Erschienen in: Information Management & Consulting 14. Jg., Heft 2, 1999, S. 11-18

Component Ware – Software-technische Konzepte und Perspektiven für die Gestaltung betrieblicher Informationssysteme

Prof. Dr. Ulrich Frank

Zusammenfassung

Anwendungsnahe Software-Komponenten stellen in Aussicht, qualitativ hochwertige betriebliche Informationssysteme kostengünstig aus sorgfältig entworfenen, wiederverwendbaren Bausteinen zusammenzusetzen. Dieser attraktiven Vision steht allerdings eine Reihe von Fragen gegenüber. Sie beginnen bei den begrifflichen Grundlagen (was ist eine Komponente?) und setzen sich fort bei den Nutzungsvoraussetzungen und Einsatzbedingungen von Komponenten. Vor diesem Hintergrund stellt der Beitrag die (nicht immer einheitlichen) software-technischen Konzepte dar, die im Zusammenhang mit Komponenten diskutiert werden. Dabei werden den berechtigten Hoffnungen, die sich an Component Ware knüpfen, die zum Teil erheblichen Herausforderungen gegenübergestellt, die noch zu bewältigen sind, um Visionen wie plug&play von Anwendungskomponenten zu realisieren.

Summary

The composition of information systems from prefabricated domain level software components offers a very attractive perspective: It promises to improve software quality and to decrease development costs at the same time. However, despite some promising concepts, there is still a number of problems to be taken into account. There is no common concept of a software component. How do components relate to other software artefacts? Retrieving, understanding and designing software from components are other non-trivial tasks to be accomplished. Against this background the article gives an overview of concepts and technologies that are related to component ware. Furthermore it points at the preconditions that have to be fulfilled in order to take full advantage of the potential offered by components.

Stichworte: Software-Komponente, Framework, Wiederverwendung, Geschäftsobjekt

Keywords: Component, Framework, Software-Reuse, Business Object

1 Die Vision

Kaum ein anderer software-technischer Ansatz sorgt zur Zeit für soviel Aufsehen wie „Component Ware“ ([1], [34]). Das gilt sowohl für die wissenschaftliche Forschung als auch für die Vermarktung einschlägiger Produkte. Dieser Umstand ist insofern verständlich als die Verheißungen, die mit Komponenten verbunden sind, nicht nur software-technisch und ökonomisch ausgesprochen attraktiv sind, sondern sich darüber hinaus auch in anschaulicher Weise vermitteln lassen: Ähnlich wie in traditionellen Industriezweigen sollen häufig wiederverwendbare Software-Komponenten ingenieurmäßig entworfen und erstellt werden. Angesichts der nahezu vernachlässigbaren Kosten für die Vervielfältigung von Software ergibt sich so eine Perspektive, die aus wirtschaftlicher Sicht geradezu unwiderstehlich wirkt:

Die Qualität von Software wird bei gleichzeitig deutlich geringeren Kosten nachhaltig verbessert. Neben Wiederverwendung ist Integration ein weiterer wichtiger Anspruch, der mit Komponenten verbunden ist. Anwendungssysteme sollen in komfortabler Weise durch das Zusammenfügen von Software-Komponenten erstellt werden. Dabei wird mitunter in Anlehnung an Hardware-Komponenten von „Plug&Play“ [6] gesprochen. Im Idealfall wächst die Systemfunktionalität auf diese Weise monoton, das heißt, das Hinzufügen einer neuen Komponente hat keine Seiteneffekte auf andere Systemteile. Die wirtschaftlichen Potentiale von Komponenten konkretisieren sich seit einiger Zeit im Entstehen neuer Märkte – für Komponenten, einschlägige Beratung, Werkzeuge für die Verwaltung und Komposition von Komponenten wie auch für Anwendungen, die wesentlich aus Komponenten konfiguriert sind.

Dessen ungeachtet ist die gegenwärtig zu verzeichnende Begeisterung allerdings auch erstaunlich, da die skizzierte Vision seit mehr als 30 Jahren diskutiert wird [18]. Immer wieder wurden während dieser Zeit in einschlägigen Publikationen zum Teil euphorische Hoffnungen geäußert. So empfahl Biggerstaff [3] Wiederverwendbarkeit als "the essence of design" zu betrachten. Cox [8] glaubte gar, hier die "silver bullets" zur endgültigen Überwindung der Software-Krise zu erkennen. Handelt es sich also nur um einen weiteren Beleg dafür, daß in der Datenverarbeitung immer wieder alte Ideen in neuem Gewand präsentiert werden – oder gibt es darüber hinaus inhaltliche Gründe für die Renaissance des Themas? Es ist wohl eine Mischung aus beidem: So ist die Vision nach wie vor attraktiv und erscheint im Unterschied zu Verheißungen, wie sie etwa von der Künstliche Intelligenz Forschung verkündet wurden, grundsätzlich umsetzbar. Gleichzeitig ist in den letzten Jahren eine Reihe von Entwicklungen zu verzeichnen, die durchaus Anlaß geben, das Thema wieder mit einer hohen Priorität zu behandeln:

- Software-technische Innovationen, wie objektorientierte Konzepte und Technologien, Frameworks und Entwurfsmuster unterstützen die Entwicklung und den Einsatz von Komponenten.
- Während frühe Bibliotheken wiederverwendbarer Komponenten, wie etwa die Ada-Komponenten von Booch [4], von anwendungsnahen Konzepten bewußt abstrahierten, werden in der gegenwärtigen Diskussion vor allem solche Komponenten diskutiert, die ein hohes Maß an Anwendungssemantik beinhalten – was sich z.B. in dem Schlagwort „business object“ ausdrückt.
- Seit einiger Zeit wird der mit dem Komponenteneinsatz einhergehende *Investitionsschutz* betont. Er bezieht sich einerseits auf die weitere Nutzung von Alt-Systemen in komponenten-orientierten Architekturen, die durch geeignete Verkapselungskonzepte in Aussicht gestellt wird. Andererseits sind verschiedene Bemühungen erkennbar, anwendungsnahe Komponenten zu standardisieren und damit die in sie getätigten Investitionen zu schützen.
- Last but not least ist der Erfolg einschlägiger Komponententechnologien wie etwa JavaBeans oder ActiveX zu berücksichtigen. Auch wenn es sich dabei typischerweise um Komponenten handelt, die weitgehend von den Besonderheiten konkreter Anwendungen abstrahieren, nährt ihre rasche Verbreitung die Hoffnung auf einen Markt für anwendungsnahe Komponenten.

2 Begriffliche Grundlagen

Es ist bezeichnend für die Mehrdeutigkeit des Komponentenbegriffs, daß in einer jüngst erschienenen Ausgabe einer Fachzeitschrift eine Reihe von Experten u.a. danach gefragt wurden, was sie unter einer Komponente verstehen. Die Antworten fielen nicht einheitlich aus [33]. Eine weitere Gegenüberstellung voneinander abweichender Definitionen findet sich in ([34], S. 164 ff.). So kennzeichnet Booch [4] eine Komponente als ein logisch zusammenhängendes Modul, das eine bestimmte Abstraktion darstellt. Demgegenüber betonen andere, daß Komponenten in binärer Form vorliegen (z.B. [16], [33]). Daneben wird mitunter auf den Anspruch verwiesen, Komponenten plattformunabhängig einsetzen zu können. Beide Forderungen sind nur unter der Voraussetzung kompatibel, daß eine Kommunikationsinfrastruktur existiert, die Unterschiede zwischen Programmiersprachen, Betriebssystemen und Hardware zu verbergen gestattet. Eine besonders breite Begriffsfassung findet sich in [31], für den Komponenten „self-contained, clearly identifiable pieces" sind, die entweder im Quellcode oder im Binärcode vorliegen können. I.d.R. wird betont, daß Komponenten wiederverwendbar sein sollen. Dabei wird Wiederverwendung dadurch unterstützt, daß die Implementierung einer Komponente verkapselt ist und lediglich über eine Schnitttstelle verfügbar gemacht wird.

Aus software-technischer Sicht stellt sich die Frage, wie sich Komponenten von Objekten, Klassen, Typen oder Modulen abgrenzen lassen. Da es keine einheitliche Terminologie gibt, sind die folgenden Ausführungen als Versuch zu sehen, eine konsistente Begrifflichkeit zu beschreiben. Dieser Versuch orientiert sich an den Vorschlägen in [34]. Im Unterschied zu Objekten sollten Komponenten keine Identität und keinen persistenten Zustand haben. Szyperski begründet diese Forderung mit Wiederverwendbarkeit und Wartbarkeit. Die Verwendung einer Komponente sollte unabhängig von ihrem Zustand erfolgen können. Die Wartung wird dadurch erleichtert, daß in einem System eine Komponente nur einmal enthalten sein sollte – wobei u.U. mehrere Instanzen existieren. Häufig werden Komponenten auf gewisse Ressourcen (andere Komponenten oder Systembestandteile) angewiesen sein. Im Hinblick auf die Wiederverwendbarkeit ist es deshalb wichtig, die explizite Festlegung solcher Abhängigkeiten zu fordern.

Ähnlich wie ein Typ wird die Schnittstelle einer Komponente (vor allem) durch die Signaturen der angebotenen Operationen spezifiziert und abstrahiert von der jeweiligen Implementierung. Komponenten können Schnittstellen von anderen Komponenten erben. In diesem Fall repräsentiert eine Komponente mehrere Schnittstellen bzw. Typen. Damit weist eine Komponente deutliche Parallelen zu Klassen auf. In der Tat kann eine Komponente in Form einer Klasse implementiert sein. Sie kann aber auch als eine Menge von Klassen implementiert sein oder überhaupt nicht objektorientiert in Form eines Moduls mit verkapseltem Inhalt. Außerdem kann eine Komponente aus anderen Komponenten bestehen. Eine Instanz einer Komponente korrespondiert mit einem Objekt. Szyperski ([34], S. 11) verweist 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.

Zusammenfassend können wir feststellen, daß Komponenten eine Abstraktion über Klassen, Modulen oder Systemen darstellen. Es gibt also im Einzelfall keinen notwendigen Unterschied zu diesen anderen software-technischen Artefakten. In der Außensicht kann eine Komponente mit einer Klasse verglichen werden. Die Besonderheit von Komponenten ergibt sich den intendierten Verwendungszweck: Komponenten sollten wiederverwendbar und zusammenfügbar sein. Die Nutzung einer Komponente vollzieht sich in einem bestimmten Kontext, innerhalb dessen das jeweilige Komponentenmodell bekannt ist: in einem (heterogenen) Netzwerk, auf einer Plattform, innerhalb eines dedizierten Frameworks oder einer bestimmten Klasse von Anwendungen (z.B. von Internet-Browsern). Um die skizzierten Anforderungen zu erfüllen, werden bestimmte Vereinbarungen bzw. Standards hinsichtlich der angebotenen Schnittstellen festgelegt. Entsprechende Spezifikationen, in Abb. 1 durch die „Standardschnittstelle" gekennzeichnet, werden auch *Komponentenmodell* genannt. Auf diese Weise wird eine Komposition von Komponenten unterstützt, die von Programmierern unabhängig voneinander erstellt wurden. Darüber hinaus wird es möglich, Standard-Werkzeuge und Middleware für die Analyse, Komposition, Modifikation und Ausführung von Komponenten zu verwenden.

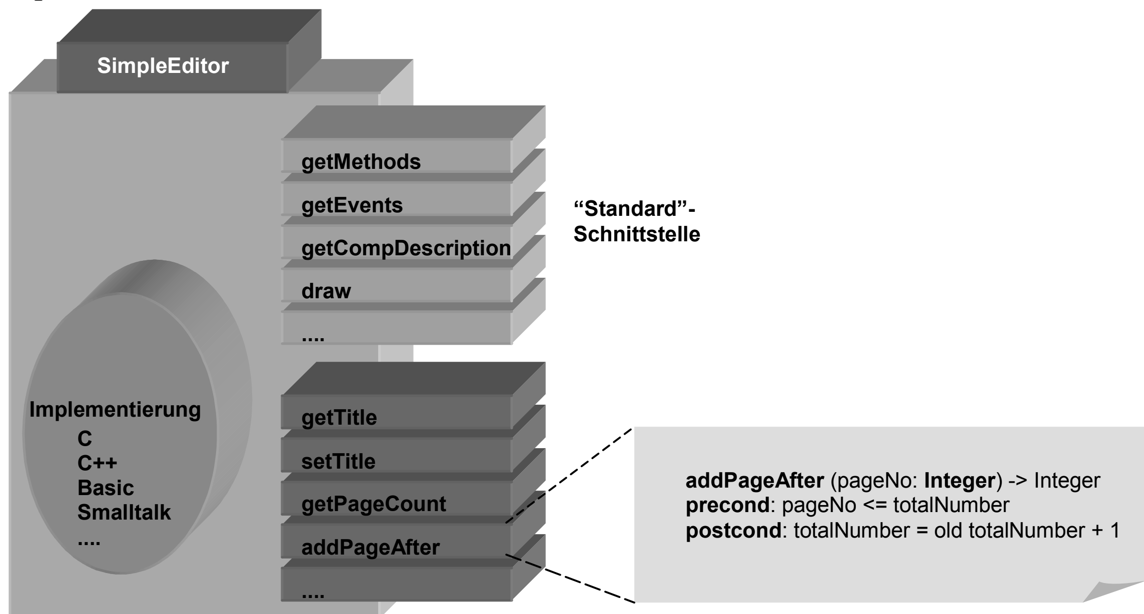

Abb. 1: Beispiel für Komponente und Vertrag

3 Nutzungsvoraussetzungen

Die effiziente Nutzung von Komponenten ist mit einer Reihe nicht trivialer Anforderungen verbunden:

- Der Entwickler muß wirksam darin unterstützt werden, nicht nur prinzipiell geeignete Komponenten zu finden, sondern diejenigen, die für den jeweiligen Verwendungszweck am besten geeignet sind.
- Dazu muß ihm - auf einer angemessenen Abstraktionsebene - verdeutlicht werden, was die Komponenten leisten. Anders formuliert: Er muß ihre Bedeutung im Rahmen seiner Entwurfstätigkeit verstanden haben.

- Die Komposition der Komponenten sollte für den Entwickler einerseits in komfortabler Weise möglich sein, andererseits sollte sie software-technischen Anforderungen an Sicherheit und Flexibilität genügen.
- Da nicht davon ausgegangen werden kann, daß alle bereitgestellten Komponenten exakt den Anforderungen einer spezifischen Implementierung genügen, ist eine komfortable und sichere Modifikation oder Substitution der Komponenten zu ermöglichen.

3.1 Retrieval von Komponenten

Es gibt eine Reihe unterschiedlicher Ansätze zur Unterstützung der Suche nach Komponenten. Ansätze, die ein symbolisches oder syntaktisches Retrieval unterstützen, sind darauf gerichtet, möglichst viele der denkbaren Assoziationen der Verwender bei der Suche zu berücksichtigen. Die Anwendung von Verfahren zur Volltext-Suche ist deshalb naheliegend. Frakes und Nejmeh [12] skizzieren einen entsprechenden Ansatz. Er basiert darauf, daß Quellcode oder ausgezeichnete Teile (wie Kommentare) desselben invertiert werden und damit eine leistungsfähige Suche nach einzelnen Zeichenketten (die auch durch boolesche Operatoren verknüpft sein können) möglich wird. Ein solcher Ansatz ist relativ leicht zu implementieren. Seine Bedeutung ist allerdings dadurch eingeschränkt, daß einzelne Wörter des Quellcodes selten repräsentativ für die Bedeutung des gesamten Fragments sind. Außerdem ist dieses Verfahren darauf angewiesen, daß die wiederverwendbaren Komponenten im Quellcode vorliegen. Diesem Problem kann durch die Verwendung von Schlagwörtern oder – besser – von Thesauri entgegengewirkt werden. Dabei ist allerdings zu bedenken, daß die Bedeutung von Bezeichnern sowie ihre Beziehungen untereinander kontextabhängig sind. Einige Ansätze zur syntaktischen Unterstützung der Suche berücksichtigen diesen Umstand, indem sie Kategorien oder Facetten definieren, denen Komponenten zugeordnet werden können (z.B. [24], [27]).

Andere Ansätze zur Unterstützung der Suche nach Komponenten berücksichtigen die Semantik von Komponenten. Bei vereinfachender Betrachtung treten sie in drei Ausprägungen auf. Im einfachsten Fall werden Typinformationen, also die Spezifikation der Schnittstellen von Komponenten verwendet (z.B. [28], [25]). Andere Ansätze werten die formale Spezifikation der angebotenen Komponenten aus (z.B. [7], [30]). Unabhängig von der Leistungsfähigkeit solcher Verfahren, ist ihre Anwendbarkeit dadurch eingeschränkt, daß gerade für anwendungsnahe Komponenten i.d.R. keine formalen Spezifikationen vorliegen bzw. ihre Erstellung zu aufwendig ist. Ein weiteres Suchverfahren basiert auf der Ausführung von Operationen („execution based retrieval“, [26]). Dazu wird zunächst durch die Auswertung von Typinformationen die Zahl der in Frage kommenden Operationen eingeschränkt. Für jede dieser Operationen werden nun Eingabewerte erzeugt. Nach der Ausführung wird durch die Analyse des Verhältnisses von Eingabewerten und Ergebnissen versucht, die Treffermenge weiter einzuschränken. Sieht man von dem Aufwand ab, der mit einem solchen induktiven Verfahren verbunden ist, bleiben nachhaltige Zweifel daran, daß es sich eignet, Operationen mit einer reichhaltigen Semantik zuverlässig zu erkennen.

3.2 Dokumentation

Auch wenn ein leistungsfähiges Retrieval dazu beiträgt, die Menge der für einen Verwendungszweck passenden Komponenten nachhaltig einzuschränken, ist damit noch nicht

gewährleistet, daß der Nutzer der Komponente deren Funktionalität hinreichend versteht. Im Hinblick auf Wiederverwendung ist dabei zu berücksichtigen, daß eine Komponente i.d.R. nicht für einen konkreten Einsatzzweck erstellt wurde, sondern eine Abstraktion darstellt. Eine Dokumentation sollte die Operationen einer Komponente verständlich und präzise beschreiben. Dies kann mit einem natürlichsprachlichen Text geschehen – sinnvollerweise ergänzt durch Beispiele, die typische Nutzungsszenarien beschreiben. Ein solcher Ansatz hat allerdings zwei Nachteile: Für das Verständnis eines Textes ist seine Struktur von erheblicher Bedeutung. Wenn die Strukturierung dem jeweiligen Autor überlassen bleibt, droht einerseits eine verwirrende Vielfalt von Erscheinungsformen, andererseits läuft der Autor Gefahr wichtige Aspekte zu vergessen. Darüber hinaus gehen die Mehrdeutigkeiten der natürlichen Sprache zu Lasten einer möglichst eindeutigen Darstellung. Dem ersten Nachteil kann durch die Verwendung einheitlicher, bewährter Strukturen begegnet werden. Um die sprachliche Präzision der Dokumentation zu erhöhen, können natürlichsprachliche Darstellungen durch (semi-) formale Beschreibungen ergänzt werden. Dazu bieten sich u.a. grafische Modelle wie statische Objektmodelle oder auch dynamische Sequenzdiagramme [20] an. Entwurfsmuster stellen einen Ansatz dar, der beide Maßnahmen zu verbinden gestattet: Sie beschreiben einen abstrakten Entwurf, der für eine Klasse von Anwendungsfällen konkretisiert werden kann. Dabei ist auch die Verwendung von Diagrammen vorgesehen. Auch wenn es keine standardisierte Struktur für Entwurfsmuster gibt, hat der Vorschlag von [14] mittlerweile eine erhebliche Verbreitung erfahren. Die für ein Entwurfsmuster verwendete Struktur kann zudem für das Retrieval von Komponenten genutzt werden. Da die Leser einer Dokumentation nicht immer gleiche Voraussetzungen mitbringen, ist es angeraten, verschiedene Detaillierungsstufen vorzusehen, die etwa durch den Einsatz hypermedialer Dokumente unterstützt werden können.

3.3 Anpaßbarkeit bzw. Konfiguration

Eine Komponente kann in unterschiedlicher Weise wiederverwendet werden. Im Falle des sog. „blackbox-reuse" ([34], S. 33) präsentiert sich eine Komponente nach außen lediglich durch eine oder mehrere Schnittstellen. Demgegenüber gibt ein „glassbox-reuse" den Blick auf die Implementierung frei, während „whitebox-reuse" nach dem Verständnis einiger Autoren darüber hinaus auch eine Veränderung der Implementierung erlaubt. Es liegt auf der Hand, daß blackbox-reuse im Unterschied zum whitebox-reuse ein höheres Maß an Zuverlässigkeit verspricht und gleichzeitig die Wartung von Systemen, die aus Komponenten bestehen, deutlich erleichtert. Demgegenüber erlaubt whitebox-reuse eine größere Flexibilität. Whitebox-reuse findet man häufig bei der Nutzung veränderbarer Klassenbibliotheken, die im Quellcode vorliegen, wie sie etwa für Smalltalk-Systeme typisch sind. Angesichts des hohen Preises, der für diese zusätzliche Flexibilität zu zahlen ist, sollte sich whitebox-reuse nur auf gut begründete Ausnahmen beschränken.

Im Hinblick auf eine verläßliche Wiederverwendung schlägt Szyperski ([34], S. 43) ähnlich wie Meyer („design by contract", [19]) vor, eine Schnittstelle als einen *Vertrag* zwischen nutzender Software und benutzter Komponente zu betrachten. Der Nutzer einer Schnittstelle verpflichtet sich, beim Aufruf eines Dienstes die zugeordnete Vorbedingung zu erfüllen, während die den Dienst implementierende Komponente das Erfüllen der Nachbedingung und evtl. ergänzender Invariante ([19], S. 364 ff.) zu gewährleisten hat. Grundsätzlich wäre eine möglichst weitreichende Formalisierung von Verträgen wünschenswert, da nur so eine

automatische Überprüfung möglich ist. Hier sind allerdings nach wie vor erhebliche Hürden zu beachten: So ist der mit einer umfassenden Formalisierung verbundene Aufwand i.d.R. zu hoch. Aber selbst dann, wenn Verträge vollständig formalisiert sind, gibt es keine generellen Verfahren, ihre Erfüllung durch eine statische Prüfung zu gewährleisten. Gängige Compiler beschränken sich zumeist auf die Überprüfung eines kleinen Teils einer Schnittstellenvereinbarung: die Typen der Eingabe- und Ausgabeparameter einer Operation.

Sieht man von white-box reuse ab, können Komponenten vor allem durch Spezialisierung an individuelle Anforderungen angepaßt werden. Dabei ist zunächst an Vererbung zu denken. Die Semantik von Vererbung ist für verschiedene Komponentenmodelle, ähnlich wie bei Programmier- oder Modellierungssprachen, nicht einheitlich definiert. Grundsätzlich kann man zwischen Implementierungs- und Schnittstellenvererbung unterscheiden. Während im zweiten Fall der Subtyp nur die Schnittstelle übernimmt, wird im ersten Fall auch die Implementierung vererbt. Im Hinblick auf die individuelle Anpassung spezialisierter Komponenten stellt sich die Frage, ob und ggfs. wie die Signaturen einer Schnittstelle redefiniert werden dürfen. Üblicherweise wird zwischen kovarianter und kontravarianter Redefinition von Parametertypen unterschieden. Im Fall kovarianter Redefinition wird ein geerbter Parametertyp durch einen spezielleren Typ ersetzt, während eine kontravariante Redefinition die Ersetzung durch einen allgemeineren Typ vorsieht. Eine eingehende Diskussion der mit Redefinitionen verbundenen Schwierigkeiten findet sich in [19], S. 621 ff..

Delegation, mitunter auch Komposition genannt, ist ein weiterer Ansatz, Komponenten an individuelle Anforderungen anzupassen. Dazu erbt eine Komponente nicht statisch die Spezifikation einer übergeordneten Komponente. Statt dessen werden Komponenten so gekoppelt, daß eine Komponente zur Laufzeit einen Methodenaufruf, den sie nicht befriedigen kann, an eine vorher dafür festgelegte Komponente delegiert. Auch wenn Vererbung und Delegation zu ähnlichen Resultaten führen, gibt es doch deutliche Unterschiede. Ein ausführlicher Vergleich von Delegation und Vererbung findet sich in [11].

Das Zusammenfügen von Komponenten wird gern mit vereinfachenden Metaphern („Lego“, „Glueing“) veranschaulicht. Eine solche statische Betrachtung übersieht allerdings die Anforderungen, die mit der Integration einer Komponente in Kontrollflüsse von Anwendungen verbunden sind. Dazu sind grundsätzlich zwei Modelle denkbar. Nach Maßgabe der prozeduralen Programmierung werden die von Komponenten angebotenen Operationen bei Bedarf von anderen Komponenten aufgerufen. Man spricht auch von einem „pull“ Modell. Dazu benötigt die aufrufende Komponente eine Referenz auf die Komponente, die die Operation anbietet. Anders in einem „push“ Modell: Hier werden Operationen nicht explizit aufgerufen. Statt dessen reagieren sie auf zuvor definierte Ereignisse. Man spricht in diesem Zusammenhang auch von „connection-oriented programming“ ([34], S. 148 ff.). Dabei wird für jede Komponente zwischen den nach außen angebotenen Operationen („incoming interface“) und den emittierten Ereignissen („outgoing interface“), die der Auslösung von Operationen in anderen Komponenten dienen, unterschieden. Mit dieser Unterscheidung kann eine dedizierte Werkzeugumgebung die Komposition von Komponenten unterstützen, indem sie nur solche Komponenten verbindet, die zueinander passende Operationen und Ereignisse aufweisen.

Auch wenn die Unterscheidung von Eingangs- und Ausgangsschnittstellen die Analogie zu integrierten Schaltkreisen nahelegt, kann ein wesentlicher Unterschied zur Hardware nicht

übersehen werden, der vor allem bei anwendungsnahen Komponenten zum Tragen kommt: die sehr viel größere semantische Vielfalt von Schnittstellen. Im Hinblick auf die Wartung ist es wichtig, daß ein Komponentenmodell die explizite Auszeichnungen von Versionen zuläßt und Richtlinien für das Versionsmanagement vorgibt. Dazu gibt es eine Reihe unterschiedlicher Ansätze (vgl. [34], S. 42 f.).

4 Middleware und Standards

Die Wiederverwendbarkeit von Komponenten wird gefördert, wenn sie auf weit verbreiteten Konzepten bzw. Standards beruhen. Zur Vermeidung von Redundanz sollte eine Komponente von verschiedenen Anwendungen eines Systems genutzt werden können. Darüber hinaus ist es wünschenswert, daß Komponenten in verteilten, heterogenen Umgebungen einsetzbar sind. Gleichzeitig empfehlen die Interessen der Anbieter eine Vermarktung in binärer Form. Die Unterstützung der anwendungs- und ggfs. plattformübergreifenden Kommunikation mit Komponenten erfordert eine geeignete Infrastruktur, häufig „Middleware“ genannt.

4.1 COM, DCOM und ActiveX

Um eine Integration von Anwendungen zu ermöglichen, die, anders als einfaches copy&paste, auch die Semantik von Daten bewahrt, hat Microsoft Anfang der neunziger Jahre die Metapher des „compound document“ aufgegriffen und mit OLE (damals: Object Linking and Embedding) eine Technologie entwickelt, die es gestattet, Daten verschiedener Anwendungen in einem Dokument zu bearbeiten. Abstrahiert man von der für OLE charakteristischen Dokument-Metapher, kommt dieser Ansatz der generellen Aufgabe gleich, Komponenten anwendungsübergreifend zu integrieren. Diesem Umstand hat Microsoft mit COM (Component Object Model, [5], [29]) Rechnung getragen, das als generelles Fundament auch von OLE genutzt wird. Dabei handelt es sich sowohl um eine Spezifikation als auch um eine Implementierung. Die Spezifikation beschreibt ein Komponentenmodell. Dabei wird offengelassen, wie eine Komponente zu implementieren ist: Es kann sich um eine Klasse oder um ein Modul handeln. Die Komponenten werden in binärer Form abgelegt. Um einen programmiersprachenunabhängigen Zugriff zu ermöglichen, verwendet Microsoft eine proprietäre Interface Definition Language (IDL), die Typunterschiede zwischen verschiedenen Programmiersprachen auszugleichen ermöglicht. Eine Komponente kann mehr als eine Schnittstelle anbieten. COM unterstützt einfache Schnittstellenvererbung, aber keine Implementierungsvererbung.

Die mit COM einhergehende Implementierung bietet eine Architektur für den Einsatz und die Nutzung binärer Komponenten über verschiedene Anwendungen hinweg. Sie unterstützt u.a. eine gemeinsame Speicherverwaltung für mehrere Instanzen von Komponenten und stellt Dienste für das dynamische Laden und Terminieren von Komponent-Instanzen bereit. Während COM zunächst nur für Microsoft-Plattformen verfügbar war, gibt es mittlerweile Portierungen auf andere Plattformen. Es bleibt aber die Einschränkung, daß COM lediglich die Nutzung von Komponenten auf einer Plattform unterstützt. Um die Realisierung verteilter Systeme zu erlauben, hat Microsoft mit Distributed COM (DCOM) eine Erweiterung von COM vorgenommen, die für den Programmierer weitgehend transparent ist.

Eine OLE-Komponente (auch: OLE control bzw. OCX) ist eine COM-Komponente, die eine Reihe vorgeschriebener Schnittstellen anbietet. ActiveX-Komponenten (ActiveX-controls) basieren auf einer Spezialisierung des COM-Komponentenmodells. Sie können in dafür vorgesehenen Internet-Browsern ausgeführt werden. Dabei ist allerdings einschränkend zu berücksichtigen, daß sie, wie alle COM-Komponenten, nur jeweils auf einer bestimmten Plattform lauffähig sind. Der Empfang von Komponenten im Binärformat ist mit der Gefahr verbunden, daß trojanische Pferde eingeschleust werden. Zur Verringerung dieser Gefahr können sich Software-Anbieter bei Microsoft um einen geheimen Identifikationscode bemühen, den sie zur Authentifikation der von ihnen angebotenen Komponenten verwenden können („authenticode“) .

4.2 CORBA und OMA

Im Unterschied zu Microsoft zielt die Object Management Group (OMG), ein Konsortium namhafter Anbieter und Anwender, darauf, einen hersteller- und plattformunabhängigen Standard für die Kommunikation von Objekten bzw. Komponenten in verteilten heterogenen Systemen zu spezifizieren. Auf diese Weise wird es prinzipiell möglich, daß Instanzen von Klassen, die in der Programmiersprache S1 geschrieben wurden und auf einer Plattform P1 residieren, Operationen eines Objekts aufrufen, dessen Klasse in der Programmiersprache S2 implementiert wurde und das auf einer Plattform P2 residiert – wenn auf beiden Plattformen ein Object Request Broker nach Maßgabe der von der OMG spezifizierten Common Object Request Broker Architecture (CORBA) verfügbar ist. Durch eine an C++ angelehnte IDL wird eine sprachunabhängige Kommunikation ermöglicht. Dazu muß für die Klassen bzw. Typen der verwendeten Programmiersprachen eine Abbildung auf die Klassen/Typen des Objektmodells der OMG („Core Object Model“) definiert sein. Die IDL erlaubt multiple Schnittstellenvererbung. Die Object Management Architecture (OMA) umfaßt neben CORBA und der IDL die Spezifikation einer Reihe von Diensten. Die CORBAservices leisten u.a. die Zuordnung sog. Universal Unique Identifiers (UUID) zu symbolischen Namen wie auch das Instanzieren, Löschen und Kopieren von Objekten. Common Facilities bieten u.a. Druckdienste, allgemeine Hilfsfunktionen, aber auch Desktop-Werkzeuge bis hin zu Frameworks für compound documents. Sie dienen damit auch der Verkapselung von Diensten, die üblicherweise von Betriebssystemen angeboten werden.

Entwickler von Klassen, die die CORBA-Infrastruktur nutzen sollen, müssen also einerseits dafür sorgen, daß alle für eine plattformübergreifende Kommunikation benötigten Schnittstellen in der IDL spezifiziert sind. Andererseits sind Referenzen auf plattformspezifische Dienste zu vermeiden. Statt dessen sind die einschlägigen CORBA-Dienste zu verwenden. Im Hinblick auf den Austausch von Klassen bzw. Komponenten über CORBA-Infrastrukturen ist zu berücksichtigen, daß anwendungsnahe Klassen bisher nicht Bestandteil des Standards sind. Wenn beispielsweise eine Komponente „Debitorenkonto“ versendet wird, reicht der Verweis auf die Spezifikationen der OMG nicht aus, um beim Empfänger eine angemessene Interpretation voraussetzen zu können. Dies soll sich nach dem Willen der OMG ändern. Innerhalb der OMG arbeitet die Business Object Domain Task Force (BOMSIG, [21]) seit einigen Jahren daran, die Standardisierung anwendungsnaher Komponenten für betriebliche Informationssysteme („business objects“) vorzubereiten. Bisher gibt es aber noch keine gehaltvollen Vorschläge. Darüber hinaus ist zu berücksichtigen, daß Klassen mit IDL-Schnittstellen erst dann als Komponente gelten, wenn

sie einem bestimmten Komponentenmodell genügen. Die OMG hat bisher noch kein entsprechendes Modell spezifiziert – es ist allerdings für die nahe Zukunft [22] angekündigt.

Im Unterschied zu COM-Komponenten hat sich bisher kein nennenswerter Markt für CORBA-konforme Komponenten etabliert. Das liegt sicher daran, daß die Entwicklung und Vermarktung dedizierter Software-Entwicklungswerkzeuge nur zögerlich voranschreitet. Außerdem braucht ein Markt eine kritische Größe, um für weitere Anbieter attraktiv zu werden. Angesichts der Vorteile, die CORBA für eine Vermarktung von Komponenten gegenüber proprietären Microsoft-Technologien bietet, und der beachtlichen wirtschaftlichen Potenz, die hinter der OMG steht, könnte sich das in Zukunft nachhaltig ändern.

4.3 JavaBeans

Mit der Programmiersprache Java zeichnet sich ein alternativer Ansatz für die Wiederverwendung von Software in verteilten, heterogenen Umgebungen ab. Java-Code wird in Java Byte Code übersetzt. Da auf allen wesentlichen Plattformen virtuelle Maschinen für die Ausführung des Java Byte Codes verfügbar sind, können Java-Programme prinzipiell auf all diesen Plattformen ausgeführt werden. Während andere Sprachen, wie etwa Smalltalk, schon sehr viel länger über ähnliche Eigenschaften verfügen, rührt die besondere Attraktivität von Java von der engen Verzahnung mit dem Internet. So sind virtuelle Maschinen für Java seit einiger Zeit obligater Bestandteil der meisten Internet-Browser. Das ermöglicht das Laden und Ausführen kleiner Java-Programme, sog. Applets mittels eines Browsers. Im Hinblick auf die Gestaltung größerer Anwendungen weisen Applets einige Nachteile auf. So ist eine direkte Kommunikation zwischen Applets nicht möglich. Statt dessen müssen Ressourcen, die einzelne Applets zur Ausführung benötigen vom jeweils zuständigen Server angefordert werden. Daneben ist eine Anpassung von Applets an individuelle Bedürfnisse kaum in wirtschaftlicher Weise zu bewältigen, da Applets typischerweise als Java Byte Code versendet werden. Die auf Java basierende Komponenten-Technologie, JavaBeans [15] genannt, wirkt diesen Nachteilen entgegen. Ein JavaBean besteht aus einer Menge von Java-Klassen. Die Kommunikation zwischen JavaBeans erfolgt ereignisgesteuert. Dazu wird für jede Komponente eine Menge von Ereignissen definiert, die sie auslösen kann, sowie eine Menge von Ereignissen, auf die sie in bestimmter Weise reagiert. Die Versendung von JavaBeans wird durch ein spezielles Dateiformat (Java ARchive file) unterstützt. Es dient der Ablage der benötigten Java-Klassen und sonstiger Ressourcen (z.B. Grafiken).

Die Erstellung und Nutzung von JavaBeans wird durch einschlägige Werkzeuge unterstützt. Dabei wird ein Software-Erstellungsprozeß unterstellt, der deutlich vom traditionellen Vorgehen abweicht. Dazu wird zwischen compile time, design time und build time unterschieden ([15], S. 4 f.). Zunächst werden Komponenten unabhängig voneinander implementiert und kompiliert (compile time). Zur weiteren Verwendung werden ausgewählte Komponenten in einem entsprechenden Werkzeug konfiguriert und miteinander gekoppelt (design time). Wenn Konfiguration und Kopplung abgeschlossen sind, können die Komponenten ausgeführt werden (build time). Die Konfiguration („customization“) von JavaBeans erfolgt i.d.R. interaktiv, indem dafür vorgesehene Eigenschaften („properties“) gesetzt werden. Beispiele für solche Eigenschaften sind Farben oder Zeichensätze für GUI-Komponenten. Wenn es für die jeweilige Anwendung sinnvoll ist, können properties auch zur Laufzeit manipuliert werden. Jede Komponente wird durch eine Menge von Eigenschaften,

Operationen und Ereignissen charakterisiert. Diese Charakteristika können mit Hilfe von Analysewerkzeugen inspiziert werden.

Ähnlich wie OLE controls oder ActiveX controls sind JavaBeans vor allem auf generische Funktionen gerichtet, die weitgehend von den Besonderheiten konkreter Anwendungen abstrahieren. Dazu gehören etwa Interaktionselemente zur Gestaltung grafischer Benutzungsschnittstellen, Desktop-Werkzeuge oder Komponenten zum Abspielen von Audio- oder Videosequenzen. Da für Java eine Sprachanbindung zur IDL der OMG existiert, können JavaBeans durch Rückgriff auf CORBA auch transparent mit Komponenten kommunizieren, die in anderen Sprachen implementiert wurden. Zudem wird Software angeboten, die die Einbettung von JavaBeans in ActiveX-Umgebungen ermöglicht. Während im Internet eine Vielzahl kostenloser JavaBeans verfügbar ist, befindet sich die Kommerzialisierung noch im Anfangsstadium. Gegenüber COM-Komponenten bieten JavaBeans vor allem für den Einsatz in großen Unternehmensnetzwerken und im Internet Vorteile, weil dort nach wie vor mit einer heterogenen Rechnerlandschaft gerechnet werden muß. Zudem bieten JavaBeans neben der zentralen Registrierung authorisierter Anbieter ein zusätzliches Sicherheitsmerkmal: JavaBeans werden in einer abgeschirmten Laufzeitumgebung („sandbox“) ausgeführt, die Seiteneffekte über ihre Grenzen hinaus ausschließt.

5 Frameworks und Anwendungsarchitekturen

Die Mehrheit der heute verfügbaren Komponenten ist auf die Gestaltung von Benutzungsschnittstellen (Grafik, Audio, Video) gerichtet. Das für JavaBeans skizzierte Entwicklungsszenario (compile, design, run), also letztlich eine bottom up Vorgehensweise, mag für solche Einsatzzwecke brauchbar sein. Wenn aber wesentliche Teile der Semantik einer Anwendung durch Komponenten abgedeckt werden sollen, wird auch ein top down Ansatz benötigt. Neben konzeptionellen Modellen, auf die noch einzugehen sein wird, ist dabei vor allem an Architekturen zu denken, die dem Entwickler auf unterschiedlichen Abstraktionsebenen angeboten werden können.

5.1 Komponenten-Frameworks am Beispiel von IBM San Francisco

Ein Framework ist eine mehr oder weniger abstrakte, mehr oder weniger vollständige Architektur einer Klasse von Software. Im Falle eines Komponenten-Frameworks besteht eine solche Architektur aus Komponenten, den Interaktionsbeziehungen zwischen Komponenten, und ggfs. einer Menge von Integritätsbedingungen [9]. Beim Entwurf eines Frameworks wird idealtypisch zwischen den für alle Ausprägungen der betrachteten Domäne invarianten Sachverhalten, abgebildet durch "generische Konzepte", und den an spezifische Einzelfälle anpaßbaren Konzepten unterschieden. Die nicht zu ändernden Teile eines Frameworks werden mitunter auch "frozen spots", die für anwendungsspezifische Modifikationen offenen "hot spots" genannt.

Nach mehreren erfolglosen Versuchen, generische Architekturen für Anwendungssoftware zu realisieren, hat IBM 1994 mit einer Reihe von Partner-Unternehmen das Projekt "San Francisco" gestartet. Ziel des Projekts ist ein erweiterbares, komponentenbasiertes Framework zur Realisierung betrieblicher Informationssysteme. Dazu wird eine mehrschichtige Architektur verwendet. Dabei nimmt von Ebene zu Ebene die anwendungsspezifische

Semantik der eingesetzten Komponenten zu. Während etwa auf der untersten Ebene Komponenten eingesetzt werden, die für Sicherheit und Persistenz zuständig sind, sind auf der zweiten Ebene von oben generische Geschäftsobjekte angesiedelt. Die oberste Ebene ist solchen Komponenten vorbehalten, die spezifisch für die jeweilige Anwendungsdomäne entwickelt wurden. Die gegenwärtig verfügbare Version des Frameworks ist vollständig in Java implementiert ([35], S. 303).

Der Nutzen von Frameworks für die Software-Entwicklung mit Komponenten liegt auf der Hand. Sie liefern gleichsam den „Rohbau“ eines Systems mit klaren Schnittstellen zum Anfügen weiterer Komponenten, die für die Anpassung an konkrete Anforderungen benötigt werden. Wenn der Bauplan eines Frameworks offengelegt ist, kann zudem ein Markt für die spezifizierten Komponenten entstehen. Auf diese Weise kann sich der Anwender die jeweils geeignetste Komponente („best of breed“) aussuchen.

5.2 Allgemeine Komponentenarchitekturen

Die Architektur eines komponentenbasierten Systems ist tendenziell anspruchsvoller als die eines vergleichbaren traditionell entworfenen Systems. Das liegt daran, daß die Semantik der verwendeten Komponenten allenfalls eingeschränkt angepaßt werden kann, gleichzeitig aber eine hohe Flexibilität des Gesamtsystems erreicht werden soll: „It is all about forming a system architecture that is independently extensible on all levels.“ ([34], S. 84). Szyperski skizziert eine Komponentenarchitektur, die auf der Annahme basiert, daß alle Teile eines Systems, beginnend beim Betriebssystem, aus Komponenten bestehen. Die Anwendungen entstehen durch die Nutzung komponentenbasierter Frameworks. Solche Frameworks bieten Protokolle für die Kommunikation der von ihnen verwendeten und ggfs. vom Anwender hinzugefügten Komponenten. Auf einer Plattform können mehrere Frameworks residieren, die u.U. miteinander kommunizieren.

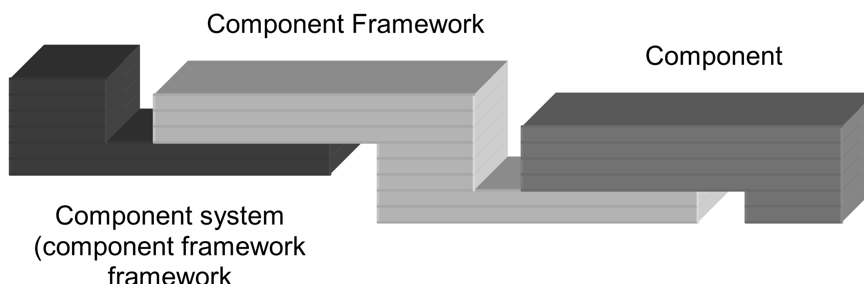

Abb. 2: Skizze einer allgemeinen Komponentenarchitektur ([Szy97], S. 277)

Um eine entsprechende Integration verschiedener Frameworks zu unterstützen, schlägt Szyperski ein Basiskomponentensystem („component component framework“) vor. Auf diese Weise entsteht eine Architektur, die drei Stufen („tier“) vorsieht, für die jeweils mehrere Schichten (wie etwa die verschiedenen Schichten des San Francisco Frameworks) existieren (Abb. 2).

6 Ausblick: Konzeptionelle Referenzmodelle als zentrale Herausforderung

Auch wenn Visionen wie die von wiederverwendbaren Geschäftsobjekten die weitere Entwicklung eines Komponentenmarkts beflügeln, so sind die gegenwärtig verfügbare Komponententechnologie sowie die zugehörigen Spezifikationen auf einem eher technischen Niveau angesiedelt. Von der spezifischen Semantik einzelner Anwendungsdomänen wird weitgehend abstrahiert. Gleichzeitig macht es wenig Sinn, anwendungsspezifische Komponenten isoliert anzubieten: Je mehr Semantik eine Komponente aufweist, desto schwieriger ist sie in beliebigen Umgebungen wiederzuverwenden. Anwendungsnahe Frameworks stellen einen wichtigen Schritt zur komfortablen Verwendung von Komponenten auf verschiedenen Abstraktionsebenen dar. Die Realisierung solcher Frameworks ist mit einem großen Aufwand verbunden. Dabei geht es nicht allein um den Entwurf einer leistungsfähigen Architektur, sondern vor allem um die Spezifikation anwendungsnaher wiederverwendbarer Komponenten. Das erfordert ein tiefes Verständnis der jeweiligen Domäne, welches wiederum eine intensive Kommunikation mit qualifizierten Fachleuten voraussetzt. Konzeptionelle Modelle stellen ein Medium für eine solche Kommunikation dar und bieten gleichzeitig eine Basis für den Software-Entwurf. Im Unterschied zu konzeptionellen Modellen, die für singuläre Domänen erstellt werden, sind solche Modelle, die eine Grundlage für die Entwicklung von Frameworks darstellen, mit dem Anspruch verbunden, für möglichst viele Anwendungsfälle in einem sinnvoll abgegrenzten Bereich gültig zu sein. Im Hinblick auf betriebliche Informationssysteme geht es dabei nicht allein um eine Generalisierung über tatsächliche Geschäftsprozesse, Informationsobjekte und die zugehörige Begrifflichkeit. Um die Potentiale von Informationstechnologie auszuschöpfen, empfiehlt es sich i.d.R., die Organisation von Leistungserstellungsprozessen zu ändern. Dazu gehört auch eine Vereinheitlichung von Begriffssystemen, deren Varianz zum Teil allein darauf zurückzuführen ist, daß Begriffe in verschiedenen Unternehmen unabhängig voneinander gewachsen sind.

Seit einiger Zeit werden konzeptionelle Modelle, die mit dem Anspruch verbunden sind, für bestimmte Bereiche, etwa für Branchen, generell einsetzbar zu sein und gleichzeitig eine, von faktischen Gegebenheiten bewußt abstrahierende Vorbildfunktion zu erfüllen unter dem Etikett „Referenzmodell“ diskutiert ([32], [2]). Im Unterschied zur Diskussion um Software-Komponenten wird dabei in erster Linie auf fachliche Konzepte geachtet. Ein vergleichbarer Ansatz ist in der Rekonstruktion anwendungsspezifischer Fachsprachen mittels semi-formaler „Normsprachen“ [23] zu sehen. In ähnlicher Weise wird in der Künstliche Intelligenz Forschung unter dem Begriff „Ontology“ [13] an der Erstellung wiederverwendbarer, zum Teil formalisierter Begriffssysteme für bestimmte Anwendungssysteme gearbeitet.

Für den Entwurf entsprechender Referenzmodelle ist einerseits zu berücksichtigen, daß sie unterschiedliche Sichten auf ein Unternehmen unterstützen sollten (z.B. Organisationsstruktur, Geschäftsprozesse, Informationsobjekte) und Transformationen in Modelle unterstützen, die die Besonderheiten von Komponenten berücksichtigen (Vorschläge dazu finden sich in [Fer+97] und [17]). Entsprechende multiperspektivische Unternehmensmodelle bieten ein Medium für die evolutionäre Entwicklung sorgfältig überprüfter Referenzmodelle [10]. Gleichzeitig stellen sie ein Medium für den Austausch zwischen verschiedenen wissenschaftlichen Disziplinen (Wirtschaftsinformatik, Betriebswirtschaftslehre, Informatik) und der Praxis dar. Eine solche Zusammenarbeit ist auch deshalb von Bedeutung, da mit der Einführung von Referenzmodellen und zugehörigen Komponentenbibliotheken die Arbeitsteilung bei der Entwicklung und Wartung von Software

neu zu gestalten ist und dabei neue, disziplinübergreifende Qualifikationsprofile entstehen werden (vgl. dazu [34], S. 335 ff.).

Literatur

[1] Allen, P.; Frost, S.: Component-Based Development for Enterprise Systems. Cambridge 1998

[2] Becker, J.; Schütte R.: Handelsinformationssysteme. Landsberg/Lech 1996

[3] Biggerstaff, T.: A Radical Hypothesis: Reusability is the Essence of Design. In: COMPSAC 84, The IEEE Computer Society´s Eigth International Computer Software & Applications Conference. Nov. 7-9, Chicago/Ill. 1984

[4] Booch, G.: Software Components with Ada: Structures, Tools, and Subsystems. Redwood City, Ca. 1987

[5] Brockschmidt, K.: Inside OLE. 2. Aufl., Redmond, Wa. 1995

[6] Bronsard, F., Bryan, D.; Kozaczynski, W. et al.: Toward Software Plug-and-Play. In: ACM Software Engineering Notes, Vol. 22 No. 3 May 1997, S. 19-29

[7] Cheng, B.; Jeng, J.: Formal Methods applied to Reuse. In: Proceedings of the fifth Annual Workshop on Software Reuse. 1992

[8] Cox, B.J.: There is a Silver Bullet. In: BYTE, October 1990, S. 209-218

[Fer+97] Ferstl, O.K.; Sinz, E., Hammel, C.; Schlitt, M.; Wolf, S.: Applications Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Bamberger Beiträge zur Wirtschaftsinformatik, Nr. 42, Bamberg 1997

[9] Frank, U.: Framework In: Mertens, P. et al. (Hg.): Lexikon der Wirtschaftsinformatik. 3. erw. Aufl., Berlin, Heidelberg et al., S. 167 f.

[10] Frank, U.: Enriching Object-Oriented Methods with Domain Specific Knowledge: Outline of a Method for Enterprise Modelling. Arbeitsberichte des Instituts für Wirtschaftsinformatik, Nr. 4, Juli 1997

[11] Frank, U.: Delegation: An Important Concept for the Appropriate Design of Object Models. Erscheint in: Journal of Object Oriented Programming 1999

[12] Frakes, W.B.; Nejmeh, B.A.: An Information System for Software Reuse. In: Tracz, W. (Hg.): Tutorial: Software reuse - emerging technology. Washington/D.C. 1988, S. 142-151

[13] Gaines, B.R.(Hg.): Shareable and reusable ontologies: shareable and reusable problem-solving methods. Calgary 1994

[14] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Reading/Mass. et al. 1995

[15] Harold, E.R.: Java Beans. Foster City, Ca. et al. 1998

[16] Jacobson, I.; Christerson, M.; Jonsson, P.; Overgaard, G.: Object-Oriented Engineering. A Use Case Driven Approach. Reading/Mass. 1992

[17] Kalkmann, J.; Lang, K.-P.; Ortner, E.: Ein Szenario für die Anwendungsentwicklung mit Komponenten. Bericht 98/03, Fachgebiet Wirtschaftsinformatik I, Technische Universität Darmstadt 1998

[18] McIllroy, M.D.: Mass produced software components. In: Naur, P.; Randell, B. (Hg.): Proceedings of NATO Conference on Software Engineering. Garmisch 1968, S. 88-99

[19] Meyer, B.: Object-Oriented Software Construction. 2nd. Ed., Upper Saddle River, N.J. 1997

[20] Nahm, R.: Designing and Documenting Componentware with Message Sequence Charts. In: Jell, T. (Hg.): Component-Based Software Engineering. Cambridge 1998, S. 111-116

[21] Object Management Group: Common Facilities RFP-4: Common Business Objects and Business Object Facility. OMG, TC 13CF/96-01-04, 199

[22] OMG: CORBA Ships with Java 2 Platform. In: OMG News. Spring 1999, S. 6

[23] Ortner, E.: Methodenneutraler Fachentwurf. Stuttgart/Leipzig 1997

[24] Onuegbe, E.O.: Software Classification as an Aid to Reuse: Initial Use as Part of a Rapid Prototyping System. In: Tracz, W. (Hg.): Tutorial: Software reuse - emerging technology. Washington/D.C., S. 161-167

[25] Park, Y.; Ramjisingh, D.: Software component base for reuse in functional program development. In: Proceedings of the International Conference on Computing and Information. 1994, S. 91-102

[26] Park, Y.; Bai, P.: Retrieving Software Components by Execution. In: Jell, T. (Hg.): Component-Based Software Engineering. Cambridge 1998, S. 39-48

[27] Prieto-Diaz, R.: Implementing Faceted Classification for Software Reuse. In: Communications of the ACM, Vol. 33, No. 5, 1991, S. 88-97

[28] Rittri, M.: Using types as search keys in function libraries. In: Journal of Functional Programming. No. 1, 1991, S. 71-89

[29] Rogerson, D.: Inside COM. Redmond, Wa. 1997

[30] Rollins, E.; Wing, J.: Specifications as Search Keys for Software Libraries. In: Proceedings of the 8th International Conference on Logic Programming. 1991

[31] Sametinger, J.: Software Engineering with Reusable Components. Berlin, Heidelberg et al. 1997

[32] Scheer, A.-W.: Wirtschaftsinformatik. Referenzmodelle für industrielle Geschäftsprozesse. 7. Aufl., Berlin, Heidelberg et al. 1997

[33] Stal, M.; Gengenbach, C.; Czarnecki, K.: Des Knaben Wunderhorn. In: Objektspektrum, Nr. 1, 1999, S. 18-20

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

[35] Vollmar, F.; Schäckermann, F.: San Francisco Application Business Process Components. In: Smalltalk und Java in Industrie und Ausbildung, Tagungsband. Erfurt 1998, S. 299-309