Prototypische Vorgehensweisen für den Entwurf anwendungsnaher Komponenten

Prof. Dr. Ulrich Frank
Dipl.-Inform. Jürgen Jung
Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
56075 Koblenz
{ulrich.frank | jjung}@uni-koblenz.de

Zusammenfassung

Der Einsatz sorgfältig entworfener, anwendungsnaher Komponenten, auch Fachkomponenten genannt, stellt eine wirtschaftlich reizvolle Vision dar, da er eine Verbesserung der Qualität betrieblicher Informationssysteme bei gleichzeitig verringerten Erstellungskosten verspricht. Die Untersuchung solcher Komponenten ist zumeist entweder auf software-technische Eigenschaften oder ihre Verwendung in Anwendungssystemen gerichtet. Demgegenüber spielt die Frage, wie Fachkomponenten entwickelt werden sollen, eine untergeordnete Rolle. Das ist insofern erstaunlich, als der Nutzen, den die Verwendung von Fachkomponenten mit sich bringt, wesentlich davon abhängt, ob sie angemessen entworfen wurden. Dabei stellt sich nicht zuletzt die Frage, ob der Entwurf von Fachkomponenten eine spezifische Vorgehensweise erfordert. Der vorliegende Beitrag ist auf die Untersuchung dieser Frage gerichtet. Dazu werden zunächst dafür bedeutsame Unterschiede zwischen Komponenten und Objekten identifiziert. Anschließend werden drei prototypische Vorgehensweisen für den Entwurf von Komponenten vorgestellt und im Rahmen einer Fallstudie miteinander verglichen.

1. Einleitung

Seit einiger Zeit ist in der Literatur ein reges Interesse an anwendungsnahen Komponenten, auch Fachkomponenten genannt, zu verzeichnen. Die an den Einsatz solcher Artefakte geknüpften Erwartungen stellen eine Wiederbelebung der alten Vision wiederverwendbarer Software-Bausteine dar. In den einschlägigen Publikationen wird neben den Grundlagen der Komponententechnologie ([Szyp97], [Grif98]) vor allem die *Verwendung* von Komponenten oder deren Entwicklung im Rahmen eines spezialisierten Entwicklungsprozesses thematisiert. Dazu gehören Arbeiten über die software-technische Komposition von Komponenten ([NiDa95]), die Architektur von komponentenbasierten Anwendungssystemen ([Pree97], [Spro00], [HeSi01]), die Erweiterung von Anwendungssystemen durch Fachkomponenten ([RTF99]) oder den Entwurf neuer Softwaresyteme im Hinblick auf die Entwicklung von Komponenten ([SoWi99], [AlFr98]). Demgegenüber spielt die Frage danach, wie anwendungsnahe Komponenten auf Basis vorhandener Entwurfsdokumente oder Softwaresy-

steme zu entwerfen sind, eine untergeordnete Rolle. Das ist einerseits verständlich, da Komponenten häufig als eine zusätzliche Abstraktion über den softwaretechnischen Konzepten von Modellierungs- oder Implementierungssprachen angesehen werden. Ebenso ist die Verwendung von Komponenten oftmals nicht an eine spezifische Implementierungssprache gebunden. Andererseits ist die Vernachlässigung des Entwurfs von Fachkomponenten auf Basis vorhandener Entwurfsdokumente erstaunlich, da der mit Komponenten verbundene Anspruch auf autonome Wiederverwendbarkeit besondere Anforderungen an deren Entwurf stellt. Dabei stellt sich zunächst die Frage, ob gängige Abstraktionen, wie sie etwa durch objektorientierte Modellierungssprachen unterstützt werden, eine hinreichende konzeptionelle Grundlage für den Entwurf von Fachkomponenten darstellen. Daneben ist zu klären, ob bewährte Vorgehensweisen für den Entwurf betrieblicher Informationssysteme auch geeignet sind, um die Entwicklung von Fachkomponenten zu unterstützen. Der vorliegende Beitrag ist darauf gerichtet, diese Fragen zu untersuchen. Vor dem Hintergrund der Diskussion von Komponenten in der Literatur wird dazu zunächst ein Komponentenbegriff konkretisiert, der eine Abgrenzung gegenüber Klassen bzw. Objekten ermöglicht. Anschließend werden anhand einer Fallstudie drei prototypische Vorgehensweisen für den Komponentenentwurf dargestellt und evaluiert.

2. Komponenten: Begriffliche Grundlagen, konzeptionelle und software-technische Anforderungen

Für die in diesem Beitrag zu untersuchenden Fragen ist es wesentlich, einen Begriff von Komponenten zu entwickeln, der geeignet ist, Unterschiede zu anderen Abstraktionen der Software-Entwicklung zu verdeutlichen. Zunächst greifen wir verschiedene Blickwinkel auf den Begriff der Komponente in der Literatur auf. Anschließend werden Komponenten gegenüber den Begriffen Klasse und Objekt aus der objektorientierten Softwareentwicklung abgegrenzt.

2.1 Der Komponentenbegriff in der Literatur

Es herrscht weitgehend Einigkeit darüber, dass eine Komponente eine software-technische Abstraktion ist. Desweiteren wird allgemein davon ausgegangen, dass eine Komponente ihre innere Struktur verkapselt und ihre Dienste über eine wohldefinierte Schnittstelle bzw. ein Protokoll anbietet: "A software component is a static abstraction with plugs." ([NiDa95, S. 5)). Ergänzend dazu wird häufig ein Ereignismechanismus gefordert (vgl. [Szyp97], [Balz00], [SGC99]), der auch in einigen Komponentenmodellen (z.B. JavaBeans, ActiveX, CORBA Component Model) verfügbar ist. Über ein Ereignis kann eine Komponente andere Module (Beobachter) bspw. über eigene Zustandsänderungen oder komponentenintern auftretende Benutzerinteraktionen informieren.

Daneben gibt es eine Reihe von Eigenschaften, die nicht einheitlich behandelt werden.[1] Einige Auto-

1. Ein Überblick über die Diskussion findet sich in [BDH+98].

ren betonen explizit, dass Komponenten in Binärform vorliegende Artefakte sind und gleichzeitig plattformunabhängig einsetzbar sein sollten (z.B. [SGC99], [Szyp97]). Beide Forderungen sind nur unter der Voraussetzung kompatibel, dass eine Kommunikationsinfrastruktur existiert, die Unterschiede zwischen Programmiersprachen, Betriebssystemen und Hardware zu verbergen gestattet. Im Hinblick auf die Verwendung von Komponenten wird häufig darauf hingewiesen, dass sie autonom eingesetzt werden können, ihre Funktionsfähigkeit also nicht von weiteren speziellen Ressourcen abhängt. In einigen Fällen wird die Größe einer Komponente thematisiert. Während Orfali et al. [OHE96] betonen, dass eine Komponente keine vollständige Applikation darstellt, verweist Szyperski ([Szyp97], S. 11) darauf, dass selbst komplexe Systeme wie ein Betriebssystem oder ein Datenbankmanagement-System als Komponenten angesehen werden können: Sie sind in hohem Maße über klar definierte Schnittstellen wiederverwendbar. Nierstrasz und Dami ([NiDa95]) bezeichnen eine Komponente als eine statische Abstraktion, weil sie sie als eine langlebige Einheit ansehen, die in einer Softwarebasis unabhängig von der sie benutzenden Anwendung gespeichert werden kann. Nach Szyperski ([Szyp97], S. 34) ist eine Komponente "a unit of composition with contractually specified interfaces and explicit context dependencies only", die keinen Zustand und keine Identität hat, d.h. sie kann in einer Anwendung nur einmal vorkommen. Am Markt angebotene Komponenten sind mit dem von Szyperski vorgeschlagenen Begriff nur eingeschränkt kompatibel. So kann ein JavaBean in einer Anwendung mehrfach vorkommen - die Unterscheidung erfolgt durch Namensgebung bei der Verwendung einer Komponente. Auch haben Komponenten "Properties", die Werte festhalten und somit auch einen Zustand besitzen. Wie sich noch zeigen wird, ist für unsere Betrachtung die Forderung, dass Komponenten auch in Kontexten einsetzbar sein sollten, die zum Zeitpunkt ihres Entwurfs nicht bekannt waren ([Szyp97], S. 34), von großer Bedeutung.

Auch die Vorstellungen darüber, was unter einer Fachkomponente zu verstehen sei, variieren erheblich. So ist für die OMG ein "business object" lediglich "a representation of a thing active in the business domain ..." ([OMG96]), wobei es sich dabei auch um eine lediglich natürlichsprachliche Repräsentation handeln kann. Nach Sims ([Sims94]) hat ein "business component" u.a. eine graphische Repräsentation und kann von Anwendern ohne Programmierkenntnisse unmittelbar genutzt werden. Im Hinblick auf den Fokus dieses Beitrags ist festzuhalten, dass auch für Fachkomponenten i.d.R. gefordert wird, sie sollen unabhängig von anderen speziellen Ressourcen ausführbar sein und mit beliebigen anderen Komponenten kommunizieren können ([RTF99]).

2.2 Wesentliche Merkmale von Komponenten und eine Abgrenzung zu Objekten

Jenseits aller Unterschiede, die sich bei der Betrachtung von Nominaldefinitionen wie auch von realen Komponentenimplementierungen zeigen, können wir zusammenfassend feststellen, dass Kom-

ponenten eine Abstraktion über Klassen, Modulen oder Systemen darstellen. Es gibt also nicht aus allen Blickwinkeln einen notwendigen Unterschied zu diesen anderen software-technischen Artefakten. Dabei ist allerdings zu berücksichtigen, dass auch die Bedeutung der Begriffe "Klasse" bzw. "Objekt" nicht einheitlich ist. Hier ist etwa an die Unterscheidung zwischen extensionalem und intensionalem Klassenbegriff, zwischen klassenbezogenem und klassenlosem Objektbegriff sowie die sich daraus ergebenden Konsequenzen etwa für die Semantik von Spezialisierungsbeziehungen zu denken. Damit stellt sich die Frage, wie sich Fachkomponenten konzeptionell sinnvoll von Objekten (bzw. Klassen) unterscheiden lassen.

In der Literatur wird eine Reihe von Unterschieden zwischen Komponenten und Objekten nahegelegt. Das gilt offensichtlich für die Forderung, Komponenten sollten keinen Zustand und keine Identität haben, sowie nicht persistent sein ([Szyp97]). Im Hinblick auf die Schaffung betriebswirtschaftlicher Fachkomponenten ist u.E. eine solche Forderung allenfalls dann sinnvoll, wenn man eine Komponente mit einer Klasse vergleicht, aus der zustandsbehaftete, persistente Instanzen (oder sonstige Datenstrukturen) mit eigener Identität erzeugt werden können. Im Licht einer solchen Interpretation würden Szyperskis Forderungen nicht implizieren, dass der Entwurf von Fachkomponenten ein anderes Vorgehen erfordert als der von objektorientierten Systemen im allgemeinen.

Der für unsere Betrachtung wesentliche Unterschied zwischen Objekten und Komponenten resultiert aus dem intendierten Verwendungszweck: Komponenten sollen unabhängig voneinander auch in solchen Domänen einsetzbar sein, deren Besonderheiten beim Entwurf der Komponenten nicht berücksichtigt wurden. Aus diesen Anforderungen folgt zunächst ein wesentliches Merkmal, das einen konzeptionellen Unterschied zwischen Komponenten und Objekten bzw. Klassen markiert: Fachkomponenten sollten untereinander eine möglichst geringe *spezifische Kopplung* aufweisen. Kopplung ist allgemein ein Indikator für das Ausmaß der Abhängigkeit zwischen Software-Artefakten. Eine Minimierung der Kopplung ist allerdings auch für Fachkomponenten kaum erstrebenswert: Es ist ja eine essentielle Anforderung, sie mit anderen Komponenten zu sinnvollen Anwendungen komponieren zu können. Unter spezifischer Kopplung verstehen wir demgegenüber eine Abhängigkeit von speziellen Ressourcen - etwa von anderen Fachkomponenten oder auch von speziellen Klassenbibliotheken. Während die spezifische Kopplung zwischen Fachkomponenten null sein sollte, empfiehlt der objektorientierte Systementwurf i.d.R. ein hohes Maß an spezifischer Kopplung (wobei deren Angemessenheit im Einzelfall zu prüfen ist) zwischen Klassen. Diese ist auch bei der Entwicklung von Komponenten solange nicht ausgeschlossen, wie es sich dabei um Klassen einer Komponente handelt. Die Unterscheidung zwischen zulässiger Kohäsion und unzulässiger Kopplung macht sich also an der Identifikation bzw. Abgrenzung von Komponenten fest. Die spezifische Kopplung konkretisiert sich in zwei formalen Anforderungen an den Entwurf von Fachkomponenten:

1. Klassen einer Komponente sollten keine Assoziationen zu Klassen außerhalb der Komponente besitzen.
2. Die Signaturen der Schnittstelle einer Komponente sollten keine speziellen Typen bzw. Klassen enthalten.

Die zweite Forderung ist teilweise bereits durch die erste abgedeckt, aber nicht redundant, da Signaturen mit speziellen Typen auch dann denkbar sind, wenn keine Assoziation zu diesem Typ besteht. Es bleibt darauf hinzuweisen, dass diese Anforderungen nicht implizieren, beim Entwurf von Komponenten auf entsprechende Abstraktionen völlig zu verzichten: Die innere Struktur einer Komponente kann ja durchaus aus Klassen bestehen, die untereinander ein hohes Kohäsionsmaß aufweisen. Vererbungsbeziehungen werden in den beiden Anforderungen nicht betrachtet, obwohl dies nicht als gänzlich unkritisch zu betrachten ist. Im Verlauf der Entwicklung einer Komponente werden benötigte Superklassen automatisch in die Komponente hineingezogen und als Teil der Komponente distribuiert. Problematisch kann dies jedoch im Rahmen der korrektiven Wartung werden, da sich Änderungen an Superklassen auch auf die Klassen einer Komponente auswirken können. Die Verwaltung von Abhängigkeiten von den eigentlichen Klassen einer Komponente zu relevanten Änderungen an ihren Oberklassen kann - ein umfangreiches Objektmodell vorausgesetzt - sehr hoch werden.

3. Fallstudie: Ein Referenzmodell für den elektronischen Handel

Die skizzierten Anforderungen an den Entwurf von Komponenten stecken lediglich formale Randbedingungen ab, sie geben wenig Aufschluss darüber, wie im Einzelfall vorzugehen ist, um zur Spezifikation angemessener Fachkomponenten zu gelangen. Um Ansatzpunkte dafür zu erhalten, wie beim Entwurf von Fachkomponenten vorzugehen ist, greifen wir im folgenden auf eine Fallstudie zurück. Es handelt sich dabei um ein Referenzmodell für Handelsplattformen im Internet ([Fran00b]). Das Referenzmodell wurde in der üblichen Vorgehensweise erstellt: Ausgehend von einer Betrachtung generischer Strategien für das Betreiben von Handelsplattformen im Internet wurden ca. 20 prototypische Geschäftsprozesstypen entwickelt, die u.a. mit Hilfe von Anwendungsfällen näher beschrieben wurden. Auf dieser Grundlage entstand ein Objektmodell mit ca. 170 Klassen. Die Implementierung erfolgte in Java. Um Objekte persistent zu machen, wird mittels eines objektorientierten Frameworks transparent auf eine relationale Datenbank zugegriffen. Das Referenzmodell enthält eine Fülle von Konzepten, die aus gängigen Systemen, die in Handelsunternehmen eingesetzt werden, bekannt sind: Kunden, Lieferanten, Produkte, Lieferdokumente etc. Um eine große Einsatzbandbreite des Modells zu erreichen, wurden Abstraktionen gewählt, die es gestatten, verschiedene Preisbildungsmechanismen abzudecken. Zur Flexibilität des Modells trägt nicht zuletzt

eine metamodell-orientierte Modellierung von Produkten bei, die es erlaubt, beliebig differenzierte neue Produkttypen ohne eine Änderung des Codes bzw. des Datenbankschemas zu erfassen ([Fran00a]). Abb. 1 illustriert die Vorgehensweise bei der Erstellung des Referenzmodells.

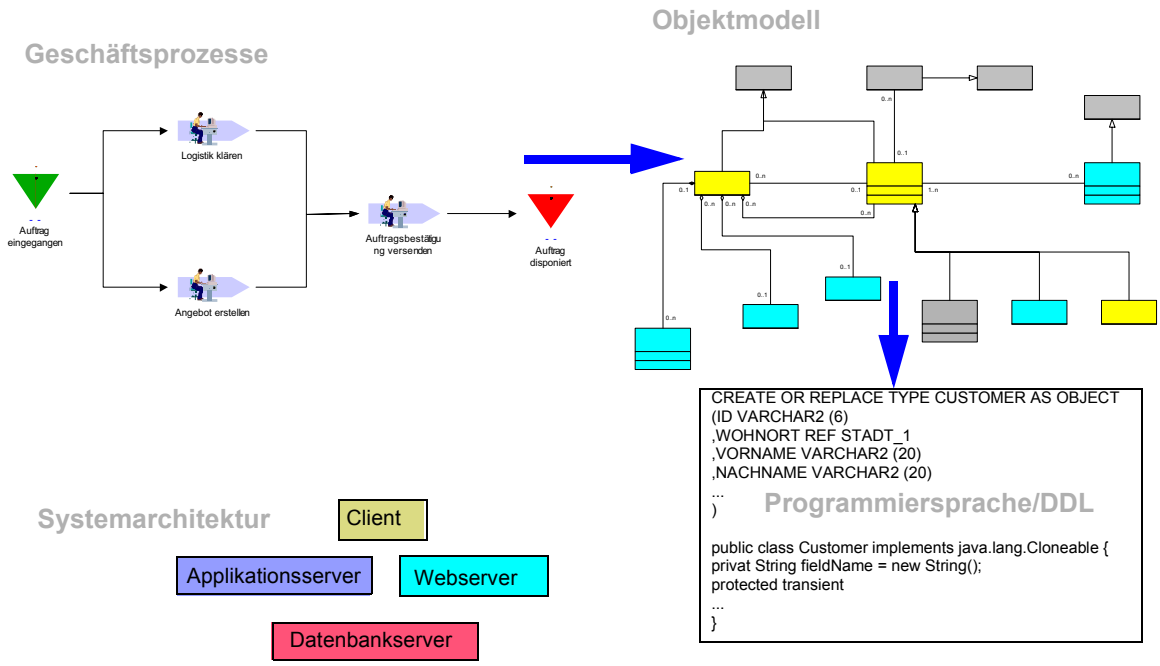


Abb. 1: Von Geschäftsprozessmodellen über das Objektmodell zu Klassendeklarationen bzw. Datenbankschemata. Links unten die Grobarchitektur der möglicher Zielsysteme

Um die Pflege des Referenzmodells zu erleichtern, wurden bei der Modellierung konsequent objektorientierte Abstraktionskonzepte (Generalisierung/Spezialisierung, Spezifikation von Attributen auch mittels Klassen) verwendet. Insgesamt weist das Modell 50 Spezialisierungsbeziehungen und über 200 Assoziationen auf. Damit kommen die im Objektmodell enthaltenen Klassen unmittelbar i.d.R. nicht als Komponentenkandidaten in Frage.

4. Prototypische Strategien für den Entwurf von Fachkomponenten

Das dargestellte Objektmodell bildet den Ausgangspunkt für die weitere Untersuchung. Zunächst betrachten wir zwei Ansätze, die auf die Rekonstruktion eines Komponentenmodells aus einem Objektmodell gerichtet sind. Anschließend setzen wir uns mit der Frage auseinander, ob es sinnvoll ist, ein Komponentenmodell direkt, ohne den Umweg über ein Objektmodell, zu entwerfen.

4.1 Ein semi-formaler Ansatz zur Identifikation von Komponenten

Im Rahmen der ersten Untersuchung wird das System unter Zuhilfenahme formaler Kriterien auf isolierbare Teilsysteme untersucht. Grundannahme für diese Untersuchung ist, dass sich Komponenten als spezielle Form von Modulen durch besondere softwaretechnische Eigenschaften auszeichnen. Verfahren zur Identifikation angemessener Module sind bereits in einigen Studien (vgl. [LöSi00], [AbGo01]) analy-

siert worden. Diese Studien basieren auf der Partitionierung existierender Softwaresysteme aufgrund geeigneter Kohäsionsmaße objektorientierter Systeme. Grundsätzlich gilt die Annahme, dass Klassen innerhalb eines Moduls eine enge Kopplung aufweisen sollten. Man spricht in einem solchen Fall auch von einer hohen Kohäsion des Moduls. Ein hierbei u.a. anvisiertes Einsatzfeld ist die Aufteilung von Klassen in disjunkte Teilmengen (Partitionen) wie bspw. Pakete (packages) in Java (vgl. [LöSi00]). Klassen innerhalb eines solchen Paketes sollten eine der Anwendung entsprechende Aufteilung der Klassen in besser wartbare Teilsysteme ermöglichen. Ein idealtypisches Verfahren für eine solche Partitionierung gliedert sich in folgende Schritte:

1.) Festlegung eines geeigneten Kopplungsmaßes für Klassen

2.) Berechnung von Distanzen zwischen den Klassen des Systems

3.) Durchführung einer Cluster-Analyse auf den zuvor ermittelten Distanzen

Der erste Schritt erfordert eine sorgfältige Evaluation und Abwägung objektorientierter Metriken. Wir haben für diese Untersuchung *Coupling between Objects* (CBO) gewählt, auch "design-fan-out" genannt ([Hend96], S. 159). Diese Entscheidung ist durch die Popularität von CBO und seine Aussagekraft bzgl. der Kopplung zwischen Objekten resp. Klassen motiviert. CBO ermittelt die Beziehungen einer Klasse zu anderen, indem Referenzen auf Klassen über Typen der Attribute oder Assoziationen sowie über formale Parameter der Methoden erfasst werden. Generalisierungsbeziehungen werden bei CBO üblicherweise nicht berücksichtigt. Wir haben im Gegensatz hierzu die Menge der direkt referenzierten Klassen um die Menge der von den Oberklassen referenzierten Klassen erweitert. Der nötige Input wird also durch ein hinreichend differenziertes Objektmodell gewährleistet. Darüber hinaus können mit CBO auch die Klassen methoden-lokaler Variablen berücksichtigt werden, die allerdings in Objektmodellen üblicherweise nicht erfasst sind.

Auf Basis dieses Software-Maßes werden paarweise Distanzen zwischen Klassen berechnet. Eine solche Distanz zwischen zwei Klassen K_1 und K_2 ist ein Indikator für die Kopplung dieser beiden Klassen. In Anlehnung an [LöSi00] wird folgende Formel zur Berechnung der Distanz übernommen:

$$distance(K_1, K_2) = 1 - \frac{|b(K_1) \cap b(K_2)|}{|b(K_1) \cup b(K_2)|}$$

Die Distanz zwischen zwei Klassen wird dabei indirekt gemessen: als das Verhältnis der Zahl der von beiden Klassen gemeinsam referenzierten Klassen zur Zahl der insgesamt von beiden Klassen referenzierten Klassen. Dabei liefert die Funktion b(K) jeweils die Menge der von der Klasse K referenzierten Klassen inkl. K selbst sowie die von ihrer Superklassen referenzierten Klassen. Die Distanz zwischen zwei Klassen ist gemäß dieser Formel tendenziell hoch, wenn sie relativ wenige

Klassen gemeinsam referenzieren. Bei disjunkten Mengen hat die Distanz den Wert eins, das Maximum.

Die Identifikation von Clustern kann von den Distanzen ausgehend manuell oder semiautomatisch erfolgen. Semiautomatische Verfahren werden bspw. in [LöSi00] und [AbGo01] vorgestellt. Im Rahmen dieser Untersuchung wurde die Auswertung mangels geeigneter Werkzeugunterstützung manuell durchgeführt. Für solch eine manuelle Untersuchung der Distanzen auf Cluster ist eine geeignete Visualisierung angeraten, da bei k Klassen die Anzahl der Distanzen $k^2/2$ beträgt. Eine entsprechende graphische Aufbereitung ist allerdings mit einer Herausforderungen verbunden: Der durch die Distanzen von n Klassen aufgespannte Raum ist im Extremfall (n-1)-dimensional. Deswegen haben wir diesen Raum auf eine dreidimensionale Darstellung projeziert (das verwendete Werkzeug und seine grundlegenden Algorithmen werden ausführlicher in [SSL00] vorgestellt). Eine solche Projektion allein liefert zunächst nur einzelne Anhaltspunkte für mögliche Cluster, die anschließend durch eine sorgfältige Betrachtung der ermittelten Distanzen zu überprüfen sind.

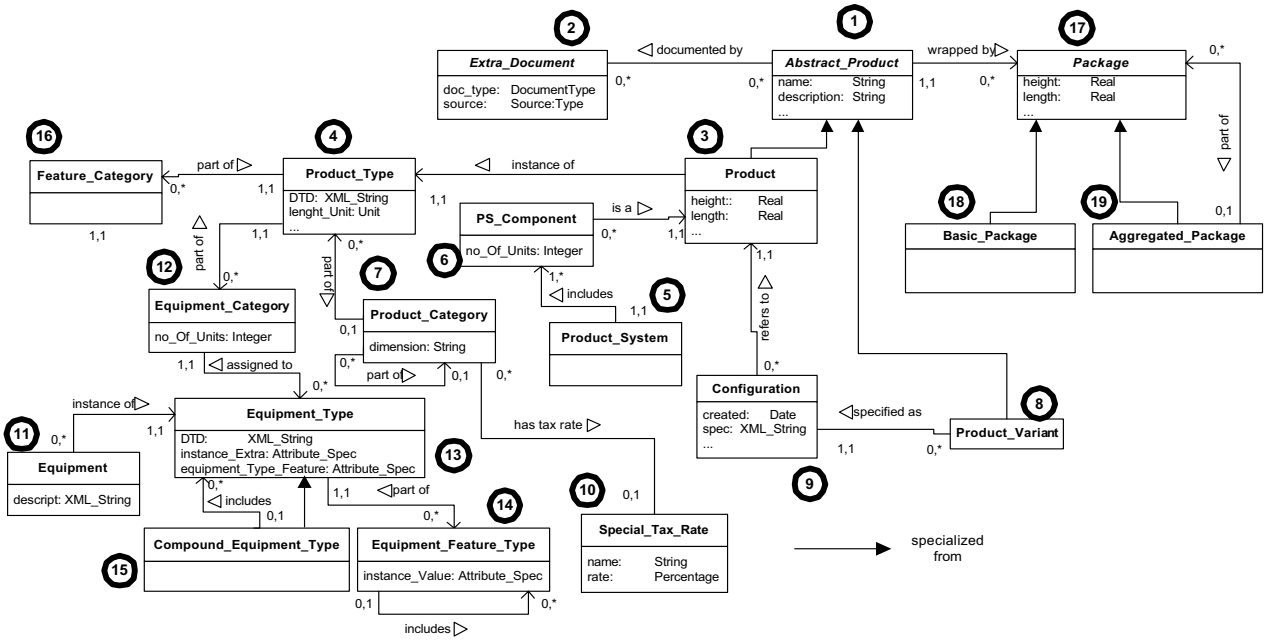


Abb. 2: Auschnitt aus dem Referenzmodell zur Beschreibung der Ergebnisse

Die Ergebnisse unserer Untersuchung werden im Folgenden exemplarisch anhand des in Abb. 2 dargestellten Ausschnitts aus dem Referenzmodell erläutert. Abb. 3 stellt die drei-dimensionale Projektion der Distanzen dieses Auschnitts dar. Als ein möglicher Komponentenkandidat wurde aufgrund geringer Distanzen der aus den Klassen 17, 18, 19 bestehende Cluster identifiziert. Hierbei handelt es sich um die Klasse *Package* (17) mit ihren beiden Subklassen *Basic_Package* (18) und *Aggregated_Package* (19). Trotz der hohen Kohäsion dieses Clusters sind die enthaltenen drei Klas-

sen keine Kandidaten für eine Komponente, da sie noch eine hohe spezifische Kopplung zu der Klasse *Abstract_Product* und deren Subklassen aufweisen. Der der aus den Klassen 11, 12, 13, 14 und 15 gebildete Cluster ist aus dem gleichen Grund kein Komponentenkandidat. Diese Klassen dienen der Beschreibung der Ausstattung spezifischer Produkte (vgl. Abb. 2). Beide Cluster könnten u.U. als anwendungsnahe Muster interpretiert werden. Ihre hohe spezifische Kopplung und die damit verbundene Komplexität der Schnittstelle steht einer Verwendung als Komponente entgegen. Auch die Klassen 1 (*Abstract_Product*) und 2 (*Extra_Document*) können trotz ihrer sehr geringen Distanz (vgl. Abb. 3) nicht zu einer Komponente zusammengefaßt werden. Beide sind abstrakte Superklassen, deren Kopplung allein aus der exklusiven gegenseitigen Referenzierung resultiert. Aufgrund der Spezialisierungen in den Subklassen steigt hier jedoch die spezifische Kopplung.

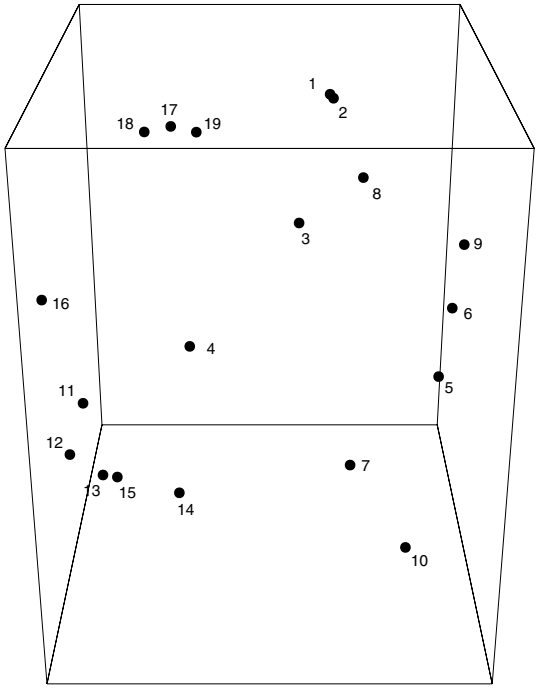


Abb. 3: Visualisierung der Distanzen des Auschnitts aus dem Referenzmodell

Zur Überprüfung unserer Ergebnisse haben wir eine ergänzende Untersuchung mit einem weiteren Werkzeug, Crocodile (vgl. hierzu [LöSi99]), durchgeführt. Das Werkzeug berechnet Distanzen mit der gleichen Formel, die Metriken basieren jedoch auf Mengen von benutzten Methoden. Trotz der scheinbar unterschiedlichen Metrik, waren die Resultate ähnlich wie bei unserer ursprünglichen Untersuchung. Vor allem bei großen, wenig übersichtlichen Objektmodellen lässt eine Clusteranalyse hilfreiche Hinweise zur Identifikation von Subsystemen im allgemeinen, von Komponenten im besonderen erwarten. Abreu und Goulao beschreiben in [AbGo01] positive Erfahrungen mit der automatischen Modularisierung objektorientierter Systeme. Auch Löffler und Simon verweisen in [LöSi00] auf ermutigende

Erfolge, die sie mit ihrem Verfahren erzielt haben.

Trotzdem sind diese Ansätze im Hinblick auf die Ermittlung von Komponenten innerhalb eines vorhandenen Objektmodells kritisch zu betrachten. Zunächst hat die Wahl geeigneter Kopplungsmaße erheblichen Einfluss auf das Resultat. Dabei ist beispielsweise zu fragen, ob ein allein auf statischen Angaben beruhendes Verfahren wie CBO hinreicht oder ob auch die Referenzierungshäufigkeit bei der Laufzeit von Systemen zu berücksichtigen ist - und wie diese ggfs. zu ermitteln ist. Auch die Ergebnisse einer u.U. durchgeführten semi-automatischen Clusteranalyse hängen vom jeweils angewandten Verfahren ab. Überdies muss die Anzahl der möglichen Cluster zu Beginn einer solchen Analyse vorgegeben werden. Auch dies hat einen nicht unwesentlichen Einfluss auf das Ergebnis. Ein weiterer Nachteil des beschriebenen Verfahrens liegt darin begründet, dass Distanzen zwischen Klassen nicht mehr unidirektionale von bidirektionalen Beziehungen differenzieren. In jedem Fall erfordert ein solches Verfahren einen sachkundigen Anwender, der die berechneten Ergebnisse vor dem Hintergrund der jeweiligen Anwendungsdomäne bewertet und ggfs. korrigiert.

4.2 Konzeptionelle Überlegungen zur Identifikation von Komponenten in Objektmodellen

Die Identifikation von Komponentenkandidaten in einem großen Objektmodell ist eine herausfordernde Aufgabe, die ein systematisches Vorgehen empfiehlt. Im Rahmen der hier dargestellten Untersuchung haben wir das folgende Vorgehensmodell verwendet, dessen Schritte in der skizzierten Reihenfolge, ergänzt um zyklische Rückkopplungen zu durchlaufen sind.

1. *Ermittlung von Differenzierungskriterien*: Die Identifikation von Komponentenkandidaten erfordert Kriterien zur Unterteilung des Gesamtmodells. Solche Kriterien sind entweder objektorientierte (nicht im engeren software-technischen Sinn) Abstraktionen über Klassenmengen, wie etwa Dokumente, Lieferdokumente, Produkte, Akteure, Organisationen etc. oder funktionsorientierte, wie z.B. Beschaffung, Kaufanbahnung, Auslieferung, Zahlungsverkehr, etc. Es ist auch denkbar, sowohl objektorientierte als auch funktionsorientierte Abstraktionen zu verwenden. Dabei ist jeweils zu überlegen, ob ein Kriterium geeignet erscheint, eine relativ autonome Mengen von Klassen zu separieren.

2. *Bildung von Teilmodellen*: Dazu wird das Objektmodell unter Verwendung der zuvor ausgewählten Differenzierungsmerkmale in Teilmodelle zerlegt. Falls für einzelne Klassen eine Zuordnung zu einem der Teilmodelle nicht offensichtlich ist, ist entweder unter Rückgriff auf Ähnlichkeiten eine Zuordnung vorzunehmen oder aber man bildet diese Klassen zunächst nicht auf Komponentenkandidaten ab. Die beiden folgenden Schritte können auch parallel ablaufen.

3. *Inhaltliche Evaluation*: Hier ist zu klären, ob es überhaupt sinnvolle Szenarien für den Einsatz der

Komponentenkandidaten gibt. Dazu gehört auch die Untersuchung der Frage, ob der autonome Einsatz einer Komponente vorstellbar ist. Von besonderer Bedeutung ist die Beurteilung der Abstraktion im Hinblick auf ihren Spezialisierungsgrad: Enthält das Teilmodell Besonderheiten, die einer weitreichenden Wiederverwendung im Weg stehen? Falls dies zutrifft, ist zu untersuchen, ob durch eine Überarbeitung des Teilmodells eine Abstraktion von diesen Besonderheiten möglich ist, die aber gleichzeitig eine Anpassung an Besonderheiten durch eine entsprechende Konfiguration zulässt.

4. *Software-technische Analyse und Überarbeitung*: Dazu ist die Kohäsion innerhalb von sowie die Kopplung zwischen den Teilmodellen zu untersuchen. Spezialisierungsbeziehungen zwischen Klassen verschiedener Teilmodelle sind im Einzelfall darauf zu untersuchen, welche Auswirkungen sie für die Pflege der betrachteten Komponenten mit sich bringen (s. Abschnitt 2.3). Dazu sind ggfs. die geerbten Eigenschaften einer betroffenen Klasse zu kopieren. Auch Assoziationen, die die Grenzen von Teilmodellen überschreiten, sind tendenziell zu vermeiden. Das gilt vor allem für bidirektionale Assoziationen und solche unidirektionalen, die eine Referenz auf eine Klasse außerhalb des jeweils betrachteten Teilmodells erfordern. Im Unterschied zu Spezialisierungsbeziehungen können Assoziationen aber nicht immer vermieden werden. Dann ist zu prüfen, ob Referenzen auf externe Objekte nicht durch die Einführung geeigneter Schnittstellenspezifikationen ersetzt werden können. Beispielsweise könnte statt eines Objekts der Klasse "Invoice" eine Instanz eines entsprechenden XML-Dokumenttyps verwendet werden, auf die zur Kommunikation mit einer anderen Komponente entsprechende Objekte abzubilden sind.

5. *Festlegung der Schnittstellen*: Dieser Schritt ist sehr aufwendig und sollte deshalb erst in Angriff genommen werden, nachdem die identifizierten Komponentenkandidaten sorgfältig auf ihre Verwendbarkeit untersucht worden sind. Zunächst sind alle Klassen innerhalb eines Komponentenkandidaten darauf zu überprüfen, ob sie von externen Klassen referenziert werden. Für jede dieser Klassen ist - evtl. mit Hilfe ergänzender Nachrichtenflussdiagramme - zu ermitteln, welche ihrer Dienste von den Objekten der sie referenzierenden Klassen benutzt werden. Anschließend ist jede betroffene Signatur darauf zu überprüfen, ob sie einen speziellen Typ bzw. eine spezielle Klasse enthält. In diesem Fall ist die Implementierung der korrespondierenden Operation so zu ändern, dass die Signatur auf allgemein verfügbare Typen reduziert wird.

Bei der Anwendung dieses Verfahrens auf das vorliegende Referenzmodell haben wir uns für eine Unterteilung des Gesamtmodells entschieden, die im wesentlichen auf objektorientierten (im betriebswirtschaftlichen Sinn) Kriterien (physische Produkte, Finanzprodukte, Organisationen und Akteure) beruht, aber auch auf funktionsorientierte Kriterien (Anbahnung, Zahlungsverkehr, Auslie-

ferung) zurückgreift. In Abb. 4 ist das Resultat näherungsweise dargestellt. Einige der Klassenmengen, die sich als teilautonome Subsysteme und damit als Komponentenkandidaten anbieten, sind durch Umrandung hervorgehoben.

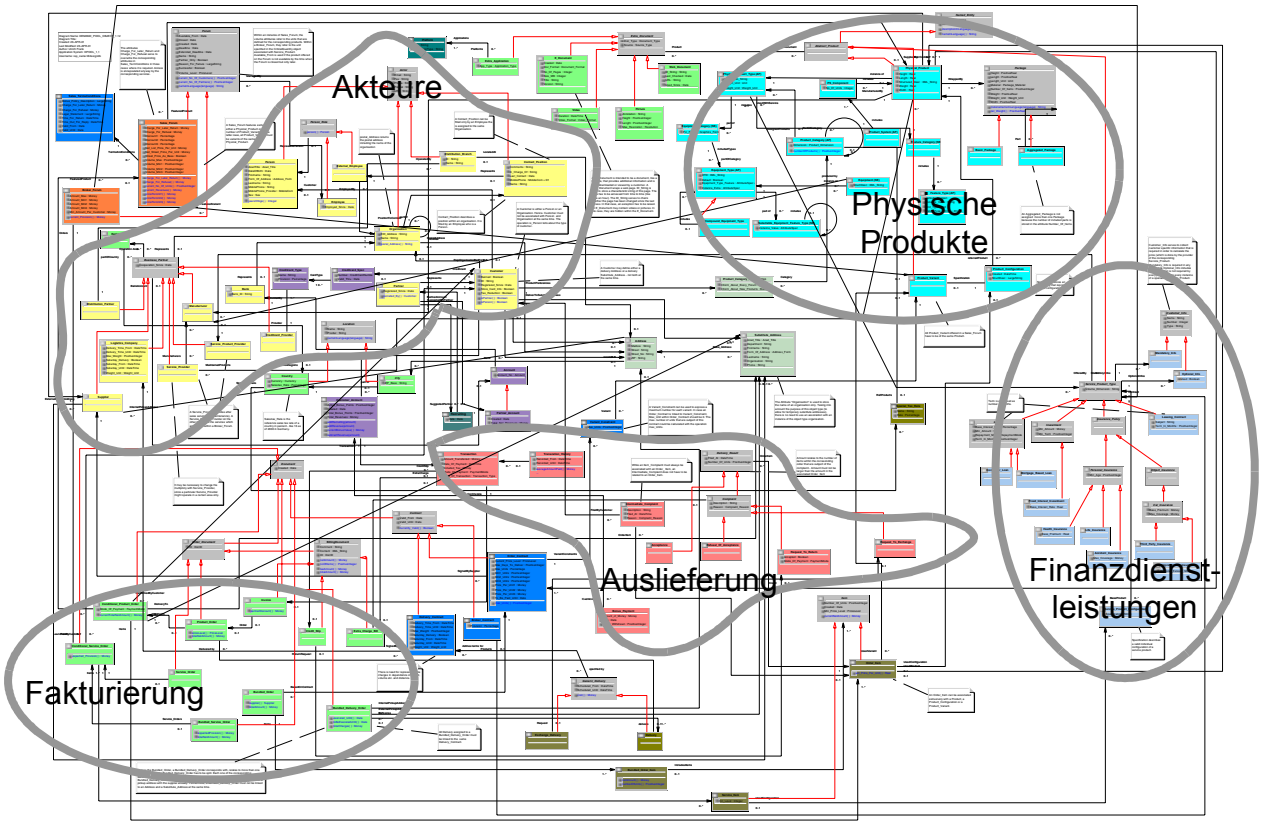


Abb. 4: Identifikation von Komponenten in einem Objektmodell auf der Grundlage konzeptioneller und software-technischer Aspekte

Die Betrachtung des Ergebnisses ist ernüchternd. So sind die Komponenten, die wir auf diese Weise mit einem erheblichen Aufwand ermittelt haben, nicht uneingeschränkt überzeugend. Wir konnten zwar die Kohäsion zwischen den Komponenten beseitigen, eine sinnvolle autonome Verwendung einzelner Komponenten ist aber dennoch kaum möglich: Auch wenn die Schnittstellen keine speziellen Typen enthalten, erfordert ihre Verwendung dennoch häufig das Vorhandensein entsprechender Objekte. Wenn z.B. eine Signatur als Parameter eine Zeichenkette vorsieht, die der Übergabe eines speziellen XML-Dokuments dient, das eine Rechnung abbildet, muß die Software, die diese Komponente verwendet, über die dazu nötigen Daten verfügen. Daneben ist ein anderer, subtilerer Umstand zu berücksichtigen, der das Beurteilung des Verfahrens betrifft: Um ein Modell dieser Größenordnung verstehen zu können, muss man sich bemühen, die verwendeten Begriffe und die damit einhergehenden Abstraktionen nachzuvollziehen. Dadurch wird u.U. der Blick für alternative Abstraktio-

nen getrübt, die die Anforderungen an Komponenten besser erfüllen würden. Der Vergleich dieses Ergebnisses mit dem zuvor in der Clusteranalyse ermittelten zeigt deutliche Unterschiede, die aber durch eine Nachbearbeitung der Cluster auf der Grundlage konzeptioneller Überlegungen reduziert werden können.

4.3 Komponentenorientierter Entwurf?

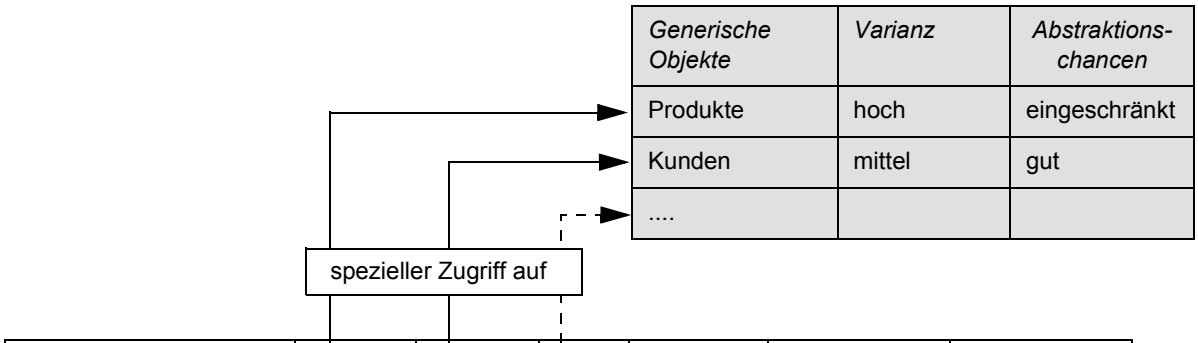
Um der Kritik an dem zuvor dargestellten Verfahren entgegenzuwirken, liegt es nahe, auf den Umweg über ein Objektmodell zu verzichten und die jeweils betrachtete Domäne direkt auf Komponenten abzubilden. Dies fällt dann besonders schwer, wenn man nachhaltig durch objektorientierte Konzepte beeinflusst ist - auch deshalb, weil Komponenten in vielfacher Hinsicht die gleiche Abstraktion darstellen wie Objekte. Wir mussten jedenfalls erkennen, dass eine Feststellung wie "The components are there for the picking." noch problematischer erscheint als das berühmte Originalzitat von Meyer, das sich auf Objekte bezieht ([Mey88], S. 51). Der Versuch, Komponenten auf der Grundlage von Geschäftsprozessmodellen zu identifizieren, erwies sich als nicht tragfähig - vielleicht auch, weil er uns im Zusammenhang mit dem Entwurf von Objektmodellen zu sehr vertraut war. Auch vorhandene Ansätze wie bspw. der Catalysis-Ansatz (vgl. [SoWi99]) liefern hier nur unbefriedigende Resultate. Catalysis zielt auf den Entwurf eines umfangreichen Softwaresystems im Hinblick auf die Entwicklung wiederverwendbarer Komponenten. Dabei steht allerdings die ausführliche Spezifikation von Komponenten im Vordergrund. Ein Vorgehen zur Identifikation von Fachkomponenten wird von Catalysis nicht beschrieben. Die Vorgehensweise wird lediglich anhand einiger Beispiele erörtert. Unterstützt wird der Prozeß durch die Formulierung einiger Muster, wobei nur wenige auf den komponentenorientierten Entwurf eingehen. Auch wird die isolierte Entwicklung von Komponenten im Hinblick auf domänenunabhängige Wiederverwendung wenig beachtet (Verletzung der Unabhängigkeit vom Wiederverwendungskontext).

Im Rahmen unserer Untersuchung haben wir uns für eine Heuristik entschieden, die auf ökonomischen und organisatorischen Überlegungen basiert.

1. *Generische Domänenbeschreibung*: Hier handelt es sich um eine natürlichsprachliche Beschreibung einer Abstraktion über die Domänen, für die die zu entwickelnden Komponenten gedacht sind. Letztlich sollte die Domänenbeschreibung auch eine Liste wesentlicher betriebswirtschaftlicher Funktionen und Objekte ergeben (*generische* Funktionen und Objekte). Eine heuristische Orientierung zur Erstellung einer solchen Liste liefern geeignete Fragen, wie etwa: "Welche Funktionen sind wesentlich für die Durchführung der Kernprozesse" und anschließend: "Welche dieser Funktionen sind auch in anderen Unternehmen der Branche obligatorisch?".
2. *Vertragsorientierte Arbeitsteilung*: In diesem Schritt sind die zuvor ermittelten betriebswirtschaft-

lichen Funktionen darauf zu untersuchen, ob ihre Konkretisierungen in unterschiedlichen Kontexten variieren und wie die Chance ist, ggfs. tragfähige Abstraktionen über diese Varianz finden zu können. Diese Betrachtung ist auch für die jeweils genutzten generischen Objekte durchzuführen. Im Hinblick auf die Anforderung nach weitgehender Autonomie einer Komponente ist zudem zu beurteilen, wie groß jeweils der Bedarf ist, mit anderen Funktionsbereichen zu kommunizieren bzw. die generischen Objekte zu nutzen. Für jeden der so identifizierten Funktionsbereiche ist ein Vertrag zu formulieren, der beispielsweise die folgende Struktur haben könnte: Leistungsangebot des Bereichs, Voraussetzungen für Leistungsanforderung (Vorbedingung), Zusicherungen für die erbrachten Leistungen (Nachbedingungen). Die Struktur eines solchen Vertrags ist durch entsprechende Ansätze zur Spezifikation von Software-Artefakten ("design by contract", [Mey97]) motiviert. In der Analysephase ist eine natürlichsprachliche Beschreibung eines Vertrags hinreichend. Wenn nach dem ggfs. zyklischen Durchlaufen dieser beiden Schritte eine Menge von Komponentenkandidaten vorliegt, ist jeder einzelne Kandidat weiter zu evaluieren und ggfs. zu konkretisieren, wozu sich die in 4.2 dargestellten Schritte 3 bis 5 anbieten. Anschließend sind die Komponenten zu entwerfen, wozu sich ein objektorientierter Ansatz anbietet.

Abb. 5 zeigt einen Teil der im zweiten Schritt zu erstellenden Differenzierung. Für die ermittelten generischen Objekte und Funktionen sind jeweils die zu erwartende Varianz und die Chancen, zu tragfähigen Abstraktionen zu gelangen, angegeben. Daneben ist für jede generische Funktion zu erfassen, ob sie einen speziellen Zugriff auf generische Objekte erfordert. Ein Zugriff ist dann speziell, wenn er auch auf spezielle Merkmale des Objekts gerichtet ist. So werden u.U. im Einzelfall für den Vertragsabschluss spezielle Integritätsbedingungen für die Konfiguration eines Produkts benötigt, die nicht generalisierbar sind. Ergänzend ist für jede generische Funktion angegeben, wie groß der Bedarf an einer differenzierten Kommunikation mit anderen Funktionen ist. Kommunikation ist umso differenzierter, je spezieller die Semantik der referenzierten Konzepte (Typen, Klassen, Funktionen) ist. Die auf diese Weise in dem dargestellten Beispiel ausgewählten generischen Funktionen sind erste Kandidaten für den Entwurf von Komponenten. Besonders gut geeignet ist danach die generische Funktion "Bonitätsprüfung", da sie keinen Zugriff auf Produkte erfordert und keinen speziellen Zugriff auf Kunden. Ebenso ist keine differenzierte Kommunikation mit anderen generischen Funktionen nötig. Da es sich um ein weitgehend standardisiertes Procedere handelt, sind die Chancen, zu einer tragfähigen Abstraktion zu gelangen, sehr gut. Da es sich hierbei zunächst um eine funktionsorientierte Abstraktion handelt und Komponenten eine objektorientierte Abstraktion empfehlen, wäre eine geeignete Komponente, etwa "Bonitätsprüfer", zu erstellen, die dann die Funktionalität der Bonitätsprüfung anbieten würde.



Generische Funktionen				*Varianz*	*Abstraktions-chancen*	*Kommunika-tionsbedarf*
Vertragsanbahnung	ja	nein		mittel	gut	gering
Preisbildung	ja	nein		hoch	gut	mittel
Bonitätsprüfung	nein	nein		gering	gut	gering
Vertragsabschluss	ja	ja		hoch	mittel	mittel
Auslieferung	nein	ja		mittel	gut	mittel
Zahlungsverkehr	nein	ja		mittel	gut	mittel
Finanzbuchhaltung	nein	nein		mittel	gut	hoch
.....						

Abb. 5: Strukturierte Beschreibung generische Funktionen

Die Beurteilung dieses Vorgehens gestaltet sich schwierig. Einerseits scheint es für die Identifikation von Komponenten erfolgversprechender zu sein, von Beginn an auf entsprechende inhaltliche Abstraktionen zu fokussieren, anstatt sie später "bottom up" aus Objektmodellen zu rekonstruieren. Schließlich weichen die spezifischen Zielsetzungen der Komponentenentwicklung in einigen Punkten (vor allem: Kopplung) von denen des objektorientierten Entwurfs ab. Andererseits ist zu berücksichtigen, dass die Implementierung von Komponenten letztlich ohnehin eine detaillierte Sichtweise erfordert. Wenn man dabei die Wiederverwendung der dazu verwendeten Klassen in mehreren Komponenten anstrebt, ist es angeraten, komponentenübergreifende Objektmodelle zu erstellen.

5. Abschließende Bemerkungen

Die mittels einer Fallstudie illustrierte Untersuchung der Frage, wie Fachkomponenten zu entwerfen sind, führt zu einem ernüchternden Ergebnis. Das gilt zum einen für die Bewertung der vorgestellten prototypischen Verfahren. Die in der Fallstudie gesammelten Erfahrungen legen die Vermutung nahe, dass es nicht angemessen ist, eine der drei Vorgehensweisen isoliert anzuwenden. Gleichzeitig ist die Beantwortung der Frage, ob Komponenten aus Objektmodellen rekonstruiert werden sollten

oder als unmittelbare Entwurfsabstraktionen verwendet werden sollten, ohne weitere Untersuchungen kaum möglich. Dabei ist auch zu berücksichtigen, dass die Qualität des Ergebnisses jeweils von der Abstraktionsfähigkeit der Entwickler abhängt und u.U. nur zu einem geringeren Teil durch das gewählte Verfahren beeinflusst wird. Vor allem aber nährt unsere Untersuchung grundsätzliche Zweifel an der Angemessenheit gängiger Anforderungen an Fachkomponenten. Das gilt vor allem für die Forderung nach autonomer Verwendbarkeit einzelner Fachkomponenten auch in unvorhergesehenen Kontexten. Vordergründig scheint diese Forderung bei anwendungsunabhängigen Komponenten wie solchen zur Gestaltung von graphischen Benutzungsschnittstellen erfüllt. Sie können eben für beliebige Anwendungen genutzt werden. Bei näherer Betrachtung ist diese Feststellung aber zu relativieren: Auch diese Komponenten erfordern eine Umgebung, die gewissen Konventionen genügt, etwa in Form eines bestimmten Kommunikationsmodells oder im Hinblick auf die Bereitstellung von Ressourcen. Daneben gilt, dass das für eine Wiederverwendung nötige Verständnis einer Komponente auch durch die jeweils möglichen Einsatzformen festgelegt wird. Das Verständnis von Fachkomponenten - wir könnten auch sagen: von Fachbegriffen - erfordert in der Regel deren Beziehung zu anderen Begriffen in der jeweiligen Domäne.[1] Dabei handelt es sich per definitionem um speziellere Begriffe als bei allgemein einsetzbaren Komponenten. Anders als in der natürlichen Sprache mit ihrer ausgeprägten Toleranz für Mehrdeutigkeiten, müssen die Schnittstellen von Fachkomponenten allerdings eindeutig beschrieben sein, was letztlich nur dann sinnvoll denkbar ist, wenn ihre Entwicklung auf der Basis eines geeigneten semantischen Referenzsystems - etwa eines Objektmodells - stattfindet.

Vor diesem Hintergrund spricht einiges für einen Ansatz, der nicht auf die isolierte Wiederverwendung von Fachkomponenten gerichtet ist. Demgegenüber bietet der Einsatz fachlicher Frameworks eine reizvolle Perspektive für die Verwendung von Komponenten: Der Verwender entsprechender Komponenten wird von dem erheblichen Aufwand befreit, selbst eine Anwendungsarchitektur zu entwickeln, die Komposition von Komponenten bzw. das Einfügen einzelner Komponenten wird durch das Framework erheblich erleichtert. Zur Entwicklung solcher Frameworks bietet sich ein objektorientierter Ansatz an. Dabei stellt sich die Frage, ob man dann überhaupt noch Komponenten benötigt. So gibt es etwa in einem von IBM angebotenen Framework für Finanzdienstleister ([MCD99]) neben einer Reihe von Subsystemen, die Kernprozesse abbilden, Komponenten ("Common Business Objects"), deren Granularität mit der von Klassen vergleichbar ist. Der Vorteil von Komponenten ist im wesentlichen darin zu sehen, dass sie - eine entsprechende Infrastruktur vorausgesetzt - leichter ausgetauscht werden können als beliebige Klassen, die u.U. eine enge Kopplung zu

1. Das gilt in ähnlicher Weise für die Worte einer Sprache: "Die Bedeutung eines Wortes ist sein Gebrauch in der Sprache." ([Witt80], 43)

anderen Klassen aufweisen. Auf diese Weise wird es denkbar, dass die Spezifikationen von Komponenten, die für bestimmte Frameworks einsetzbar sind, einen Wettbewerb zwischen den Anbietern entsprechender Komponenten anregt, der dazu beiträgt, dass die Anwender im Sinne eines "best of breed" die jeweils am besten geeignete Komponente erwerben können. Es bleibt die Frage zu analysieren, wie allgemein oder speziell ein Framework konzipiert sein sollte. Jenseits wirtschaftlicher Erwägungen im Einzelfall gibt es wohl keine allgemeine Antwort. Aus wissenschaftlicher Sicht bietet sich ein evolutionärer Ansatz an: Man beginnt mit der Entwicklung relativ spezieller Objektmodelle für verschiedene Bereiche. Anschließend werden Ähnlichkeiten genutzt, um gemeinsame Teile zu identifizieren. So kann nach und nach ein mehrschichtiges System von Referenzmodellen und korrespondierenden Frameworks entstehen.

Literatur

[AbGo01] Abreu, F. B.; Goulao, M.: Coupling and Cohesion as Modularization Drivers: Are we being over-persuaded? Erscheint in: Proceedings of the 5th European Conference on Software Maintenance and Reengineering, Lisbon, 2001

[AlFr98] Allen, P.; Frost, S.: Component-Based Development for Enterprise Systems: Applying the SELECT Perspective. Cambridge University Press 1998

[Balz00] Balzert, H.: Lehrbuch der Softwaretechnik 1. 2. Auflage, Heidelberg: dpunkt 2000

[BDH+98] Broy, M.; Deimel, A.; Henn, J.; Koskimies, K.; Plasil, F.; Pomberger, G.; Pree, W.; Stal, M.; Szyperski, C. (1998): What characterizes a (software) component? In: Software - Concepts & Tools, 19. Jahrgang, Heft 0, S. 49-56

[Fran00a] Frank, U.: Die Modellierung von Produkten für Handelsplattformen im Internet: Ein Ansatz auf der Basis von Metakonzepten. In: Jasper, H.; Küng, J.; Vossen, G. (Hg.): Informationssysteme für E-Commerce: EMISA-2000. Linz: Universitätsverlag Rudolf Trauner 2000, S. 169-185

[Fran00b] Frank, U.: Entwurf eines Referenzmodells für Handelsplattformen im Internet. In: Tagungsband der Fachtagung KnowTech, Leipzig 2000

[Grif98] Griffel, F.: Componentware: Konzepte und Techniken eines Softwareparadigmas. Heidelberg: dpunkt-Verlag

[Hend96] Henderson-Sellers, B.: Object-Oriented Metrics. Upper Saddle River, New Jersey: Prentice Hall

[HeSi01] Herzum, P.; Sims, O.: Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. New York: Wiley 2001

[LöSi00] Löffler, S.; Simon, F.: Semiautomatische, kohäsionsbasierte Subsystembildung. In: Dumke, R.; Lehner, F. (Hg.): Software-Metriken: Entwicklungen, Werkzeuge und Anwendungsverfahren. Wiesbaden: Deutscher Universitäts-Verlag, S. 153-170

[MCD99] Monday, P.; Carey, J.; Dangler, M.: San Francisco Component Framework: An Introduction. Reading, Mass., et al.: Addison-Wesley 1999

[Meye88] Meyer, B.: Object-oriented Software Construction. 1. Auflage, Upper Saddle River, NJ: Prentice Hall 1988

[Meye97] Meyer, B.: Object-oriented Software Construction. 2. Auflage, Upper Saddle River, NJ:

Prentice Hall 1997

[NiDa95] Nierstrasz, O.; Dami, L.: Component-Oriented Software Technology. In: Nierstrasz, O.; Tsichritzis, D. C. (Hg.): Object-Oriented Software Composition. London; New York; Toronto; Sydney; Tokyo; Singapore; Madrid; Mexico City; Munich: Prentice Hall, S. 1-28

[OMG96] Object Management Group: Common Facilities RFP-4: Common Business Objects and Business Object Facility. OMG, TC 13CF/96-01-04, 1996

[OHE96] Orfali, R.; Harkey, D.; Edwards, J.: The Essential Distributed Objects Survival Guide. New York: Wiley 1996

[Pree97] Pree, W.: Komponentenbasierte Softwareentwicklung mit Frameworks. dpunkt Verlag 1997

[RTF99] Rautenstrauch, C.; Turowski, K.; Fellner, K. K.: Fachkomponenten zur Gestaltung betrieblicher Anwendungssysteme. In: Information Management & Consulting,14.Jg., Nr. 2, 1999, S. 25-34

[SoWi99] D'Souza, D.F.; Wills, A.C.: Objects, Components and Frameworks with UML: The Catalysis Approach. Boston et al.: Addison-Wesley 1999

[Spro00] Sprott, D.: Componentizing the Enterprise Application Packages. In: Communication of the ACM, Vol. 43, No. 4, 2000, pp. 63-69

[SSL00] Simon, F.; Steinbrückner, F.; Lewerentz, C.: 3D-Spring Embedder for Complete Graphs. Computer Science Reports, Technical University Cottbus, September 2000

[Szyp97] Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Reading, Mass. et al.: Addison-Wesley 1997

[Witt80] Wittgenstein, L.: Philosophische Untersuchungen. 2. Auflage, Frankfurt/M.: Suhrkamp Taschenbuch 1980