Komponenten, Komponentenframeworks und Gluing[1]

Oscar Nierstrasz, Markus Lumpe
Software Composition Group, University of Berne[2]

Abstract. In der letzten Zeit wird immer häufiger von komponentenorientierter Softwareentwicklung gesprochen, wobei meistens nicht klar ist, was darunter eigentlich zu verstehen ist. Was macht ein Stück Software zur Komponente? Wir sagen, daß Softwarekomponenten in einer speziellen Art und Weise konstruiert werden müssen, um mit anderen Komponenten zu einer Applikation zusammengefügt werden zu können. Mit anderen Worten, eine Softwarekomponente ist Teil eines Komponentenframeworks, daß (i) eine Bibliothek von Black-Box-Komponenten zu Verfügung stellt, (ii) eine wiederverwendbare Softwarearchitektur definiert, in der die Komponenten geeignet integriert sind und (iii) eine bestimmte Art von Glue, die es uns erlaubt, Komponenten miteinander zu verbinden. In diesem Artikel versuchen wir, den Ist-Zustand der Komponententechnologie wiederzugeben und behaupten, daß nur eine bessere Unterstützung im Bereich Frameworks und Gluing die Komponententechnologie vorwärts bringen kann.

1. Einleitung

Noch vor ein paar Jahren wurde nur eine objektorientiert entwickelte Applikation als modern betrachtet. Allerdings beschrieb objektorientiert ein weites Spektrum der Anwendungsentwicklung, wie "in Smalltalk programmiert" bis hin zu "mit Visual Basic erstellt". Heute hat sich ein allgemeines Verständnis darüber etabliert, was objektorientierte Softwareentwicklung ausmacht. Allerdings gelang es mit dieser Methode nicht, die entscheidenden Probleme der Softwareproduktion, wie z.B. die Erhöhung des Grades der Wiederverwendbarkeit, wirklich zu lösen. Eine neue Methode, die "komponentenorientierte" Softwareentwicklung, soll nun diese Probleme einer Lösung zuführen. Aber wie bereits bei der objektorientierten Methode gibt es im Augenblick auch hier ein ziemlich diffuses Bild darüber, was komponentenorientiert wirklich bedeutet.

Ein wirklich interessantes Faktum hingegen ist, daß sowohl der Begriff "Objekt" als auch "Komponente" bereits in den 60er Jahren in die Softwaretechnik Einzug gehalten haben. Mit der Programmiersprache Simula wurde die "objektorientierte Programmierung" geboren, und M. D. McIlroy führte anläßlich einer Softwareengineering Tagung des NATO Science Commitee 1969 den Begriff "komponentenorientierte Softwarekonstruktion" ein. Was sich aber seit

1. HMD — Theorie und Praxis der Wirtschaftsinformatik, September 1997, pp. 8—23.

2. Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland.
E-mail: {lumpe,oscar}@iam.unibe.ch. WWW: www.iam.unibe.ch/~scg.

jener Zeit geändert hat, ist unsere Sicht, insbesondere unserer Erwartungen an die damit verbundenen Methoden. Für den Entwickler und Anwender von Software ist ein Objekt ein natürliches und solides Modell. Für den Entwickler stellen Objekte die geeignete Abstraktionen für die Entwicklung der Applikation dar, während der Anwender Objekte benutzt, um die Applikation zu verstehen, wobei er seine eigene konzeptionelle Sicht benutzt. Hierin liegt nun auch eines der Probleme des objektorientierten Paradigmas, ob die Eigenschaft "objektorientiert" mehr mit Programmiersprachen oder mit Benutzerschnittstellen zu tun hat. Es muß aber auch gesagt werden, daß die Verwendung von objektorientierten Prinzipien nicht notwendigerweise zu Software führt, die leicht adaptierbar, rekonfigurierbar oder wiederverwendbar ist, noch sich leicht mit anderen Applikationen kombinieren läßt. Weiterhin kommt es immer wieder vor, daß auch gut strukturierte Applikationen unter dem sogenannten "Spaghettieffekt" leiden, der seine Ursache in den komplexen, wechselseitigen Abhängigkeiten der verwendeten Klassen hat.

Die Welt hat sich in den letzten Jahren stark verändert. Sie ist kleiner geworden und unser Schreibtisch ist nun ein Fenster zur Welt und er bietet uns eine Vielzahl von neuen Diensten an. Allerdings ist es nicht immer leicht, diese Dienste so anzuwenden, wie wir es möchten, ob wir nun Entwickler oder Endanwender von Software sind. Leider helfen uns bei unserer Arbeit Objekte nicht sehr viel, da sie nicht objektorientiert genug sind. Ein Objekt ist ein Ding, das wir uns in die Tasche stecken können und mit nach Hause nehmen können. Softwareobjekte besitzen diese Eigenschaft nicht. Man kann sie nicht einfach aus ihrer Umgebung herauslösen, ohne sie dabei zu zerstören. Was wir also brauchen, sind Softwareobjekte, die dinglicher sind als bisher, die in einem System zusammenarbeiten können und die wir wie Bausteine zusammenfügen können. Was wir brauchen, sind "Komponenten".

Komponenten definieren weder einen neuen Objektbegriff, noch ersetzen sie Objekte in ihrer bisherigen Bedeutung. Unter Komponenten sollten wir vielmehr eine neue Art und Weise verstehen, mit der man objektorientierte Softwareentwicklung betreibt. Mit anderen Worten, eine gut strukturierte, flexible, objektorientierte Applikation ist auch komponentenorientiert. Der Grundgedanke, den wir an dieser Stelle weiterentwickeln möchten, ähnelt dem der Komposition von Musik. Komponenten repräsentieren Softwareabstraktionen, die wie Noten (auf einem Notenblatt) kombiniert bzw. komponiert werden können. Das Ziel dabei ist, ein neues Gesamtwerk (Applikation) zu schaffen. Eine Komponente kann daher auch nicht losgelöst vom ihrer Umgebung betrachtet werden. Sie gehört immer zu einem bestimmten System (Framework), wie auch bestimmte Noten zu einer Tonart gehören.

Im weiteren wollen wir, beginnend mit einer allgemeinen Definition von Komponenten, eine Reihe bereits existierender Komponentenmodelle, Frameworks und Gluing-Mechanismen analysieren und dabei versuchen, aktuelle Trends und Probleme näher zu beleuchten.

Wir können nicht auf alle bereits existierenden Komponentenmodelle eingehen. Wir werden bestimmte exemplarisch verwenden, um spezifische Eigenschaften und Probleme näher zu erläutern. Die wichtigsten Modelle, auf die wir uns beziehen wollen, sind OpenDoc von Apple und IBM [4], COM/OLE/ActiveX von Microsoft, Delphi von Borland [2] und JavaBeans von JavaSoft [3].

2. Was sind Komponenten?

Eine "Softwarekomponente" ist ein "Element eines Komponentenframeworks".

Obwohl diese Definition rekursiv ist, beschreibt sie jedoch gerade dadurch die entscheidenden Eigenschaften von Komponenten. Komponenten eines Hifi-Systems oder eines modularen Möbelsystems sind nicht einfach nur deshalb Komponenten, weil sie so entworfen wurden. Auch wenn jede einzelne Komponente eine eigene Funktionalität besitzt und man sie individuell benutzen kann, so liegt doch ihr wahrer Wert in der Fähigkeit, mit anderen Komponenten kombiniert werden zu können. Eine Komponente, die zu keinem übergeordneten System gehört, oder die nicht mit anderen Komponenten kombiniert werden kann, ist ein Widerspruch in sich selbst.

Eine Softwarekomponente ist eine “statische Softwareabstraktion mit Plugs”.

Unter einer Softwareabstraktion verstehen wir mehr oder weniger eine sogenannte “Black-Box”. Die Haupteigenschaft dieser Black-Box ist, daß sie die Implementationspezifika vor dem Benutzer verbirgt. Die Eigenschaft “statisch” bedeutet in diesem Zusammenhang, daß wir diese “Black-Box” in einer geeigneten Weise instanziieren müssen, bevor wir sie benutzen können. In objektorientierten Programmiersprachen (sofern sie die Abstraktion Klasse unterstützen) unterscheiden wir zwischen “Klassen” und “Objekten”, wobei Objekte Instanzen einer Klasse sind. Leider gibt es für Komponenten keine derartige Unterscheidung. Wenn wir mit Komponenten arbeiten, benutzen wir meistens Instanzen von Komponenten. Dies ist im wesentlichen dadurch begründet, daß wir entweder gerade mit einem Komponentenframework (im weitesten Sinne handelt es sich hierbei um eine komponentenorientierte Entwicklungsumgebung) arbeiten, oder daß wir unsere Applikation ablaufen lassen. Selbst als Komponentenentwickler benutzt man eher die Instanzensicht, um die Komponente besser auf ihre spezifische Benutzbarkeit auszurichten.

Unter Plugs verstehen wir die Softwareschnittstelle einer Komponente Diese Schnittstelle definiert aber nicht nur ein Serviceinterface, daß eine Komponente zur Verfügungen stellt, sondern sie definiert auch die Dienste, die eine Komponente von ihrer Umgebung erwartet. Der beste Vergleich in diesem Zusammenhang ist der von Stecker und Steckdose. Sowohl Stecker als auch Steckdose sind Plugs, mit deren Hilfe wir Komponenten zusammenfügen können. Das Interface von Stecker und Steckdose ist aber länderspezifisch und die Art und Weise der Verbinder gibt Auskunft über Bedingungen, unter denen eine Verbindung möglich und sinnvoll ist. Eine echte “Black-Box-Komponente” benutzt ausschließlich “public Interfaces” um ihre Dienste und Anforderungen bekannt zu machen. Es gibt keine versteckten Abhängigkeiten.

Plugs sind die entscheidende Voraussetzung für die Komposition von Komponenten. Allerdings sind die Arten von Plugs und wie eine Komponente über diese Plugs mit anderen Komponenten verbunden werden kann, im wesentlich von ihrem Komponentenframework abhängig und unterscheiden sich im allgemeinen von Komponentenframework zu Komponentenframework.

Ein Komponentenframework besteht aus einer Bibliothek von Komponenten und einer zugehörigen Softwarearchitektur, welche die Basiseigenschaften der Plugs und die Art und Weise der Komposition festlegt.

Diese Definition schließt den Kreis. Der entscheidende Punkt ist, daß wir Komponenten nicht isoliert betrachten können. Man muß sie immer im Zusammenhang mit der ihnen zugrunde liegenden Softwarearchitektur des Komponentenframeworks sehen. Die Architektur und die Kompositionsregeln eines Komponentenframeworks hängen von einer Vielzahl von Fakto-

ren ab. Ein Kriterium ist z.B. die Granularität der Komponenten (Stellen Komponenten nur einfache Dienste zur Verfügung, implementieren sie ein kompliziertes Protokoll oder adressiert ein Komponentenframework ein spezifisches Anwendungsgebiet?).

Was wir im Augenblick noch nicht richtig definieren können, ist der Begriff “Glue” (In einigen Fällen spricht man auch “Scripting”. Scripting-Sprachen zur Komposition von Komponenten sind unter anderem “AppleScript” und “Visual Basic”.). Glue bedeutet übersetzt "Klebstoff" und besitzt in der Regel eine amorphe Struktur. Software-Glue besitzt diese Eigenschaft ebenfalls. Einige Arten von Glue können nur in bestimmten Situationen und für bestimmte Materialien benutzt werden, andere hingegen erlauben es, Objekte in Strukturen einzubinden, die a priori überhaupt nicht dafür entwickelt wurden. Software-Glue kann in ähnlicher Weise für eine Vielzahl von Anwendungsfällen benutzt werden.

3. Componentware in der Praxis

Eine Klassifikation für Componentware aufzustellen ist nicht einfach. Es existiert eine Vielzahl von Techniken, Mechanismen und Verfahren, so daß der Versuch, verschiedene Komponentensysteme miteinander zu vergleichen, notwendigerweise dazu führt, daß bestimmte Eigenschaften mehr Beachtung finden, während andere vollständig außer acht gelassen werden.

Aus diesem Grund werden wir hier keine Klassifikation von Componentware aufstellen. Wir werden vielmehr versuchen, Komponenten, Frameworks und Glue in ihren gemeinsamen Eigenschaften, ihren Unterschieden, den aktuellen Trends ihrer Entwicklung und den mit ihrer Benutzung verbundenen Probleme darzustellen. Mit diesem Ansatz hoffen wir, ein allgemeines Bild von Componentware darstellen zu können, das den “State-of-the-art” in Componentware widerspiegelt.

3.1 Komponenten

Die bereits existierenden Komponentenmodelle zeichnen sich durch eine Reihe gemeinsamer Eigenschaften aus, die man als Standardeigenschaften vom Komponenten bezeichnen kann. Leider decken diese Eigenschaften nur einen kleinen Teil aller Variablen ab. Dies mündet in der Tatsache, daß die verbleibenden Unterschiede von Komponentenmodell zu Komponentenmodell noch enorm sind.

Im Zusammenhang mit Komponenten stellen sich uns nun viele Fragen: Welche Granularität besitzen Komponenten? Liegen sie in Quellform oder als vorkompilierte Einheit vor? Sind sie plattformunabhängig oder plattformabhängig? Sind es “White-Box” oder “Black-Box” Einheiten? Besitzen sie einen Zustand? Gibt es Interfacestandards für Komponenten?

3.1.1 Granularität

Die eigentliche Frage in diesem Zusammenhang ist, ob Komponenten größere oder kleinere strukturelle Einheiten repräsentieren als Objekte, oder ob Komponenten vielleicht die gleiche Größe haben wie Objekte. Und die noch wichtigere Frage ist, ob sie auch ganze Applikationen repräsentieren können.

Wenn wir uns die oben gegebenen Definition näher ansehen, wird offensichtlich, daß Komponenten von einer beliebigen Granularität sein können. Mixins sind z.B. Komponenten gerin-

ger Granularität (fine grained) (Mixins werden mit Hilfe von Mehrfachvererbung aus unterschiedlichen abstrakten Teilklassen konstruiert. Dabei stehen die Basisklassen in keiner Beziehung zueinander). Eine weitere Klasse von Komponenten dieser Kategorie sind einfache Dialogelementkomponenten wie Buttons oder Labels, wie sie in den Komponentensystemen von Delphi oder JavaBeans zum Einsatz kommen. Diese haben nur eine geringe Funktionalität, bzw. reagieren nur auf wenige Ereignisse. Ein weiteres Beispiel sind Komponenten, die mit dem Component Object Model (COM) von Microsoft implementiert wurden. Dieses Model empfiehlt ausdrücklich, Komponenten möglichst klein und einfach zu gestalten [1]. Je einfacher die Interfaces der Komponenten sind, desto besser eignen sie sich für die Komposition.

Komponenten mit hoher Granularität (coarse grained) können ganze Applikationen repräsentieren. Ein Beispiel dafür sind die Parteditoren von OpenDoc oder die Einbettung von Exceldateien in Microsoft Word mit Hilfe von OLE (jetzt ActiveX). Unter UNIX kann man Filterprogramme wie z.B. grep, awk, sed als Komponenten betrachten. Durch die Verknüpfung der Filterprogramme mit dem Pipe-Operator ("|") kann man komplexeste Textmanipulationen konstruieren.

Komponenten mit höherer Granularität eignen sich auch besonders für verteilte Applikationen, wobei die Komponenten auf verschiedenen Knoten in einem verteilten System liegen können. Ein Beispiel dafür sind Client/Server Anwendungen.

Mit DCOM (Distributed Component Object Model) von Microsoft, das unter Windows NT4.0 und Windows 95 zu Verfügung steht, kann man ebenfalls Applikationen mit verteilten Komponenten erstellen. Dabei ist DCOM eine Erweiterung von COM. DCOM unterstützt dabei sowohl Multithreading als auch echt verteilte Komponenten. Für den Benutzer ist aber der Mechanismus transparent. Die zugehörigen Werkzeuge (z.B. MIDL der IDL-Compiler von Microsoft) unterstützen dabei den Applikationsprogrammierer entscheidend, da sie die meisten systemabhängigen Operationen, wie Netzwerkkommunikation und Datenkonvertierung, automatisieren.

3.1.2 Sourcecode oder vorcompiliert?

Die meisten existierenden Komponentenmodelle unterstützen vorcompilierte Komponenten. Diese Verfahrensweise orientiert sich an der "Black-Box"-Strategie. Komponenten werden dann in der Regel in Komponentenbibliotheken integriert. Bei COM bzw. OLE/ActiveX kommen hierbei DLL's (Dynamic Link Library) zum Einsatz (Es gibt auch die Möglichkeit, Komponenten in anderen Applikationen zu implementieren. Für den Benutzer ändert sich dabei allerdings kaum etwas. Der Hauptunterschied zur Verwendung einer DLL ist, daß die Komponenten in einem anderen Adressraum als die Applikation ausgeführt werden.). DLL's können eine oder mehrere Komponenten aufnehmen. Die DLL wird dann zur Laufzeit geladen. Das Laden ist Teil des COM-Systems. Der Komponentenentwickler ist dafür verantwortlich, daß die DLL die nötigen Prozeduren zu Verfügung stellt, die für das Laden von Komponenten nötig sind.

In Delphi und JavaBeans kommt ein ähnlicher Ansatz zum Einsatz, wobei die Komponentenbibliothek von Delphi eher den Charakter einer gewöhlichen Bibliothek (.lib) hat, da verwendete Komponenten zur Applikation gelinkt werden. (Dies ist einer der großen Nachteile von Delphi-Applikationen, da durch diese Vorgehensweise wiederum große monolithische

Applikationen entstehen.) JavaBeans verwendet Archivdateien (.jar), die zur Laufzeit der Applikation geladen werden, um die darin enthaltenen Komponenten (Beans) zu benutzen. Zu einer Komponente gehören eine oder mehreren Klassen.

Eine krasse Ausnahme bilden in diesem Zusammenhang C++-Templates, insbesondere die Standard Template Library (STL), die eine Sammlung von Containerklassen, Basisalgorithmen und Datenstrukturen zur Verfügung stellt. Die Ausnahme wird nun dadurch begründet, daß für C++-Templates Code generiert wird, wenn sie das erste Mal benutzt werden. Eiffel und Ada unterstützen ebenfalls eine ähnliche Art generischer Komponenten, aber im Unterschied zu C++ wird für diese kein zusätzlicher Code generiert. Man kann solche Komponenten dann auch als “Black-Box” Klassen ansehen, obwohl sie in Quellform vorliegen.

3.1.3 Komponentenmodell: homogen oder heterogen?

Komponenten mit einer geringen Granularität besitzen eher eine homogene Struktur. Ein wesentlicher Grund dafür ist, daß sie in der Regel für eine bestimmte Programmierumgebung entwickelt wurden, wie z.B. Smalltalk, Oberon, Eiffel, Delphi oder Java. Weiterhin besitzen solche Komponenten oftmals ein eher prozedurales Interface. Ihr Konfigurationsinterface ist nur schwach entwickelt und besitzt meistens nur 1-2 konfigurierbare Eventhandler. (Bei COM hat man eigentlich überhaupt kein Konfigurationsinterface, da alle Operationen erst zu Laufzeit ausgeführt werden.)

Ein typisches Beispiel für solche Komponenten sind wiederum Dialogelementkomponenten wie Buttons oder Labels. Ihre Struktur ist homogen, da sie für ein spezielles Komponentensystem erstellt wurden und ihr Interface verfügt nur über ein kleines Konfigurationsinterface. Die Aktionen, die ausgeführt werden sollen, wenn ein Button gedrückt wurde, müssen vom Programmierer spezifiziert werden. Im Falle von Delphi und JavaBeans wird dafür dieselbe Programmiersprache verwendet, mit der schon die Komponenten erstellt wurden.

Bei komplexen (coarse grained) Komponenten tritt das prozedurale Interface eher in den Hintergrund. Diese Komponenten definieren andere Arten von Interfaces, wie Streams, Events, oder RPC (Remote Procedure Call). Typische Vertreter dieser Klasse von Komponenten sind UNIX-Filter. Filter lesen vom einem Eingabestream und schreiben auf einen Ausgabestream. Verschiedene Filter werden mit dem Pipe-Operator verknüpft. Die dadurch entstehenden Filtersequenzen können die verschiedensten Aufgaben erfüllen. Das Hauptanwendungsgebiet dieser Art der Komposition ist die Textmanipulation. Die resultierenden Ergebnisse sind oftmals sehr verblüffend. Allerdings wird bei dieser Art der Komposition nicht überprüft, ob das Ergebnis des einen Filters ein zulässiger Input für den nächsten Filter ist. Der Anwender ist dafür verantwortlich, daß die Filter in der richtigen Reihenfolge angeordnet werden und die richtige Konfiguration erhalten. Ist nur ein Filter falsch konfiguriert, wird das gesamte Ergebnis verfälscht. Man hat keine Möglichkeit, vor der Anwendung der Filtersequenz zu überprüfen, ob die Konfiguration korrekt ist (gemeint ist hier eine Art statischer Typprüfung).

Heterogene Komponentenmodelle sind z.Zt. eher ungebräuchlich. Delphi ist ein System, das die Integration von Fremdkomponenten unterstützt. Für die Fremdkomponenten werden Delphi-Proxy-Komponenten erstellt. Für den Benutzer erscheinen diese Komponenten so, als wären sie mit Delphi selbst implementiert worden. Der einzige Unterschied zu Delphi-Komponenten ist, daß sie nicht zur Applikation gebunden werden. Im Falle von Visual Basic VBX/

OCX-Komponenten müssen diese außerdem noch im System registriert worden sein (entweder im Verzeichnis SYSTEM von Windows vorhanden sein, oder in der Registrierdatenbank des Systems eingetragen worden sein). Ein anderes Modell für heterogene Komponenten ist DCOM. Hier können Komponenten auf verschiedenen Rechnern unterschiedlichster Architektur kommunizieren, sofern DCOM auf der entsprechenden Plattform unterstützt wird. Für die korrekte Kommunikation und Datenkonvertierung sorgt DCOM.

Ob die Struktur von Komponenten eher homogen oder heterogen ist, hängt zu einem großen Teil vom Problembereich ab, für den sie bestimmt sind. Allerdings geht die Tendenz eher in Richtung homogener Komponentenstruktur. Nicht zuletzt durch die Definition von Industriestandards wie CORBA oder COM/ActiveX kommt einer homogenen Struktur eine immer größere Bedeutung zu. Allerdings schließt diese Entwicklung die Verwendung von heterogen Komponenten nicht aus. So definiert die Spezifikation von JavaBeans, daß das dort benutzte Komponentenmodell auf die plattformspezifischen Komponentenmodelle abgebildet werden kann, wobei die Abbildung für den Benutzer transparent sein muß.

3.1.4 "White-Box" oder "Black-Box" Komponenten?

Im Zusammenhang mit Komponenten betrachten wir das "Black-Box" Modell als das beste Modell. Es muß allerdings auch gesagt werden, daß dieses Modell nicht dogmatisch angewendet werden sollte. Auch der Mensch braucht eine geeignete Umgebung, um zu funktionieren. Ohne die entsprechenden Hilfsmittel ist es uns unmöglich, auf dem Mond oder auf dem Grund des Ozeans zu existieren. Komponenten verhalten sich in dieser Beziehung ähnlich. Smalltalk-Objekte können nicht einfach aus der Smalltalk-Umgebung herausgelöst werden. Es bedarf spezieller Mechanismen, um eine solche Operation zu ermöglichen.

Aber auch dann, wenn eine Abbildung von einem System in ein anderes definiert ist, müssen wir sehr viel Sorgfalt walten lassen, um alle Voraussetzungen bzw. Nebenbedingungen zu erfüllen. Smalltalk- und C++-Komponenten können a priori nicht vernünftig zusammenarbeiten. Der Garbagecollector von Smalltalk muß in diesem Falle darüber informiert werden, daß C++-Objekte Smalltalk-Objekte referenzieren können. Weiterhin muß man ihm mitteilen, welche C++-Objekte Smalltalk-Objekte referenzieren.

Auch in einem homogenen Umfeld sind versteckte Abhängigkeiten keine Seltenheit. Objekte und Prozeduren können oftmals nur richtig arbeiten, wenn sie vorher initialisiert worden sind. Die Ausdruckskraft eines prozeduralen Interfaces reicht in der Regel nicht aus, diese Art von Abhängigkeiten auszudrücken. Nur einige wenige Komponentenmodelle versuchen hier eine Lösung zu schaffen.

Dieses Problem wird im Falle von COM noch viel größer. Hier spielt die Initialisierung eher eine untergeordnete Rolle. Was vielmehr ins Gewicht fällt, ist das "Reference Counting" für Interfacezeiger. Wenn der Komponentenanwender die verwendeten Interfaces nicht korrekt zählt (aufrufen der Funktionen AddRef und Release), dann kann es passieren, daß Komponenten nie aus dem Speicher entfernt werden, oder was noch viel problematischer ist, Komponenten zu früh aus dem Speicher entfernt werden. Es gibt gegenwärtig noch keine sichere Methode, diese Probleme zu lösen. In Form von Klassenbibliotheken (Microsoft Foundation Classes für C++) gibt es zwar schon Unterstützung für die Behandlung des "Reference Counting", allerdings

kann es immer noch zu den genannten Problemen kommen (zirkuläre Verwendung von Interfaces in zwei Komponenten).

Es kann eigentlich nicht die Aufgabe des Komponentenanwenders sein, alle notwendigen Initialisierungen bzw. Rahmenbedingungen für den Einsatz bestimmter Komponenten zu garantieren. Hier ist der Komponentenentwickler als auch das Komponentenframework gefragt. Die meisten Komponentenmodelle verlangen vom Komponentenentwickler, daß er seine Komponenten so programmiert, daß es keine Abhängigkeiten gibt, die nach außen gereicht werden. Hier ist ein besserer Ansatz nötig, der durch das Komponentenframework hinreichend unterstützt werden muß.

3.1.5 Komponenten mit Gedächtnis

Am Abschnitt 2 haben wir gesagt, daß Komponenten “statische Softwareabstraktionen” sind. Daraus folgt, daß sie keine Erinnerungsvermögen bzw. Gedächtnis besitzen. Technisch gesehen würde das heißen, daß Komponenten keinen Zustand besitzen. Viele Komponentenumgebungen verwischen aber den Unterschied zwischen Komponenten und ihren Instanzen. In einer visuellen Programmierumgebung haben Komponenten wie Buttons oder Labels einen Zustand. Dieser widerspiegelt exakt, welches Erscheinungsbild diese Komponenten haben und auf welche Ereignisse sie reagieren. Typische Zustandseigenschaften sind Position, Farbe und Beschriftung, bzw. Button gedrückt.

Die Ursache für diesen Widerspruch liegt in der Tatsache, daß wir eigentlich nicht mit Komponenten (Klasse Button), sondern mit ihren Instanzen arbeiten. Interfacebuilder benutzen in der Regel konkrete Instanzen, während Programmiersprachen mit Klassen arbeiten. Allerdings unterscheiden Interfacebuilder typischerweise zwischen Entwurf und Laufzeit. Es gibt eine Reihe von Komponentenfunktionen, die nur beim Entwurf benötigt werden und andere, die nur zur Laufzeit ausgeführt werden können. Einige Komponentenmodelle definieren sehr konkrete Anforderungen an Komponenten sowohl für den Entwurf als auch für die Laufzeit.

JavaBeans und Delphi sind die besten Beispiele für Komponenten, die über einen Zustand verfügen. Mit Hilfe von sogenannten “Properties” kann eine Komponente in diesen Systemen anwendungsspezifisch konfiguriert werden. Beide Systeme benutzen hierfür eine spezielle Datei, um die Konfiguration zu speichern. Damit ist es möglich, die meisten anwendungsspezifischen Eigenschaften bereits beim Entwurf der Applikation festzulegen, ohne dabei zusätzlichen Code programmieren zu müssen (die Ausnahme sind dabei Eventhandler). Die Konfiguration wird dann zur Laufzeit wieder geladen. Das Komponentenmodell sieht einen entsprechenden Mechanismus vor, der nicht vom Programmierer aktiviert werden muß.

Damit man Komponenten möglichst einfach und effektiv konfigurieren kann, bieten moderne Umgebungen “Property-Editoren” an. Property-Editoren sind selbst wieder Komponenten, aber im Unterschied zu den Applikationskomponenten dienen sie nur zur Konfiguration und werden nicht zur Applikation gebunden. Damit besteht eine Komponente nicht nur aus der operationstragenden Einheit, sondern eine Komponente ist in Wahrheit ein ganzes Paket von Komponenten, wobei die meisten nur beim Entwurf benutzt werden.

3.1.6 Meta-Komponenten

Im letzten Abschnitt sind wir kurz auf “Property-Editoren” eingegangen. Diese Komponenten sind eigentlich “Meta-Komponenten”. Meta-Komponenten dienen ähnlich wie Meta-Objekte zur Beeinflussung des Verhaltens von Komponenten und ihrer Umgebung. Die meisten Komponentenmodelle unterstützen kein explizites Meta-Modell, besitzen aber Ausdrucksmittel, Meta-Komponenten zu konstruieren.

Die einfachste Möglichkeit, Meta-Komponenten in die Umgebung zu integrieren, bietet Delphi. Da Delphi mit Delphi implementiert wurde, bietet es die besten Voraussetzungen, das Verhalten der Delphi-Umgebung zu modifizieren. So kann man beispielsweise Meta-Komponenten integrieren, welche die automatische Benennung von Komponenten verändern, oder man kann seine eigenen Inspektoren (ähnlich dem Standardobjektinspektor) definieren. Damit wird Delphi zu einer offenen Umgebung, die es gestattet, beliebige Erweiterungen in das System zu integrieren. Die einzige Voraussetzung ist, daß man die Komponentenbibliothek recompiliert.

3.1.7 Standardinterfaces

Ein großes Problem, sowohl in der objektorientierten als auch bei der komponentenorientierten Programmierung, ist die Standardisierung von Schnittstellen. Bei Komponenten kommt diesem Problem eine noch viel größere Bedeutung zu. Die Verwendung von Komponenten ermöglicht bei einigen Systemen unter anderem, daß Komponenten zur Laufzeit durch aktuellere Versionen ersetzt werden können. Die Applikation darf dadurch aber weder beeinflußt noch zum Absturz gebracht werden. Damit dies aber realisiert werden kann, muß die neue Komponente zumindest die Interfaces der alten Komponente zur Verfügung stellen. Dabei reicht es nicht aus, daß die Spezifikationen übereinstimmen, auch das Laufzeitverhalten muß gleich sein.

Es gibt nur wenige Systeme, die Standardinterfaces unterstützen. Komponenten, die mit COM erstellt wurden, besitzen diese Eigenschaft. COM verlangt von einem publizierten Interface, daß es für alle Zeit unveränderlich ist. Dadurch wird garantiert, daß eine Applikation, die ein bestimmtes Interface benutzt, immer die gleiche Funktionalität erhält, wenn sie dieses Interface benutzt. COM benutzt dafür sogenannte “globally unique identifier”. Zu einem Interfacenamen gehört ein eineindeutiges Interface und damit immer die gleiche Implementation. Mit diesem Schema wird garantiert, daß wenn eine Applikation nach einem bestimmten Interface nachfragt, sie genau dieses erhält. Kann das gewünschte Interface nicht bereitgestellt werden, so wird ein Nullzeiger zurückgegeben.

Was es bedeutet, keine Standardinterfaces zu haben, wird am Beispiel von Delphi deutlich. Verschiedene Versionen von QuickReport sind untereinander nicht kompatibel. So wurden zwischen einigen Versionen bestimmte Properties als auch Methoden- und Eventspezifikationen verändert. Dadurch entsteht für der Applikationsprogrammierer ein erheblicher Aufwand, der eigentlich durch die Verwendung von Komponenten nicht mehr nötig sein sollte. Was hier fehlt, ist ein Schema wie bei COM, das es verbietet, publizierte Komponenten in ihrem publizierten Interface zu modifizieren.

3.1.8 Versionsverwaltung

Die Versionsverwaltung ist eng mit der Standardisierung von Komponenteninterfaces verbunden. Neuere Versionen einer Komponente dürfen vorher gültige Spezifikationen nicht verletzen bzw. verändern. Applikationen, die das alte Interface einer Komponente erwarten, benutzen auch das alte Interface der neuen Komponente, sofern sie das alte Interface unterstützt. Neue Applikationen, die Kenntnis vom neuen Interface der Komponente haben, können dann beide Interfaces benutzen.

Bei COM bedeuten neue Versionen eines Interfaces faßt immer die Einführung eines neuen Interfacenamens. Dies garantiert, das alte Applikationen weiterhin stabil laufen, auch wenn sie die neuere Version einer Komponente benutzen.

3.2 Frameworks

Ein Framework, in seiner allgemeinsten Bedeutung, ist eine Struktur bzw. das Skelett eines Projektes. Ein objektorientiertes Framework ist eine Sammlung von Klassen, die das Skelett definieren. Daraus folgt, daß diese Klassen ebenfalls die Architektur der Applikation bestimmen. Ein Komponentenframework definiert die Architektur der Applikationen durch die Instanziierung von Softwarekomponenten, aus denen die Applikation konstruiert wird.

Erst Frameworks verleihen Komponenten eine Bedeutung. Mit anderen Worten, eine Komponente, die zu keinem Framework gehört, ist keine Komponente. Betrachten wir z.B. Interfacekomponenten. Ihr eigentlicher Wert liegt nicht in der von ihnen zur Verfügung gestellten Funktionalität, sondern vielmehr in der Tatsache, daß sie so entworfen wurden, daß sie mit anderen Komponenten zusammenarbeiten können und somit in ihrer Gesamtheit viel mächtigere Benutzerinterfaces bilden können.

Nicht alle Frameworks sind objektorientiert. Objektorientierte Frameworks benutzen oftmals Vererbung als Basisprinzip für die Applikationsentwicklung. Vererbung ist eine typische Form von “White-Box-Reuse”. Der Applikationsprogrammierer muß meistens in den Frameworkcode hineinschauen, um das Framework angemessen und mit dem größtmöglichen Nutzen zu verwenden. Die Ableitung von neuen Klassen ist in der Regel nur möglich, wenn der Applikationsprogrammierer intime Kenntnisse von der Implementation der Basisklassen besitzt.

In Bezug auf moderne Frameworkentwicklung hat eine Trendwende eingesetzt. Man versucht zunehmend sogenannte “Black-Box-Frameworks” zu entwickeln. In diesen Frameworks benutzt man vorrangig die Komposition von Objekten. Das Prinzip der Vererbung tritt hier in den Hintergrund. Delphi und JavaBeans sind typische Vertreter dieser Klasse.

In Delphi basiert die visuelle Programmierung auf sogenannten “Forms” (wie Visual Basic). Ein Form ist selbst eine Komponente. Der Applikationsprogrammierer erstellt die Anwendung mittels “Drag und Drop” von Komponenten aus der Komponentenpalette in die Formkomponente. Im eigentlichen Sinne handelt es sich hierbei um Komposition. Vererbung wird hier nur im Zusammenhang mit dem gerade erstellten Form eingesetzt. Ein neues Form wird immer von der Basisklasse für Formkomponenten abgeleitet. Der Programmierer kommt aber mit dieser Vererbung nur insoweit in Berührung, als daß er bei der Definition von Eventhandlern auch auf das geschützte (protected) Interface der Basisklasse zugreifen kann. Im eigentlichen visuellen

Programmierprozeß kann man das Wissen um die implizit angewendete Vererbung aber vernachlässigen. Man programmiert im Black-Box Stil.

Die Spezifikation von JavaBeans legt ein ähnliches Verfahren fest, stellt aber ausdrücklich fest, daß es Aufgabe des Interfacebuilders ist, das geeignete Programmierparadigma zu unterstützen.

Am allgemeinsten stellt sich der Black-Box Stil bei COM dar. Hier liegt das Hauptaugenmerk auf der Spezifikation von Komponenteninterfaces. Eine Wiederverwendung von Code durch Vererbung gibt es nicht. Um bereits existierende Funktionalität wiederzuverwenden muß man Komposition verwenden. Man kann aber zur Definition von Interfaces Vererbung benutzen. Dieses Verfahren entspricht dem von Java.

COM unterstützt zwei verschiedene Arten von Komposition: “Containment” und “Aggregation”. Containment bedeutet, daß eine äußere Komponente einen Zeiger auf eine innere Komponente besitzt und die Dienste der inneren Komponente über ein eigenes Interface nach außen reichen kann. Dieser Mechanismus entspricht am ehesten dem von Interfacewrappern. Bei Aggregation besitzt die äußere Komponente ebenfalls einen Zeiger auf die innere Komponente. Aber im Gegensatz zu Containment wird das Interface der inneren Komponente direkt nach außen gereicht, ohne daß die äußere Komponente darauf Einfluß nimmt.

3.3 Gluing

Komponenten und Frameworks reichen allein nicht aus. Wir benötigen noch einen Mechanismus, um Komponenten miteinander zu verbinden. Diesen Mechanismus bezeichnet man als “Gluing” oder “Component Scripting” (im weiteren verwenden wir eher den Begriff Scripting als Gluing). Dieser Mechanismus kann viele Formen annehmen. Diese Formen sind im wesentlichen von der Natur und der Granularität der Komponenten, von der Natur und Domäne des Problembereiches, sowie vom unterstützten Kompositionsmodell abhängig. Komposition (engl. auch gluing) kann zu verschiedenen Zeitpunkten erfolgen. Es sind sowohl Compile-Time, Link-Time als auch Run-Time Modelle möglich. Komposition kann aber auch sehr restriktiv und statisch arbeiten, wie bei der Expansion von C++-Templates. Es gibt aber auch sehr flexible und dynamische Modelle, wie daß in Tcl (Tool Command Language) oder Visual Basic der Fall ist.

Scripting ist kein neuer Mechanismus. Unter UNIX ist er Arbeitsprinzip. Viele Programme sind in Wahrheit Shellscripts, die in jedem Bereich des Systems eingesetzt werden. Heutige Scriptingsprachen haben viel gemeinsam mit der Shellprogrammierung.

Eine der populärsten Scriptingsprachen im PC-Bereich ist Visual Basic. Sie ist im Prinzip der Wegbereiter für die komponentenoriertierte Programmierung unter Windows. Da Visual Basic eine Scriptingsprache ist und eine Implementierung von Komponenten in Visual Basic bisher nicht möglich war, ist es aber auch nicht verwunderlich, daß die meisten Visual Basic Komponenten in C oder C++ geschrieben sind. Allerdings hat Visual Basic als Sprache das Komponentenmodell maßgeblich beeinflußt. So sind die unterstützten Typen im Komponentenmodell auf Basic zugeschnitten, wie z.B. BSTR (Basic String).

Visual Basic benutzt einen dynamischen Typmechanismus (dynamische Typen sind in Scriptingsprachen üblich). Eine Typüberprüfung wird zur Laufzeit durchgeführt. Dieses Sy-

stem führt in Visual Basic aber auch dazu, daß nicht alle Typen unterstützt werden. Nur Typen, für die ein sogenannter “VARIANT Tag” existiert, können benutzt werden. Dies ist aber in Bezug auf die Visual Basic Controls ein kleiner bzw. nicht vorhandener Nachteil.

Delphi und JavaBeans benutzen eine objektorientierte Programmiersprache für die Komposition. Komponenten und Komposition werden mit derselben Sprache beschrieben. Allerdings hat dies auch zur Folge, daß die Beschreibung restriktiver ist, da die Typüberprüfung bereits zur Compile-Time erfolgt. Der Vorteil ist, allerdings in Java durch Interpretation des Codes abgeschwächt, daß die Applikationen in der Regel schneller ablaufen, als in reinen Scriptingansätzen.

Das Kernproblem der komponentenorientierten Programmierung ist aber, daß es z.Zt. noch kein allgemeines Verständnis darüber existiert, was Scripting von Komposition ist. Dieses Gebiet erfordert noch einige Jahre intensivster Forschung.

4. Trends, Probleme und die Zukunft

Trotz anderer Voraussagen hat sich seit Ende der 80er noch kein ein echter Komponentenmarkt etabliert. Softwarekomponenten können nicht in der gleichen Art und Weise vertrieben werden, wie z.B. CD’s. Eine Ursache dafür ist, daß sich Standards für Komponenteninterfaces gerade erst etablieren (COM ist ein solcher binärer Standard). Ein anderes Problem ist, daß es noch kein geeignetes kommerzielles Model gibt, mit Komponenten Geld zu verdienen. Brad Cox, der Erfinder von ObjectiveC, formulierte es einmal sinngemäß so: Wenn ich eine Maschine produzieren möchte, kann ich mir die nötigen Einzelteile dafür kaufen. Wenn die Maschine fertiggestellt ist, sind auch die Einzelteile verarbeitet, eine weitere Maschine zu bauen verlangt, daß ich neue Einzelteile kaufe. Entwickle ich hingegen Software, so kann ich mir einmal die notwendigen Softwarekomponenten kaufen und sie durch einfaches Kopieren so oft wie nötig wieder verarbeiten, ohne dafür erneut Geld zu bezahlen. Aus Sicht des Applikationsprogrammierers ist das kein schlechtes Szenario, aber für den Komponentenentwickler stellt dieses ein erhebliches Investitionshemmnis dar, da er nur einmal aus einer verkauften Komponente Wert schöpfen kann. In diesem Zusammenhang ist ActiveX zu nennen, da man hier versucht, eine Lizenzverwaltung in das Komponentenmodell zu integrieren.

Es gibt noch ein weiteres Problem, daß eine nicht zu unterschätzende Rolle spielt. Oftmals sind die Entwickler von Komponenten überhaupt nicht daran interessiert, ihre Komponenten zu vermarkten. Das in den Komponenten steckende Know-How stellt ein strategisches Potential für eine Firma dar, das sie benutzt, um schneller auf Veränderungen am Markt zu reagieren, bzw. ganze Produktfamilien zu vermarkten und damit einen Vorteil zu erzielen [5].

Obwohl gute Komponentenframeworks mit ihren vielen Werkzeugen einen maßgeblichen Produktivitätsgewinn bedeuten, ist die Entwicklung solcher Frameworks eher noch eine Art schwarze Kunst. Eine der Schwierigkeiten ist, daß die meisten modernen Softwareentwicklungsmethoden zu stark auf die Einhaltung gegebener Anforderungen reflektieren, aber Techniken zur Entwicklung von wiederverwendbaren und generischen Komponenten außer acht lassen. Diese Situation verbessert sich erst allmählich, insbesondere dadurch, daß wir ein besseres Gefühl dafür entwickeln, wie Frameworks aufzubauen sind und neue Komponentenmodelle immer wieder versuchen, bereits existierende Ansätze zu verbessern und zu erweitern.

Die meisten Werkzeuge in diesem Bereich adressieren überwiegend die Konstruktion von graphischen Interfaces. Diese Situation ist dadurch begründet, daß diese Werkzeuge überwiegend das visuelle Programmieren/Gestalten von User-Interfaces unterstützen. Die Gestaltung des Layouts, die Instanziierung der Komponenten und ihre Verbindung geschieht zur Designtime. Im Augenblick kann man noch nicht sagen, wie weit sich diese Tendenz fortsetzen wird, um diese Methode zu einer allgemein nutzbaren Methode für die Programmierung zu machen. Auf jeden Fall ist dieses Konzept gut geeignet, Komponenten interaktiv zu einer Applikation zusammenzufügen, allerdings muß auch gesagt werden, daß dieses Konzept nur für relativ kleine und überschaubare Anwendungen funktioniert. Für große Applikationen ist dieses Konzept ungeeignet, da es unweigerlich zu unverständlichen und unleserlichen Darstellungen auf dem Bildschirm führt.

Die Erfahrung lehrt uns weiterhin, daß sich visuelle Programmierung sehr gut für kleine Projekte eignet, hingegen für Beschreibung komplexer Algorithmen eher ungeeignet ist. In diesen Fällen ist die Formulierung der Aufgabe mit Hilfe einer geeigneten Programmiersprache immer noch die beste Lösung. Die Kombination von visueller Programmierung mit einer herkömmlichen Programmiersprache bzw. Scriptingsprache, wie dies im Fall von Visual Basic, Delphi oder auch JavaBeans der Fall ist, erscheint der richtige und notwendige Weg zu sein, um eine Entwicklungsumgebung verfügbar zu haben, die für beliebige Problembereiche eingesetzt werden kann.

Ein weiteres und ernstes Problem ist, daß es noch keine bzw. wenige allgemeine Methoden gibt, Softwarearchitekturen zu beschreiben [6]. Wir haben gesehen, daß die Softwarearchitektur ein entscheidender Bestandteil des Komponentenframeworks ist, da sie gerade die Kombinierbarkeit der Komponenten bestimmt. Allerdings ist die exakte Beschreibung der Architektur eine fast unlösbare Aufgabe. Wenn wir den Quellcode einer Applikation anschauen, können wir mit Leichtigkeit die darin enthaltenen Strukturen, wie Prozeduren und Klassen, herauslesen. Doch die zugrunde liegende Architektur bleibt uns verborgen. Solange es uns nicht gelingt, Architekturen auf einfache Art und Weise zu beschreiben, wird es schwierig sein, Komponenten effektiv zu nutzen.

Referenzen

[1] Dale Rogerson, Inside COM, Microsoft's Component Object Model, Microsoft Press, 1997

[2] Delphi Anwender Dokumentation, Borland International Inc., 1995

[3] JavaBeans 1.0 API specification, 1996 http://www.javasoft.com/beans/beans.100A.ps

[4] Feiler and Meadow, Essential OpenDoc, Cross-Plattform Development for OS/2, Macintosh, and Windows Programmers, Addison-Wesley Developers Press, Reading, Mass., 1996

[5] Kenny Rubin and Adele Goldberg, Succeeding With Objects: Decision Frameworks for Project Management, Addison-Wesley, Reading, Mass., 1995

[6] Mary Shaw and David Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996