Komponentenfindung in monolithischen objektorientierten Anwendungssystemen

Johannes Maria Zaha, Stephan Kelch

Lehrstuhl Wirtschaftsinformatik und Systems Engineering
Universität Augsburg, Universitätsstrasse 16
86159 Augsburg
johannes.maria.zaha@wiwi.uni-augsburg.de
s.kelch@gmx.de
Tel: +49(821)598-4431, Fax:-4432
http://wi-se.wiwi.uni-augsburg.de

Zusammenfassung: Durch das Redesign einer objektorientiert implementierten Anwendung vom Monolithen hin zu einer komponentenorientierten Architektur erschließen sich zunächst Vorteile wie leichtere Erweiterbarkeit und Wartbarkeit, in folgenden Softwareentwicklungsprojekten evtl. die Wiederverwendung von einzelnen Komponenten. In diesem Betrag wird ein Lösungsalgorithmus vorgestellt, mit dem sowohl Komponenten als auch Frameworks in einem existierenden monolithischen, objektorientiert implementierten Anwendungssystem identifiziert werden können. Dieser Lösungsalgorithmus stellt eine Spezialisierung eines allgemeinen Ansatzes der Autoren zur Komponentenfindung dar, bei dessen Anwendung in einem Praxisprojekt die Verbesserungspotentiale in Bezug auf objektorientierte Anwendungssysteme erarbeitet wurden. Aufbauend auf den dort gewonnen Erfahrungen wurde der Lösungsalgorithmus erweitert, um die Charakteristika der Objektorientierung zu berücksichtigen.

Schlüsselworte: Objektorientierte monolithische Anwendungssysteme, Software-Komponente, Komponentenfindung.

1 Einleitung

Die Idee, Anwendungssysteme aus wieder verwendbaren Softwarebausteinen zu erstellen, wird wenigstens seit der Publikation von McIllroy im Jahre 1968 [Mc68] verfolgt. Seitdem wurden mehrere Techniken zur Wiederverwendung von Software-Artefakten, wie etwa Code- and Design Scavenging [Sa97] oder Generative Techniken [CE00], entwickelt. Kompositorische Wiederverwendung soll die Vorteile von Standard- und Individualsoftware durch eine Plug-and-Play-Wiederverwendung von Black-Box-Komponenten verbinden. Diese Komponenten sollen auf Komponentenmärkten gehandelt werden, wodurch der Aufwand für die Erstellung von Anwendungssystemen verringert und die Qualität erhöht wird. Für die Komponentenfindung in existierenden Anwendungssystemen stehen neben diesen beiden allgemeinen Gründen, die für eine

komponentenbasierte Architektur sprechen, vor allem die verbesserte Wartbarkeit und Erweiterbarkeit der Software im Vordergrund. Diese beiden Vorteile sind bei komponentenorientierten Ansätzen dadurch gegeben, dass die einzelnen Systemteile nicht mehr wie in monolithischen Systemen eng miteinander vermascht sind und dadurch Änderungen keine Auswirkungen auf andere Systemteile haben, wodurch die erforderlichen Testanstrengungen verringert werden. Gleichzeitig steigt jedoch der Aufwand bei der komponentenorientierten Entwicklung in den ersten Phasen eines Softwareentwicklungsprojektes, da der Spezifikations- und Koordinationsaufwand erhöht ist. Da der größte Anteil der Kosten eines Softwaresystems aber in den Phasen nach Abschluss der Implementierung entsteht (vgl. [Li98] (und damit dort auch die größten Einsparungspotentiale liegen), ist aus Kostengesichtspunkten auch dann die Entwicklung auf Basis einer komponentenbasierten Architektur sinnvoll, wenn eine marktliche Verwendung der Komponenten nicht angestrebt wird.

Die Trennung einzelner Systemteile ist bei kompositorischer Wiederverwendung per Definition festgelegt. Nach [Tu03] ist eine Komponente wie folgt definiert:

> Eine *Komponente* besteht aus verschiedenartigen (Software-)Artefakten. Sie ist wiederverwendbar, abgeschlossen und vermarktbar, stellt Dienste über wohldefinierte Schnittstellen zur Verfügung, verbirgt ihre Realisierung und kann in Kombination mit anderen Komponenten eingesetzt werden, die zur Zeit der Entwicklung nicht unbedingt vorhersehbar ist.

Das dabei verfolgte Paradigma der Unterteilung eines komplexen Problems in leichter zu lösende Teilprobleme wird durch andere Strukturierungen der Architektur von Software-Systemen ebenfalls verfolgt. Prominentester Vertreter dabei ist die Gliederung in die Schichten Benutzeroberfläche, Anwendung und Datenhaltung die zu einer Drei-Schichten-Architektur führt (vgl. [Ba96], S. 640ff). Für die Identifikation von Schichten existieren Vorgehensmodelle, allerdings widerspricht dieses Prinzip dem kompositorischen Widerverwendungsparadigma, da dabei Komponenten für unterschiedliche Schichten entwickelt werden und damit die Freiheitsgrade bei der Wiederverwendung stark eingeschränkt sind. Prinzipiell kann der hier vorgeschlagene Lösungsalgorithmus auch auf einzelne Schichten analog angewendet werden und innerhalb der jeweiligen Schicht zur Komponentenfindung eingesetzt werden. Der Einfachheit halber wird hier eine Anwendung unter Vernachlässigung der Schichten gezeigt.

Das Leitbild der kompositorischen Wiederverwendung für betriebliche Anwendungssysteme – welches im Weiteren betrachtet wird – basiert auf der Untergliederung der Anwendungsfunktionalität in sog. Fachkomponenten (vgl. [Tu03], S. 19).

> Eine *Fachkomponente (FK)* ist eine Komponente, die eine bestimmte Menge von Diensten einer *betrieblichen* Anwendungsdomäne anbietet.

Um die Wiederverwendung von Fachkomponenten sicherzustellen, werden zusätzliche Systemteile benötigt, sog. Komponenten-Anwendungs-Frameworks und Komponenten-System-Frameworks [vgl. Tu03, S. 39f]. Unter einem *Komponenten-Anwendungs-Framework* wird ein Systemteil verstanden, der Fachkomponenten anwendungsdomänenbezogene (Standard-)Dienste bereitstellt und für diese eine Integrationsplattform darstellt. Darüber hinaus werden von sog. *Komponenten-System-Frameworks* anwen-

dungsinvariante, middlewarenahe Dienste zur Verfügung gestellt. Beispiele für Komponenten-System-Frameworks sind Datenbank-Management-Systeme (DBMS) oder Workflow-Management- Systeme (WFMS).

Um nun existierende Anwendungsysteme in architektonischer Hinsicht einem Reengineering zu unterziehen, existieren Vorgehensmodelle (vgl. z.B. [Fo00] oder [BA04]. Allerdings sind diese sehr allgemein gehalten und eignen sich nicht für die Komponentenfindung in objektorientierten Anwendungssystemen. [AL03] schlagen ein Vorgehensmodell für die Komponentenfindung vor, gehen dabei allerdings von einer Neuentwicklung aus und berücksichtigen deshalb auch nicht spezifische Charakteristika von objektorientierten Anwendungssystemen.

Da existierende Ansätze für den Einsatz zur Komponentenfindung in monolithischen objektorientierten Anwendungssystemen nicht geeignet sind, wird im Folgenden ein Lösungsalgorithmus als Basis herangezogen, der von den Autoren bereits vorgeschlagen wurde [KZ03]. Allerdings ist dieser Lösungsalgorithmus allgemein gehalten, um sowohl die Gegebenheiten in prozedural als auch objektorientiert implementierten Systemen zu berücksichtigen. Beim Versuch, diesen Lösungsalgorithmus in einem Praxisprojekt anzuwenden, hat sich gezeigt, dass er dafür weiter spezialisiert werden muss. Dieser spezialisierte Algorithmus als methodisches Ergebnis des Praxisprojektes – einer Architekturoptimierung einer webbasierten J2EE-Applikation – wird in dieser Arbeit vorgestellt.

2 Objektorientierung und Komponentenfindung

Die Komponentenfindung zielt auf einen Umbau der Architektur eines monolithischen Anwendungssystems ab. Dadurch bedingt müssen die jeweiligen relevanten Grundprinzipien der einzelnen Programmierparadigmen bei einer Anwendung des Algorithmus miteinbezogen werden. Im existierenden Lösungsalgorithmus [KZ03] wurden die speziellen Mechanismen zu Gunsten der Allgemeingültigkeit vernachlässigt. Bezüglich der Objektorientierung sind Mechanismen bzw. Konstrukte relevant, die sich auf die Objektstrukturen und -abhängigkeiten auswirken.

Die Objektorientierung [Ba99] baut, wie der Name impliziert, auf einer Betrachtung von Software in Form von Objekten auf. Die Konstruktion bzw. die Beschreibung der Objekte erfolgt in Form von Klassen, die den Bauplan für die einzelnen Objekte darstellen. Eine Klasse ist der abstrakte Bauplan, ein Objekt ist eine konkrete Instanz einer Klasse. Das Objekt kann Operationen ausführen, die innerhalb der Klasse in Form von Methoden definiert wurden. Mit Hilfe des Mechanismus der Vererbung ermöglicht die Objektorientierung, Methodendefinitionen innerhalb einer Klasse an weitere Klassen zu vererben. Eine Klasse aus der sich andere Klassen ableiten, wird als Superklasse bezeichnet, eine Klasse die sich aus einer Superklasse ableitet wird als Subklasse der jeweiligen Superklasse bezeichnet. Innerhalb der Subklasse sind alle Definitionen der Superklasse verfügbar und können dort erweitert oder überschrieben werden. Durch den Mechanismus der Vererbung kommt es zu Abhängigkeiten innerhalb der Anwendung die ähnlich einem Stammbaum aufgebaut sind.

Für die Komponentenfindung ergeben sich bei der Durchführung des existierenden Lösungsalgorithmus [KZ03], angewandt auf ein objektorientiert implementiertes, monolithisches Anwendungssystem, zwei Probleme. Beide Probleme resultieren aus der Vernachlässigung einiger Grundprinzipien der Objektorientierung.

Das erste Problem ergibt sich aus der Abgrenzung Methode und Klasse. In diesem Lösungsalgorithmus wird eine elementarfunktionale Zuordnung auf Methodenebene vorgeschlagen und der Objekt- bzw. Klassenzusammenhang vernachlässigt. Die Extraktion der Methoden aus der jeweiligen Klasse ist jedoch so nicht auf die Objektorientierung anwendbar. Die Klasse stellt die Struktur bereit, aus der Objekte abgeleitet werden. Die Objekte stellen die in der Klasse definierten Methoden bereit. Auf Methoden kann nur zugegriffen werden, wenn eine Instanz der Klasse, also ein Objekt, erzeugt wurde. Dies bedeutet, dass die Methoden zwar einen Anteil zur Erfüllung der Elementarfunktion beitragen, die Elementarfunktion aber vom Objekt erfüllt wird. Eine elementarfunktionale Zuordnung von Methoden ist daher nicht möglich und muss auf Klassenebene vorgenommen werden.

Das zweite Problem resultiert aus den Objektstrukturen innerhalb einer objektorientierten Anwendung. In objektorientierten Systemen existieren grundsätzlich zwei Strukturen. Durch die Objektorientierung und den Einsatz der objektorientierten Mechanismen entsteht innerhalb der Anwendung eine hierarchische Struktur der Klassen. Einige Klassen dienen nur als Superklassen aus denen sich die weiteren Klassen ableiten. Die zweite Struktur innerhalb der Anwendung ergibt sich aus der grundsätzlichen Aufgabe der Anwendung. Prinzipiell dient eine Anwendung der Erfüllung spezifizierter Funktionen. Der existierende Algorithmus zur Komponentenfindung berücksichtigt nur die funktionale Struktur und vernachlässigt die Objekthierarchien. Eine rein elementarfunktionale Zuordnung der Klassen, die in der Objekthierarchie ganz unten stehen und aus denen sich keine weiteren Klassen mehr ableiten, ist ohne Probleme möglich. Bei der Zuordnung der Klassen, die nur als Überbau für weitere Klassen dienen, kommt es zu einer Kollision der beiden Anwendungsstrukturen. Die Problematik wird an einem Beispiel verdeutlicht, das in Abbildung 2.1 dargestellt ist. Eine fiktive Funktion *Job* lässt sich in zwei Elementarfunktionen, *Job anlegen* und *Job anzeigen*, untergliedern. Den beiden Elementarfunktionen können die beiden Klassen *submitJob* und *showJob* zugeordnet werden, die sich beide aus einer Superklasse *superJob* ableiten.

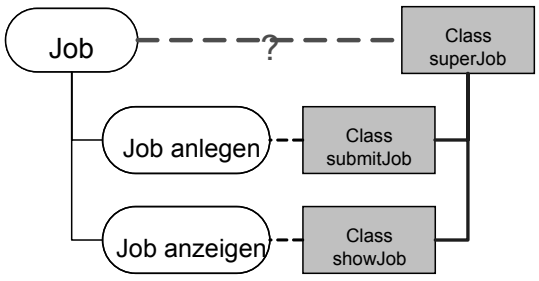

Abb. 2.1: Darstellung der Superklassenproblematik

Der existierende Algorithmus sieht nur eine elementarfunktionale Zuordnung vor. Falls

keine elementarfunktionale Zuordnung möglich ist, werden die entsprechenden Methoden neben dem Diagramm notiert und im Arbeitsschritt 3 dem Komponenten-System-Framework zugeordnet. Die Klasse *superJob* kann keiner Elementarfunktion zugeordnet werden, da sie lediglich ein Rahmengerüst für die beiden anderen Klassen darstellt. Eine derartige Zuordnung der Superklasse *superJob* wäre jedoch nicht korrekt, da das Komponenten-System-Framework nur anwendungsinvariante, middlewarenahe Dienste zur Verfügung stellt. Logisch wäre eine Zuordnung der Superklasse zur Funktion *Job*, da sie nur für die Bereitstellung der Elementarfunktionen dieser Funktion benötigt wird.

Da der bestehende Algorithmus diese beiden Problematiken nicht berücksichtigt, sind hier Anpassungen vor einem Einsatz für objektorientierten Anwendungen zwingend erforderlich.

3 Lösungsalgorithmus

Der existierende Algorithmus wird im Folgenden an die Objektorientierung angepasst, um die beiden Probleme bei der Durchführung zu beheben. Das Problem der Behandlung der Klassen und Methoden kann sehr einfach behoben werden. Die Zuordnung erfolgt lediglich auf Klassen- und nicht mehr wie ursprünglich vorgesehen auf Methodenebene. Die Berücksichtigung der Objektstrukturen erfordert Anpassungen an den einzelnen Arbeitsschritten.

Schritt 1: Funktionale Zerlegung

Durch die funktionale Zerlegung des Systems sollen die einzelnen Funktionen des Systems identifiziert und in Elementarfunktionen unterteilt werden, um so eine Basis für die zukünftige Architektur zu schaffen. Die Auflösung der Funktionalität eines Softwaresystems in Funktionen ermöglicht es, kleine Einheiten für die Schaffung von Komponenten zu finden. Die Elementarfunktionen stellen die angebotenen Dienste der späteren Komponente dar. Die erzielten Ergebnisse werden in einem Funktionsdekompositionsdiagramm (bzw. Funktionsbaums, vgl. [Sc91], S. 65) dargestellt (vgl. Abb. 3.1).

Um die Superklassen in die Zuordnung der Klassen in Schritt 3 mit einzubinden und damit deren Zugehörigkeit zu einzelnen Komponenten oder den Frameworks auszudrücken, werden das Gesamtsystem bzw. die einzelnen Funktionen jeweils um einen Superklassencontainer (SK-Container) erweitert.

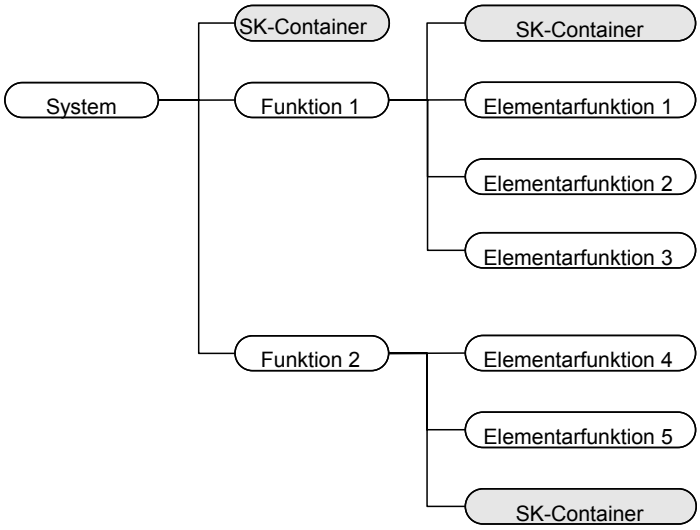

Abb. 3.1: Funktionsdekompositionsdiagramm

Schritt 2: Isolieren der Superklassen und Extraktion der Objekthierarchie/-struktur

Im zweiten Schritt werden die Objekthierarchien und -strukturen des Systems abgebildet. Die Abhängigkeiten der Klassen untereinander werden in einem Stammbaum abgebildet. Begonnen wird mit den Superklassen, denen aufgrund der Vererbungsbeziehungen bei der Komponentenfindung in objektorienterten Systemen eine besondere Bedeutung zukommt. Eine funktionale Zuordnung ex-ante ist nicht möglich, da die Superklassen nicht in einem rein elementarfunktionalen Zusammenhang gesehen werden. Die Superklassen werden aus der Anwendung herausgelöst und mögliche Vererbungshierarchien der Superklassen untereinander in Form eines Baumes Bereich dargestellt. Mit Hilfe dieser Darstellungsweise werden auch die Beziehungen der Superklassen untereinander erkannt. Sind alle Superklassen im Stammbaum abgebildet werden die restlichen Klassen in den Stammbaum aufgenommen und, falls nötig, mit der jeweilige Superklasse verbunden. Als Ergebnis erhält man eine vollständige Abbildung der Klassenabhängigkeiten im System (vgl. Abb. 3.2).

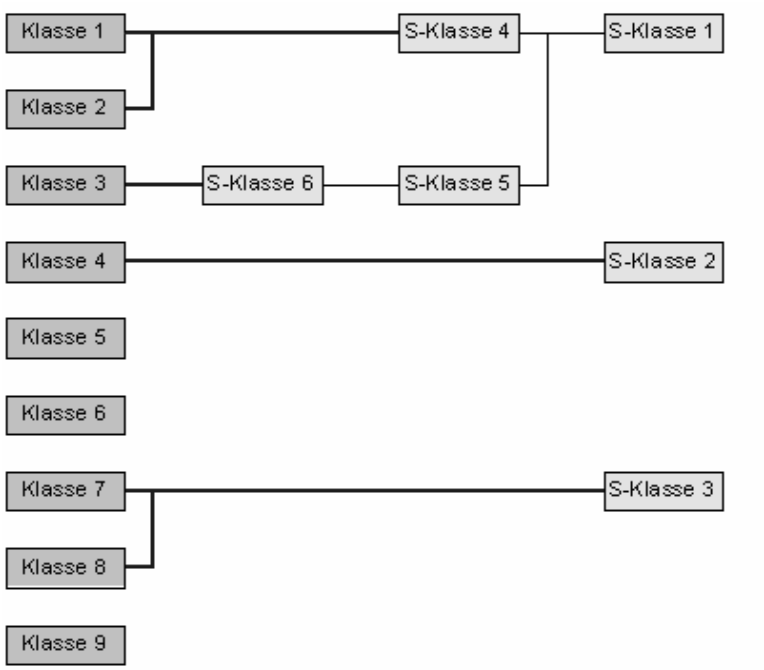

Abb. 3.2: Abbildung der Objektstrukturen

Die Informationen zur Durchführung der Isolation der Superklassen und zur Durchführung der Extraktion der die Objekthierarchien und -strukturen des Systems sollten aus der Spezifikation der Anwendung gewonnen werden. Liegt diese nicht oder in nicht ausreichender detaillierter Form vor, ist eine Untersuchung des Codes mit anschließender nachträglicher Dokumentation unabdingbar. Vor diesem Identifikationsschritt muss, falls die grundlegenden Prinzipien des Software Engineerings nicht befolgt wurden, noch ein Refactoring der implementierten Lösung gestellt werden (vgl. z.B. [Fo00]).

Schritt 3: Funktionale Zuordnung der Klassen

Nach der Gegenüberstellung der funktionalen Struktur und der Objektstruktur werden die Klassen den Elementarfunktionen zugeordnet, die sie erfüllen. Die Zuordnung erfolgt durch eine Verbindung der jeweiligen Klasse mit den jeweiligen Elementarfunktionen. Die Superklassen können an dieser Stelle noch nicht elementarfunktional zugeordnet werden. Bei der Zuordnung ergeben sich, je nach Aufgabe der Klasse, verschiedene Möglichkeiten. Es können folgende drei Kategorien unterschieden werden:

- *Eindeutig elementarfunktionaler Bezug:* Eine Klasse kann direkt einer Elementarfunktion zugeordnet werden, da sie nur von einer einzigen Elementarfunktion benutzt wird. Ein Beispiel dafür stellt die *Klasse 1* in Abbildung 3.3 dar, die lediglich von *Elementarfunktion 1* benutzt wird.

- *Kein eindeutiger elementarfunktionaler Bezug:* Darunter fallen Klassen, die für die Erfüllung verschiedener Elementarfunktionen genutzt werden. Diese Kategorie besitzt zwei Unterkategorien: zum einen Klassen, die zwar mehreren Elementarfunktionen zugeordnet sind, diese Elementarfunktionen aber nur einer Funktion zugeordnet sind (Beispiel: *Klasse 3* in Abbildung 3.3). Zum anderen Klassen, die mehreren Elementarfunktionen zugeordnet sind, die wiederum mehreren Funktionen zugeordnet sind (Beispiel: *Klasse 4* in Abbildung 3.3).

- *Ohne elementarfunktionalen Bezug:* Klassen, die sich keiner Elementarfunktion zugeordnet werden können, werden neben dem Diagramm notiert. Ein Beispiel dafür sind die Klassen 7, 8 und 9 in Abbildung 3.3.

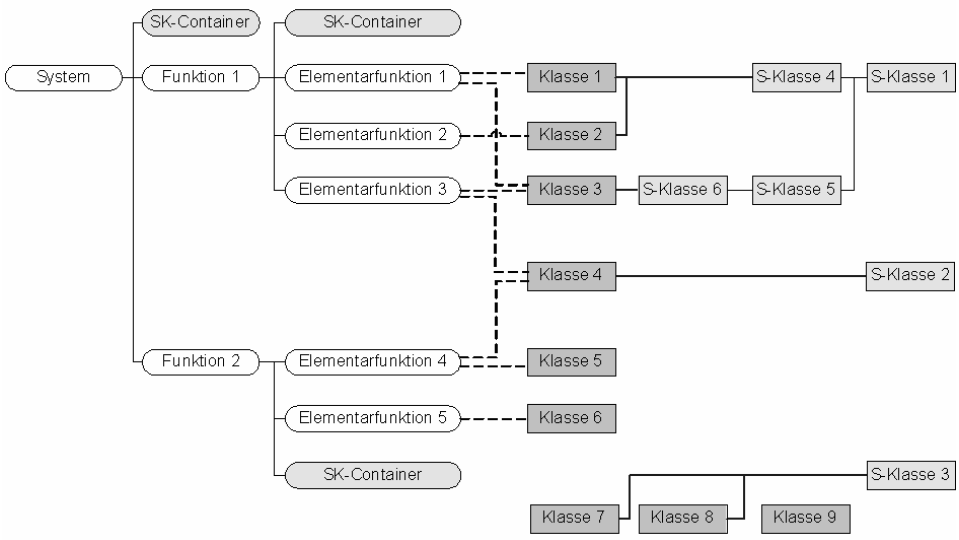

Abb. 3.3: Elementarfunktionale Zuordnung der Klassen

Die Superklassen werden lediglich den einzelnen Klassen zugeordnet und aus Übersichtlichkeitsgründen am rechten Rand notiert.

Schritt 4: Identifikation der Frameworks und Zuordnung der Superklassen zu den Frameworks

Nach Zuordnung der einzelnen Klassen können die Komponenten-Frameworks identifiziert werden. Alle Klassen ohne elementarfunktionalen Bezug und damit ohne einen Bezug zu den fachlichen Aspekten des untersuchten Anwendungssystems bilden das Komponenten-System-Framework. Klassen mit Zuordnung zu mehreren Elementarfunktionen verschiedener Funktionsbereiche werden dem Komponenten-Anwendungs-Framework zugeordnet (vgl. Abb. 3.4), da bei diesen Klassen zwar ein Bezug zu den fachlichen Aspekten der Anwendung vorhanden ist, allerdings eine Zuordnung der Klas-

Klassen zu mehreren Fachkomponenten aufgrund der dadurch entstehenden Redundanz nicht sinnvoll ist.

Nachdem alle Klassen zugeordnet und die Frameworks identifiziert wurden, können die Superklassen zugewiesen werden. Die Zugehörigkeit ergibt sich aus den Vererbungsbeziehungen der jeweiligen Superklasse. Zu beachten ist hier, dass Superklassen einzelner Vererbungsbäume vom tiefsten Punkt beginnend zugeordnet werden müssen. Begonnen wird mit den Superklassen ohne Subklassen, im Beispiel die Superklassen 1, 2 und 3. Wurden diese zugewiesen, so können die restlichen Superklassen zugewiesen werden, wenn alle aus ihnen abgeleiteten Klassen bereits zugeordnet wurden. Unter Berücksichtigung der Beziehungen der Superklassen untereinander existieren vier Möglichkeiten, diese zuzuweisen:

- *Elementarfunktionale Zuordnung der Superklassen:* Superklassen, aus denen nur (Super-)Klassen abgeleitet werden, die einer spezifischen Elementarfunktion angehören, werden dieser Elementarfunktion zugeordnet. In Abbildung 3.4 erfüllt die *Superklasse 6* diese Anforderung. Aus ihr leitet sich nur die *Klasse 2* ab, die nur zur Erfüllung der *Elementarfunktion 2* dient.

- *Funktionale Zuordnung der Superklassen:* Superklassen, aus denen nur (Super-)Klassen abgeleitet werden, die verschiedenen Elementarfunktionen zugeordnet sind, die alle einem Funktionsbereich angehören, werden dem Superklassencontainer des Funktionsbereiches zugeordnet. In der Abbildung erfüllt die *Superklasse 2* diese Anforderung. Aus ihr leiten sich die Klassen 1 und 3 ab, die aber nur zur Erfüllung der Elementarfunktionen der *Funktion 1* eingesetzt werden.

- *Überfunktionale Zuordnung der Superklassen:*

  - Superklassen werden dem Komponenten-Anwendungs-Framework zugeteilt, wenn aus ihnen (Super-)Klassen abgeleitet werden, die unterschiedliche Funktionsbereiche oder Elementarfunktionen unterschiedlicher Funktionsbereiche bedienen. In der Abbildung erfüllen die Superklasse 1, 3 und 4 diese Bedingung.

  - Superklassen, aus denen nur Klassen des Komponenten-System-Frameworks abgeleitet werden, werden dem Superklassencontainer des Komponenten-System-Frameworks zugeteilt. Diese Bedingung ist durch *Superklasse 5* in Abbildung 3.4 erfüllt..

- *Zuordnung auf Anwendungsebene:* Eine Superklasse aus der sich alle anderen (Super-)Klassen der Anwendung ableiten, ist dem Superklassencontainer des Systems zuzuweisen.

Durch die eingeführten Container ist bei einer Erweiterung des Systems oder einer spezifischen Funktion schnell ersichtlich, welche Vererbungsbeziehungen existieren und aus welchen Klassen die die Anwendung erweiternden Klassen abgeleitet werden müssen.

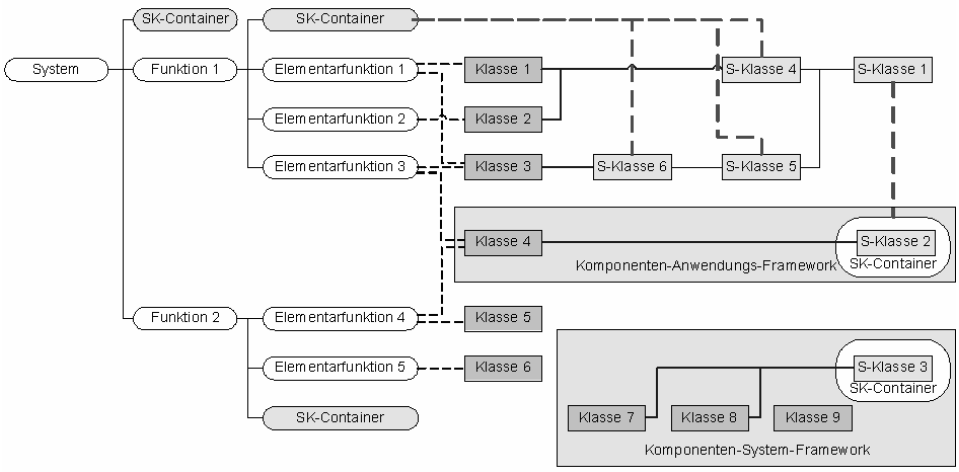

Abb. 3.4: Identifikation der Frameworks und Zuordnung der Superklassen

Die Zugehörigkeiten der einzelnen Superklassen zu einem Superklassen-Container ist in Abb. 3.4 gestrichelt dargestellt.

Schritt 5: Identifikation der Fachkomponenten

Zum Abschluss des Lösungsalgorithmus werden die Fachkomponenten identifiziert. Eine Fachkomponente setzt sich zusammen aus den Klassen, die eindeutig einem Funktionsbereich zugeordnet wurden. Es sind also nur Klassen enthalten, die zur Erfüllung der einzelnen Elementarfunktionen dieser Komponente verwendet werden.

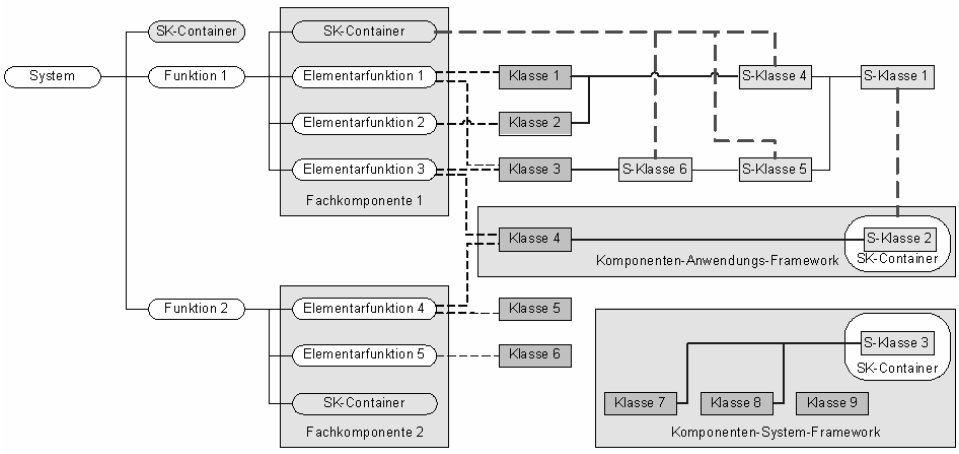

Abb. 3.5: Identifikation der Fachkomponenten

Das Prinzip, das hinter dieser Einteilung der Fachkomponenten steht, ist die Minimierung des Kommunikations- und Koordinationsaufwandes zwischen zwei Fachkomponenten. Bei Anwendungssystemen, deren Funktionen aus einer extrem hohen oder einer extrem niedrigen Anzahl an Elementarfunktionen bestehen, kann diese Regel zur Identifikation der Fachkomponenten nicht mehr sinnvoll sein. Dann liegt es nahe, im ersten Fall die Elementarfunktionen einer Funktion auf mehrere Fachkomponenten zu verteilen, oder im zweiten Fall die Elementarfunktionen mehrerer Funktionen zu einer Fachkomponente zu vereinen. Die Verteilung der Superklassen auf die Superklassen-Container wäre entsprechend zu ändern.

4 Zusammenfassung und Ausblick

In diesem Beitrag wurde ein Lösungsalgorithmus zur Komponentenfindung und Identifikation von Komonentenanwendungs- und -systemframework in monolithischen objektorientierten Systemen vorgestellt, um Anwendungen auf eine wiederverwendungsgetriebene Softwareentwicklung vorzubereiten und damit Vorteile wie etwa eine erleichterte Erweiterbarkeit und Wartbarkeit zu erschließen. Dabei werden die einzelnen Komponenten identifiziert, indem die funktionalen Eigenschaften des Systems einer Dokumentation der Implementierung gegenübergestellt werden. Die Dokumentation der Implementierung muss zur Durchführung des Algorithmus die Objektbeziehungen abbilden, um eine Zuordnung von Klassen zu Funktionen durchführen zu können.

Ausgangspunkt für den vorgestellten Lösungsalgorithmus ist die Spezialisierung eines von den Autoren entwickelten allgemeinen Ansatzes zur Komponentenfindung. Die darin vorhandenen, durch den Anspruch der Allgemeingültigkeit bedingten Ungenauigkeiten, wurden durch den Einsatz in einem Praxisprojekt herausgearbeitet. Aufbauend auf den in diesem Projekt erlangten Erkenntnissen, wurde der Lösungsalgorithmus angepasst, um eine Berücksichtigung der Charakteristika objektorientierter Systeme zu gewährleisten.

Der dabei herangezogene Basisalgorithmus kann noch in zweierlei Hinsicht spezialisiert werden. Zum einen muss eine Validierung des Algorithmus für prozedural implementierte Systeme vorgenommen werden, wobei sich aufgrund der nicht vorhandenen Vererbungsproblematik keine großen Änderungen am Algorithmus ergeben dürften. Zum anderen ist eine Vereinigung der Paradigmen von kompositorischer Wiederverwendung und Drei-Schichten-Architektur vorgesehen, was in Folge eine Anpassung des in diesem Beitrag vorgestellten Lösungsalgorithmus bedeutet.

Literatur

[AL03] Albani, A.; Keiblinger, A.; Turowski, K.; Winnewisser, C.: *Domain Based Identification and Modelling of Business Component Applications.* In: Kalinichenko, L.; Manthey, R.; Thalheim, B.; Wloka, U. (Hrsg.): 7th East-European Conference on Advances in Databases and Informations Systems

(ADBIS-03), LNCS 2798. Dresden, Deutschland 2003, S.30-45.

[Ba96] Balzert, H.: *Lehrbuch der Software-Technik – Software-Entwicklung.* Heidelberg, Spektrum, Akademischer Verlag, 1996.

[Ba99] Balzert, H.: *Lehrbuch der Objektmodellierung – Analyse und Entwurf.* Heidelberg, Spektrum, Akademischer Verlag, 1999.

[BA04] Bayer, J.; Girard, J.-F.; Knobel, J.; Kolb, R.; Muthig, D.: *Architekturentwicklung, basierend auf existierenden Systemen* In: Böckle, G.; Knauber, P.; Pohl, K.; Schmid, K. (Hrsg.): Software-Produktlinien – Methoden, Einführung und Praxis. Heidelberg, dpunkt.verlag, 2004, S.165-176.

[CE00] Czarnecki, K.; Eisenecker, U.: *Generative programming : methods, tools, and applications.* 2000, Boston: Addison Wesley.

[Fo00] Fowler, M.: *Refactoring: Wie Sie das Design vorhandener Software verbessern.* Addison-Wesley, München 2000.

[KZ03] Krammer, A.; Zaha, J. M.: *Komponentenfindung in monolithischen betrieblichen Anwendungssystemen.* In: Turowski, K. (Hrsg.): 5. Workshop Komponentenorientierte betriebliche Anwendungssysteme. Augsburg 2003, S.155-166.

[Li98] Lim, W.C.: *Managing Software Reuse.* 1998; New Jersey: Prentice Hall, S. 102 - 129.

[Mc68] Mcllroy, M. D.: Mass Produced Software Components. In: *P. Naur; B. Randell (Hrsg.)*: Software Engineering: Report on a Conference by the NATO Science Comittee. NATO Scientific Affairs Division, Brussels 1968, S. 138-150.

[Sa97] Sametinger, J.: *Software engineering with reusable components.* 1997; Berlin, New York: Springer.

[Sc91] Scheer, A.-W.: Architektur integrierter Informationssysteme. Springer, Berlin 1991.

[Tu03] Turowski, K.: *Fachkomponenten: Komponentenbasierte betriebliche Anwendungssysteme.* 2003, Aachen, Shaker Verlag.