegende Entscheidungen getroffen werden müssen. Wenn diese dann aber getroffen sind, folgt die Implementierung einem ganz strikten Schema.

Nehmen wir als Beispiel die Persistenz. Es gibt verschiedene Möglichkeiten, die in Frage kommen. Abspeicherung in objektorientierten Datenbanken, als XML Dokumente oder, üblicherweise, in relationalen Datenbankmangementsystemen. Wenn die Entscheidung beispielsweise für ein RDBMS gefallen ist, so muss (etwas vereinfacht) nur noch angegeben werden, welche Attribute in welche Tabelle/Spalte geschrieben werden sollen und welche Datenbank zum Einsatz kommt. Der Rest kann dann in irgendeiner Weise automatisiert werden.

Ziel könnte es also sein, die nicht-fachlichen Aspekte mittels verschiedener Frameworks zu implementierten. Auch hier ist wieder eine systematische Analyse des Anwendungsgebietes notwendig, auch wenn es sich hierbei um ein nicht-fachliches handelt. Beispielsweise gibt es verschiedenste Möglichkeiten, Objektgraphen (incl. Referenzen und Vererbung) auf ein relationales Schema abzubilden, Performanz oder die

Kompatibilität zu einem bereits definierten Datenbankschema können dabei eine Rolle spielen.

Ist diese Analyse erfolgreich durchgeführt und das Framework implementiert, so wird der Aspekt der Persistenz bei den zukünftigen Anwendungen sehr einfach zu implementieren sein. Dies ist ja auch ein gängiges Vorgehen. Es gibt auf dem Markt diverse Objekt-Relational-Mapping Frameworks, die die Tauglichkeit dieses Ansatzes unter Beweis stellen. Für die anderen technischen Aspekte lässt sich prinzipiell genauso vorgehen.

Komponentenarchitekturen sind üblicherweise selber Frameworks

Wie oben bereits erläutert, brauchen Komponenten üblicherweise eine Laufzeitumgebung, die die nicht-fachlichen Aspekte erledigt. Diese kann prinzipiell selbst implementiert werden. Allerdings liegt der Vorteil von Komponenten eben gerade darin, dass diese nicht-fachlichen Aspekte so allgemein anwendbar sind, dass es eben auch generische Frameworks dafür geben kann. Genau dies ist die Aufgabe von Komponentenarchitekturen wie Enterprise JavaBeans. Dies soll im Folgenden Beispiel kurz angerissen werden.

Der Programmierer implementiert (im Idealfall einer EntityBean mit Container Managed Persitence) nur die Businesslogik. Er muss bei der Programmierung der Komponente einigen Regeln folgen, damit die Komponente dann im Framework des Containers existieren kann. Beispiele sind Namenskonventionen für den Zusammenhang zwischen dem Interface der Bean und deren Implementierung, oder das Vorhandensein bestimmter Operationen (*create/ejbCreate()*). Damit die nicht-fachlichen Aspekte korrekt ausgeführt werden können, muss der Programmierer der Bean (und/oder der Deployer) noch diverse Einstellungen vornehmen. Dies entspricht der Konfiguration des Frameworks. Das Framework in Form des Containers bzw. Application Servers kann die Bean dann z.B. per Java Reflection ansprechen und im Container zum Laufen bringen.

Zusammenfassung

Das Ziel von komponentenbasierter Entwicklung ist, durch die aspektuelle Trennung die Möglichkeit zur Wiederverwendung bei Fachklassen zu erhöhen. Frameworks dienen dazu, die Architektur und das Design von Anwendungsfamilien wiederverwendbar zu machen.

Wie in diesem Artikel ersichtlich wurde, sind komponentenorientierte Entwicklung und Frameworks kein Widerspruch, sondern zwei Technologien, die sich gegenseitig ergänzen können: Komponenten sind das Ziel, und Frameworks können einen effektiven Weg dorthin darstellen.

Literatur

[AOP] Xerox PARC, *Aspect Oriented Programming*, http://www.parc.xerox.com/csl/projects/aop/

[ECGP] U. W. Eisenecker, K. Czarnecki, *Generative Programming*, Addison-Wesley 2000

[PREE] Wolfgang Pree, *Design Patterns for Object-Oriented Software Development,* Addison-Wesley 1995

[SEI] Software Engineering Institue, *Feature-Oriented Domain Analysis*, http://www.sei.cmu.edu/str/descriptions/foda.html