Zusammenfassung "Komponentenbasierte Softwareentwicklung" bei Dr. Zeidler (SS 2000)

Michael Jaeger

2.10.2000

Was ist eine Komponente?

Aus Praxis, ad-hoc-Einführung

Als Einführung zu diesem Thema sollte zunächst geklärt werden, was man sich unter einer Komponente vorstellen sollte. Da der Begriff der *Komponente* sehr unpräzise ist - was vor allem daran liegt, dass er aus der Praxis kommt und nicht aus einem theoretischen Gebäude heraus entstanden ist - wurden eigentlich alle Definitionen erst nachdem der Begriff relativ ad-hoc eingeführt wurde entwickelt. Als besondere Kennzeichen einer Komponente möchte ich folgende Eigenschaften anführen:

single abstraction, zustandsbehaftet, konfigurierbar, dynamisch, reaktionsfähig

- Ein Modul mit einer *einfachen Abstraktion* (*single abstraction*), das *zustandsbehaftet*, *konfigurierbar*, *dynamisch* und *reaktionsfähig* ist.

Schnittstellen, repository, klassifizierbar, plug&play, Kooperation, Wiederverwendung

- Eine statische Abstraktion mit *Steckverbindungen* (*Schnittstellen*), die in einem *repository* gespeichert werden kann (also *klassifizierbar* ist) und *plug&play*, *Kooperation* und die *Wiederverwendung* unterstützt.

Framework, ad-hoc Interaktion

- Einzelstücke einer Komposition mit *vertraglich festgelegter Schnittstelle*, die einzeln verkauft werden können (dabei ist die interne Implementation *nicht zugänglich* - *"black-box"*), die aber wieder *Teil einer neuen Komponente* sein können.
- Ein Stück Software, das ein *Framework* berücksichtigt und *ad-hoc Interaktion* erlaubt, die bei der Entwicklung nicht unbedingt vorgesehen war.

pragmatisch, grösserer Funktionsumfang, fest umrissenere Aufgaben, mehr Abstand, skriptfähig

Im Gegensatz zu Objekten der OO erlauben Komponenten einen leichteren Übergang zu realen Gegenständen und Modellvorstellungen eines Objektbereichs. Die vorwiegend *struktuierende* Sicht der OO wird im Komponentenparadigma zwar beibehalten, jedoch zu Gunsten der Betonung einer *interagierenden* Sicht eigenständiger Softwareteile (→*pragmatische* Herangehensweise) erweitert. Im Allgemeinen haben Komponenten einen *grösseren Funktionsumfang*, *fest umrissenere Aufgaben* und *mehr Abstand zueinander* als Objekte. Auch *Skriptfähigkeit* wird als ein weiteres Abgrenzungsmerkmal gegenüber Objekten angeführt.

Kontrakt

In Zusammenhang mit Komponenten taucht immer wieder der Begriff *Kontrakt* auf, der die Spezifikation gegenseitiger Verpflichtungen auf vertraglicher Basis regelt. Der Kontrakt ist dabei die Basis der Zusammenarbeit der Komponenten und spezifiziert das Interaktionsverhalten bzgl. Rollen und verpflichtet zum Annehmen dieser Rollen.

Warum Komponenten?

Wiederverwendung, Bausteinprinzip, Zusammenstecken, legacy wrapping, erweiterbar, Adaption, modular, Separierbarkeit der Aspekte, Standardisierung, Domänenspezialisierung

Als erstes Argument für den Einsatz von Komponenten wird immer die *Wiederverwendung* angeführt. Das mag auf den ersten Blick nachdenklich stimmen, war dies doch auch der oberste Anspruch der objektorientierten Softwareentwicklung. Wie oben erwähnt ist die Herangehensweise bei der CBSE jedoch pragmatischer, da Software im besten Fall aus Bausteinen zusammengesetzt werden kann, ohne vorher eine objektorientierte Analyse durchzuführen, wie es das OOD verlangen würde. Ein weiteres Argument, das häufig für den Einsatz der CBSE angeführt wird, ist nämlich die Erstellung von Anwendungen durch Anwender mit *wenig Programmierkenntnissen*, da der Anwender (im Idealfall) die Software aus Komponenten "zusammenstecken" kann, ähnlich wie Objekte beim Spielen mit dem weitverbreiteten LEGO-Spielzeug entstehen. Auch die Einbindung alter Systeme in die moderne Komponententechnologie kann durch *legacy wrapping* erreicht werden. Neben der besseren *Erweiterbarkeit, Adaption* und *Modularität* gelten desweiteren die *Separierbarkeit der Aspekte* und die durchgängigere *Standardisierung* als Hauptvorteile der CBSE. Es wird eine langlebige Komponenteninfrastruktur durch *Domänenspezialisierung* als Ziel angestrebt.

Hohe Anspüche, robust, Qualität, Dokumentation

Wegen der hohen Ansprüche bezüglich *Qualität* und *Robustheit*, die die CBSE an Komponenten stellt, wird im Gegenzug die Entwicklung von Komponenten sehr viel anspruchsvoller, da sie auch in nicht vorgesehenen Kontexten eingesetzt werden können, was schon bei der Entwicklung berücksichtigt werden muss. Auch die *Dokumentation* bekommt einen ganz neuen Stellenwert, da die Komponente von Anwendern eingesetzt werden können muss, die im schlechtesten Fall über nur sehr wenige Programmierkenntnisse verfügen.

Offen: Lizensierung, Interoperabilität

Offene Fragen sind desweiteren die *Lizensierung* und die *Interoperabilität* zwischen den verschiedenen Frameworks, da es bis heute kein "Standard-Framework" gibt und wahrscheinlich auch niemals geben wird.

Voraussetzungen für den Einsatz von Komponenten.

Komponentenmodell (grundlegende Systemstrukturen, Organisationsprinzipien, Qulitätsattribute) Komponenten-Framework (konkrete Realisierung, Unterstützungsdienste, sichert Gültigkeit des Komp-Modells)

Bei der Verwendung des Begriffs "Komponente" sollte klar sein, dass eine Komponente nie für sich selbst steht, sondern stets in einem Kontext (Framework*)* existiert. Dabei muss man zwischen dem *Komponentenmodell* und dem der Implementierung dieses Modells, dem *Komponenten-Framework* unterscheiden. Ersteres stellt ein theoretisches Konzept dar, das die *grundlegenden Systemstrukturen* und *Organisationsprinzipien* und damit die *Qualitätsattribute* des Modells definiert, während letzteres eine konkrete Realisierung des Systems darstellt, das *Unterstützungsdienste* für Konstruktions- und Laufzeit zur Verfügung stellt und die Gültigkeit des verwendeten Komponentenmodells sichert. Die Trennung dieser beiden Ebenen reicht bei den momentan existierenden Komponentensystemen von "nicht existent" (Microsoft *COM*) bis "sehr ausgeprägt" (*CORBA*). Im wesentlichen stellt sich die CBSE als *architekturgetrieben* dar, mit *Frameworks für spezielle Domänen.*

Während sich die erste Generation von Komponentenmodellen noch in erster Linie auf GUI-Modelle bezog (COM, JavaBeans), bezieht sich die zweite Generation bereits auf verteilte Systeme (COM+, EJB). Leider sind diese Systeme noch sehr jung, und so existieren nur wenig Erfahrungen mit ihnen.

Voraussetzung: Schnittstellenbeschreibung, Verhaltensdefinition, vertragliche Absicherung, Def. expliziter Abhängigkeiten, graphische Repräsentation

Auf jeden Fall sollte klar sein, dass als Voraussetzung für den erfolgreichen Einsatz von Komponenten eine *ausreichende Schnittstellenbeschreibung*, eine genaue *Verhaltensdefinition* (mit *Vor-* und *Nachbedingung*), eine *vertragliche Absicherung*,

eine *Definition expliziter Abhängigkeiten* und am besten noch die *Unterstützung für eine grafische Repräsentation* von Nöten sind.

Einfluss auf die Softwareentwicklung.

Neues Überdenken des SE-Prozesses, visuelle Programmierung, programming-in-the-small/large, Entwurfsmuster, Dokumentation Komponentenmarkt, Verträge

Bei der Entwicklung mit Komponenten greifen die in der klassischen Software-Entwicklung erprobten Modelle nicht mehr. Alle Entwicklungsprozesse müssen neu überdacht werden, was auch eine grosse Veränderung bei der Funktionalität integrierter Programmentwicklungswerkzeuge mit sich führt (bis hin zur *visuellen Programmierung*). Bei der Programmierung muss nun unterschieden werden zwischen *programming-in-the-large* und dem *programming-in-the-small*. Auch der Einsatz von *Entwurfsmustern* ist zwingend erforderlich. Vor allem die *Dokumentation* spielt bei der CBSE eine herausragende Rolle.

Ziel ist eine Reduktion des Eigenaufwandes und die Entstehung eines *Komponentenmarktes*. Dies muss organisatorisch durch eine Belohnung für die Verwendung von Komponenten unterstützt werden, wobei *Verträge* die Verantwortlichkeiten im Fehlerfall klar regeln sollten.

Die Komponentenbasierte Softwareentwicklung (CBSE).

Semantikbasierter Ansatz, Domänenspezifikation→R
Domänenanalyse→Kompo

Im Allgemeinen ist der Ansatz der CBSE *semantikbasiert*: unabhängig entwickelte Komponenten werden zur Automatisierung "zusamengeschraubt". Dabei liefern *Domänenspezifikationen* die *Referenzarchitekturen/Framworks*, die der Entwickler dann mit "Fleisch" füllt. Voraussetzung dafür ist die *Domänenanalyse*, deren Ergebnis eine *Komponenteninfrastruktur* ist. Die Phasen der CBSE gliedern sich somit auf in:

- ▷ Anforderungsbeschreibung.
- ▷ Modellbildung.
- ▷ Architektur.
- ▷ Komponentensuche, Programmierung.
- ▷ Integration / Test.
- ▷ Installation / Einsatz.

evolutionäre Vorgehensweise, Spiralmodell, bottom-up

Das herkömmliche *Wasserfallmodell* erweist sich für eine derartige Vorgehensweise allerdings als zu starr, da es die *evolutionäre Vorgehensweise* beim Entwickeln mit Komponenten nicht unterstützt. Als neuer vielversprechender Ansatz gilt das *Spiralmodell*, das das *Bottom-up-Prinzip* wesentlich besser fördert.

Rollen: bean provider, assembler, deployer, container provider, server provider

Auch die *Rollen* der am Entwicklungsprozess beteiligten Personen werden in der CBSE neu festgelegt. Bei den Enterprise JavaBeans wird zwischen dem Bean *Provider*, dem *application assembler*, dem *deployer*, dem *container provider* und dem *server provider* unterschieden.

Im speziellen gibt es folgende Methodiken für das Entwickeln mit Komponenten.

Der Rational Unified Process (RUP).

Reduktion des Risikos, Prototypen, use-cases, iterativ, Rollen: Architekt, use-case Designer, Designer, Design-reviewer

Ziel dieses Ansatzes ist die *Reduktion des Risikos* durch Aufteilung des Software-Projektes in eine Reihe sich entwickelnder *Prototypen*, die anhand sog. *use-cases* definiert werden, die wiederum die zu realisierenden Prozesse beschreiben. Die Entwicklung erfolgt dabei *iterativ*: in jeder Phase gibt es viele mini Wasserfallmodell-Iterationen, wobei jede Phase mit einem definiertem Prototypen endet. Die Rollenaufteilung sieht Architekten, Use-case-Designer, Designer und Design-reviewer vor. Speziell gibt es die Phasen:

- *Inception*: Identifizierung der use-cases, Meilensteine, Visionen, des Domänenmodells, des Wörterbuchs und einer Risikoanalyse mit dem Ziel der Definition der Projektziele und -kosten.
- *Elaboration*: Nach der Beschreibung der use-cases und der nicht-funktionellen Anforderungen nebst Prototyp, Plattform/Technologie und Überprüfung von allem ist das Ergebnis eine Architekturdefinition.
- *Construction:* In dieser Phase beginnt die eigentliche Produktentwicklung in Form von Iterationen durch Definition der use-cases, Systementwurf-Aktualisierung, Implementation des Entwurfs und Test.
- *Transition*: Am Ende der Fehlerbehebung, der Integration von Benutzerwünschen, der Beobachtung der Benutzerzufriedenheit und der Prozessverbesserung steht als Ergebnis eine Beta-Version, die als Basis für einen breiter angelegten Tests verwendet werden kann.

Meilensteine

Jede Iteration bekommt als Eingabe die bisherigen Ergebnisse oder neue use-cases und endet mit erfüllten Aufgaben (*Meilensteinen*).

abgegrenzte Entwicklungseinheiten, Priorisierung

Der Vorteil dieses Ansatzes liegt in den *abgegrenzten Entwicklungseinheiten* und der *Priorisierung* von Projektteilen dem Risiko entsprechend in frühen Phasen. Arbeitsvorgänge werden in Form von use-cases beschrieben und geplant.

aufwendige Projektverwaltung

Die aufwendige Projektverwaltung durch viele (parallele) Aktivitäten benötigt eine ausgeklügelte Änderungsverwaltung und Konfigurationsverwaltung, was als Hauptkritikpunkt dieses Ansatzes gesehen wird.

Business Objects.

Komponentenfabrik, logische Schichten: Benutzer, Arbeits, Unternehmen, Datenquelle Business Objekt

Bei diesem Ansatz wird eine klare Trennung zwischen der Anwendung und der Infrastruktur gefordert und man spricht von einer *Komponentenfabrik*. Die Infrastruktur wird dabei im Laufe der Zeit mit Komponenten "gefüllt". Eine Trennung der Infrastruktur in *logische Schichten* erweist sich als gute Grundlage für die Wiederverwendung. Man spricht in diesem Zusammenhang von der *Benutzer-*, *Arbeitsraum-*, *Unternehmens-* und *Datenquellenschicht*. Ein sog. *Business Object* addressiert spezielle unternehmensweite Belange, ist über das Netzwerk erreichbar, besitzt eine große Granularität und kapselt Geschäftskonzepte.

Architektur- und Komponentenbasiert, Komponentenunabhängig, Kooperationsmodell, evolutionäre Integration neuer Komponenten

Dieser Ansatz ist sowohl *Architektur-* als auch *Komponentenbasiert* und unterstützt die *Komponentenunabhängigkeit*. Er bietet *Koorperationsmodelle* an und die iterative use-case-basierte Vorgehensweise unterstützt eine *kontinuierliche und evolutionäre Integration neuer Komponenten*. Eine gute Grundlage für die Wiederverwendung ist also gegeben.

Catalysis.

Spezifikation, Prozeßmuster, use-cases

Schichten: Domainmodel, Systementwurf, Systemanforderungen

component kit

Diese Methodik baut sehr stark auf *UML* auf und wird deshalb auch mit "UML + eindeutige Vorschrift + flexibler Prozess" umschrieben. Die Wiederverwendung von Modellen und (UML-)Entwürfen spielt hier eine große Rolle. Elemente von Catalysis sind *Spezifikationen*, *Prozeßmuster* und use-cases als Zusammenarbeit kleinerer *use-cases* (*Komponenten*), wobei *Konnektoren* generische Vorgänge (*Schnittstellen*) definieren. Auch hier redet man von Schichten: dem *Geschäfts ("Domain")Modell*, dem *Systementwurf* und den *Systemanforderungen*.

Das Hauptziel von Catalysis ist die Generierung von Produktfamilien aus Komponenten (*component kit*), sowohl durch Neuerzeugung als auch durch Einbindung existierender Teile als Komponenten.

Probleme dieser Methodiken.

zu feingranular, OOA/OOD-fixiert, Haftung, Versionierung/Konfiguration, Overhead, Schnittstellenbeschr., Unterstützung/Kopplung von Frameworks

Designpatterns, Hot-Spots, Active Recipes, semantische Konstrukte

Beim heutigen Entwicklungsprozeß wird der Komponentenbegriff oft zu *feingranular* verstanden und das Software-Engineering ist historisch bedingt zu sehr *OOA/OOD-fixiert*. Damit die Componentware ihre Stärken ausspielen kann, müssen noch *offene Probleme* betreffend Haftung, Versionierung/Konfiguration, Overhead und Schnittstellenbeschreibung gelöst werden. Auch unterstützen Frameworks den Software-Entwicklungs-Prozess noch nicht methodisch und die Kopplung verschiedener Frameworks ist auch noch nicht gelöst.

Schlagworte wie *Designpatterns, Hot-Spots* und *Active Recipes* versprechen Besserungen in Bereichen der CBSE. Auf jeden Fall muss aber die IDL um *semantische Konstrukte* erweitert werden, die für eine automatische Konstruktion genutzt werden können. Im Entwicklungsprozess sollten also zu Anfang Komponenten ausgewählt und vor allem versucht werden, Anforderungen auf Komponenten abzubilden. Dabei sollten grobgranulare Komponenten verwendet werden.

Komponentenmodelle.

Im folgenden werden einige praxisrelevanten Komponentenmodelle vorgestellt, die den momentanen Stand der Technik repräsentieren.

CORBA.

Integrationsplattform für verteilte Objekte, factory-Ansatz, factory-finder

Ziel der Entwicklung von CORBA war die Schaffung einer *Integrationsplattform für verteilte Objekte*, unabhängig von Rechnerarchitektur, Betriebssystem und Programmiersprache. Es ging in erster Linie um die Integration von Systemen, ohne Vorgabe der Programmiersprache und des Zielsystems. Dabei wurde der *Factory-Ansatz* gewählt, bei dem ein Client bei einem *Factory Finder* eine Suchanfrage stellt, die dann mit einer Referenz auf das gesuchte Objekt beantwortet wird.

OMA, object-, application-/domain-Services, common facilities

remote objects, proxy-Objekte, stub, IDL-Compiler

Die *Object Management Architecture* (*OMA*) besteht aus dem *ORB*, der die Logik repräsentiert, um alle Objekte einer ORB-Domäne zu verknüpfen, den *Object Services* auf der einen und den *Application-/Domain-Interfaces* und den *Common Facilities* auf der anderen Seite.

Der Zugriff auf entfernte Objekte (*Remote Objects*) geschieht über Kommunikationsprotokolle transparent durch *Proxy Objekte*, die zu jeder Implementierung eines entfernten Objektes eine korrespondierende Implementierung eines (nahen)

Stellvertreter-Objekts geben, das als Stub automatisch durch einen *IDL-Compiler* generiert wird.

SII, DII, IR

Die *Interface Definition Language (IDL)* definiert Attribute und Operationen, die entfernte Objekte für Clients zur Verfügung stellen. Das *Static Invocation Interface (SII)* generiert die Schnittstellen des Proxy-Stubs, während das *Dynamic Invocation Interface (DII)* die Pseudo-Schnittstellen für die Methode `CORBA::Request`. In dem sog. *Interface Repository (IR)* werden Schnittstelleninformationen persistent erfasst.

Rollen, Server, Servant, POA

Die Rollenverteilung in CORBA ist aufgeteilt in *Servant* (konkrete Realisierung eines CORBA-Objekts) und *Server* (Rechenkontext, in dem sich ein Servant befindet). Die Assoziation zwischen Objektreferenz und Servant wird über sog. *Portable Object Adapter (POA)* realisiert.

Nebenläufigkeit? Concurrency.

Leider gibt es in CORBA keine Spezifiktaion zu dem Thema *Nebenläufigkeit*. Lediglich der Dienst *Concurrency* bietet Sperren, Transaktionen und Koordination. Die Implementierungen von Objekten können über *persistent, object* und *method server* bereitgestellt werden.

nur Schnittstellenvererbung, callbacks, Speicherverwaltung, proxy-Objekte, Sprachabbildungen, dynamische Aufrufe

Da nur *Schnittstellenvererbung* erlaubt ist, ist die Verwendung von Quelltexten verschiedener ORBs fast unmöglich. Dafür können aber Ausnahmen in der IDL beschrieben werden und mittels *callbacks* eine Umkehrung der client-server-Beziehung erreicht werden. *Speicherverwaltung* wird durch Referenzzählung realisiert (abhängig von der Programmiersprache) und *Proxy-Objekte* sorgen für die transparente Nutzung entfernter wie lokaler Objekte. Die Möglichkeit zur Nutzung heterogener Objekte wird durch *Sprachabbildungen* und *dynamische Aufrufe* realisiert.

Standard, sprachneutral, heterogen, Basisdienste

Die grossen Vorteile von CORBA sind die konsequente *Standardisierung*, die *Sprachneutralität*, die *Heterogenität* und die Bereitstellung von *Basisdiensten* (*CORBA-Services*)

CORBA Component Model.

Serverkomponenten, Containerfunktionalität auf Basis von POA's

Das CCM stellt einer *Architektur für Serverkomponenten und ihre Interaktion* zur Verfügung. Dank ihr lassen sich Komponenten *sprachunabhängig paketieren.* Die *Containerfunktionalität* wird auf Basis spezieller POA zur Verfügung gestellt (z.B. Sicherheit, Transaktion, Notifikation und Persistenz).

Mehrfachschnittstellen, CCM Container Modell, CIDL, Home-Schnittstelle

Das CCM erweitert das CORBA Objekt Modell um Mehrfachschnittstellen, Komponentenkonstruktion und dem CCM Container Modell. Die IDL wurde zur *CIDL* erweitert. Die Navigation erfolgt über Schnittstellen, wo insbesondere die *Home-Schnittstelle* hervorzuheben wäre, die für Finder-Operationen von grosser Bedeutung ist.

Life-Cycle-Operationen, Anfragefunktionen, XML-basierte Konfig.-Dateien

Der Containter bietet sog. *Life-Cycle-Operationen* und *Anfragefunktionen* an, die für die Verwaltung der Komponenten und die Nutzung deren Dienste nötig sind. Es stellt ein Framework für externe Schnittstellen und lokal Systemdienste dar. Die Komposition von Komponenten wird also über Verknüpfungen von Verbindungen der äußersten Schicht zu einer Applikation realisiert, wobei die Konfigurationsdateien auf XML basieren und eine dynamische Komposition erlauben. Aus der CIDL wird *Implementierungs-Code* erzeugt, der dann zur Startzeit interpretiert wird.

Server Containter: service, session, process, entity

Die Server Container lassen sich verschiedenen Kategorien zuordnen:

- ▷ *Service:* stateless, no ID, ausschließlich für Operationen

- *Session:* session-state, no ID, Methoden (Iterationen), Transaktionen, Komponenten, Container
- *Process:* durable, no ID, s.o. (abgeschlossene Prozesse)
- *Entity:* durable, ID, s.o. (Business Objects)

Die Einbindung von EJB wird durch sog. *Bridges* ermöglicht. Außerdem existiert für JavaBeans ein extra CORBA Container.

Microsoft's COM.

OLE

Das *Component Object Model* entstand aus *OLE,* einer Technik zur Erstellung sog. Verbunddokumente. Hauptmotivation war die bessere Wartbarkeit der grossen Office-Suite, woran man schon den pragmatischen Hintergrund erkennen kann.

Interaktion von Komponenten, binärer Standard, sprachunabhängig, aggregation, containment, Multithreading, persistente Referenzen (Monikers), connectable objects

Mittels COM ist die Interaktion von Komponenten im gleichen Prozess, auf dem gleichen Rechner und imt Netz (über DCE-RPC-Mechanismen) möglich, und zwar völlig transparent. Bei COM handelt es sich um einen *binären Standart*, der aber sprachunabhängig ist (jede Komponente muss allerdings `IUnknown`-Schnittstelle unterstützen). Zur Wiederverwendung werden die Modelle *aggregation* und *containment* unterstützt. Eine 128Bit GUID sorgt für eine Eindeutigkeit der Komponenten und Multi-threading ist ebenso vorgesehen, wie persistente Referenzen duch sog. *monikers.* Die Notigikation erfolgt über *connectable objects*, die Typauswahl ist allerdings sehr eingeschränkt.

MIDL basiert auf DCE IDL, Einfachvererbung, Mehrfachschnittstellen, Sutbs von MIDL-Compiler

Die *MIDL* basiert auf der DCE IDL und unterstützt neben Einfachvererbung auch Mehrfachschnittstellen und eine Sprachanbindung an C/C++. Proxy- und Stub-Dateien werden vom MIDL-Compiler automatisch erzeugt.

registry, directory service, garbage collection, structured storage, compund document architecture, callbacks

Die Registrierung von Komponenten erfolgt mittels der Windows Registry oder ab Windows 2000 über einen Directory Service. Die um verteilte Objekte erweiterte Version von COM heißt *DCOM* und unterstützt momentan C/C++ und Java, wobei die Weitergabe von Zeigern über monikers realisiert ist. Wie bei CORBA wird eine *garbage collection* über Referenz-Zählung realisiert und ein *structured storage* wird mittels der *compound docoment architecture* erreicht. Die Ausnahmebehandlung ist proprietär über einen `HResult`-Wert, von dem 16Bit für eigene Codes zur Verfügung stehen und *connectable objects* ermöglichen *callbacks.*

Die gute Unterstützung durch VC++ und Windows, sowie die Möglichkeit der Erstellung von *compound documents*, gepaart mit einem existenten Komponentenlizensierungsmechanismus, machen COM zu einem durchaus attraktiven Komponentenmodell. Die enge Integration in Windows steht der Plattformunabhängigkeit allerdings entgegen

SUN's Enterprise Java Beans.

Java, als von Grund auf neu konzipierte Sprache, ist *streng typisiert*, hat *keine Zeiger*, ermöglicht *dynamisches Binden*, ist *plattformneutral* (da portabler Byte-Code erzeugt wird), hat grosse Ähnlichkeiten mit C++, bietet *Introspektion*, ein *Sicherheitskonzept* und *umfangreiche Bibliotheken.*

JavaBeans: getter-, setter-Methoden, publisher-subscriber, Intra-Prozess

JavaBeans stellen eine client-seitige Software-Komponententechnologie dar, die als Framework für komponentenbasierte Anwendungsentwicklung (vor allem GUI-basierter

Programme) dient (ähnlich ActiveX-controls, die zu Erstellung von compound documents verwendet werden). Attribute werden über *getter-* und *setter-Methoden* manipuliert und das Event-Handling geschieht über den *publisher-subscriber-Ansatz*. Java Beans stellen ein *Intra-Prozess Komponentenmodell* dar.

Java Enterprise Beans: inter-Prozess, serverseitig, nicht-visuell, multi-tier, Transaktionsunterstützung, Ausführumgebung, erzeugen/initialisieren/löschen/auslagern, home-Schnittstelle

Im Gegensatz zu JavaBeans stellen *Enterprise JavaBeans* eine server-seitige *Inter-Prozess* Komponententechnologie dar, die nicht-visuell ist, *multi-tier-Anwendungen* unterstützt, implizite *Transaktionsunterstützung* bietet und einen Container als *Ausführumgebung* benötigt. Ein EJB-container *erzeugt/initialisiert/löscht Beans*, sorgt für *Transaktionsfähigkeit*, exportiert Beans-Referenzen in den *JNDI-Namensraum* und sorgt für die *Auslagerung* von Session-Beans. *Optionale Persistenz* und *Sicherheitsdienste* sind weiter Aspekte eines Containers. Die *Home-Schnittstelle* eine Objektes stellt Methoden zum Erzeugen eines Objektes zu Verfügung (*ejbCreate*) und die *ejbRemote*-Schnittstelle ermöglicht den entfernten Zugriff auf die Bean. Bean-Methoden werden jedoch niemals direkt vom Client benutzt, dafür stehen *ejbObjects* zur Verfügung, die als Proxy fungieren.

Es werden die folgenden Bean-Kategorien unterschieden, die über den sog. *deployment descriptor* identifiziert werden:

Deployment descriptor: session, entity, stateless session, stateful session

- ▷ *Session Beans* bieten einen Dienst für einen Client, sind nicht persistent und können an Transktionen beteiligt sein. Sie sind kurzlebig und bieten keine Zustandssicherung.
- ▷ *Entity Beans* repräsentieren Daten in der Datenbank, können transaktional und von mehreren Benutzern (gleichzeitig) genutzt werden. Die Zustandssicherung muss durch den EJB-Server gewährleistet werden.
- ▷ *Stateless Session Beans* besitzen keinen internen Zustand, ermöglich dafür aber ein pooling und können durch beliebig viele Clients benutzt werden.
- ▷ *Stateful Session Beans* sind zustandsbehaftet und stellen eine 1:1-Relation innerhalb einer Session zwischen Client und Bean dar.

bean managed persistence, container managed persistence

In Sachen Persistenz wird unterschieden zwischen der *bean-managed persistence* (der Beans-Anbieter implementiert den DB-Zugriff selber) und der *container-managed persistence*, bei der persistente Variablen im deployment descriptor beschrieben werden müssen. Als APIs stellen die EJB JNDI, JIDL, JDBC, JTA und JTS zur Verfügung.

+: dynamisches Nachladen, objektorientiertes sprachkonzept, Introspektion

Vorteilhaft an diesem Ansatz ist die Möglichkeit der Erstellung und Nutzung von Komponenten *sowohl für den Client, als auch für den Server*, mit *dynamischem Nachladen*, einem klar *objekt-orientierten Sprachkonzept*, *Introspektion* durch Reflection und nicht zuletzt dem umfangreichen JDK.

-: Interoperabilität von Containers, RMI, Weiterreichen des Transaktionskontextes, Sicherheitsdomänen, Geschwindigkeit, Templates, schlechte Unterstützung

Dem gegenüber steht die nicht festgelegte *Interoperabilität zwischen Containern* (aufgrund einer fehlenden Standard Container API), die Frage nach der *RMI-Implementierung*, der *Weiterreichung des Transaktionskontextes* und der *Unterstützung unterschiedlicher Sicherheitsdomänen*. Auch die *Geschwindigkeit* läßt zu wünschen übrig, eignet sich die Sprache doch nicht für echtzeitfähige Anwendungen. Das *Fehlen parametrisierter Typen* (Templates) und die *nicht hinreichende Unterstützung beim Erzeugen von Komponenten* wird ebenfalls bemängelt.

Probleme des CBSE heute.

Auch beim heutigen Einsatz von Komponenten bestehen immer noch offene Fragen bezüglich

Notifikation

- *Notifikation*: Alle aktuellen Komponentenmodelle arbeiten mit Nachrichten, um Komponenten über aufgetretene Ereignisse zu informieren. Was passiert jedoch bei *Mulitcasts*? Was machen *Änderungen in der Sendezeit* aus? Was ist mit *Ausnahmebehandlungen*?

Nebenläufigkeit

- *Nebenläufigkeit*: Laufen Komponenten parallel läßt sich ihr Verhalten nur sehr schwer abschätzen oder berechnen. Auch eine *Synchronisierung* wird schwer. Mittels *Reentranz* wird versucht *Transaktionalität* und *Message Queueing* zu verwirklichen.

Granulatrität

- Bei der *Granularität* unterscheidet man zwischen
  - *feinkörnig*: passive, genutzte Komponenten (z.B. Combo-Box)
  - *mittelkörnig*: (inter-)aktive Komponenten (z.B. zur Verschlüsselung, Abrechnung)
  - *grobkörnig*: aktive, kontrollierende, architekturelle Komponenten (z.B. Lagerverwaltung, Reisekostenabrechnung)
- Design- vs. Runtime?
- Persistenz vs. flüchtigen Zustand?
- Implementierung (Kontrakte, Vererbung, heterogene Komponenten)?
- Offenheit der Komponenten (black/white/grey-box)

Erstellungsaufwand

Den 3-4-fachen Entwicklungskosten einer Komponente steht eine Kostenreduktion von etwa nur $\frac{1}{4}$ gegenüber, so dass eine Komponente 3- bis 5-mal wiederverwendet werden muss, bis der *Erstellungsaufwand* abgedeckt ist. Dieser ist vor allem wegen der gründlicheren Konzeption und der umfrangreichen Dokumentation um ein vielfaches umfangreicher, als bei bisherigen Software-Entwicklungs-Konzepten.

Auch die organsatorischen Aspekte sind bei der Entwicklung mit Komponenten nicht zu vernachlässigen. Im besonderen sind dies:

Versionsverwaltung

- *Versionsverwaltung:* Spezifikation, Implementierung, UML-Modelle, Abhängigkeiten, Versionsverträglichkeiten, Ressourcen (zur Laufzeit).

Abhängigkeiten

- *Abhängigkeitsverwaltung:* Verträglichkeits-/Ausschlußlisten, Toolunterstützung bei Schnittstellen, Diensten und Kontrakten, Abhängigkeiten überprüfen und minimieren.

Beschreibung

- *Komponentenbeschreibung:* Extra Dokumente, im Code selbst, Administrationsbeschreibung.

Test

- *Test:* Wahl einer "sicheres" Implementationssprache, Kontrakt, aufwendig.

Geschäftsmodell

- *Geschäftsmodell:* Inner- und Aussergeschäftlich.

Suchen

- *Suchen:* Klassifikation von Komponenten.

Archivieren

- *Archivierung:* Konfigurationsverwaltungstools, DB, Repositories, fehlende Klassifizierung, Strukturierung der Archive.

Rollen

- *Rollen:* Architekt, Entwickler (Komponentensystem / Komponenten Framework / Komponente), Verwalter, Administrator

Paketierung

Dokumentation

- ▷ *Paketierung:* Verkaufseinheit, Auslieferungseinheit.
- ▷ *Dokumentation:* Name, administrative Aufgaben, Benutzerdoku, Entwicklerdoku, Wörterbuch, Diagramme, Anmerkungen.

Zusammenfassend stehen den beantworteten noch mehr offene Fragen gegenüber, was durch den sehr unpräzisen Komponentenbegriff nicht gerade gefördert wird.

Fragen

Was ist eine middleware?
Dies ist eine Sofwareinfrastruktur, die die Interaktion zwischen den auf heterogenen Systemen ablaufenden Anwendungskomponenten unterstützt. Sie wird dem Betriebssystem (Netzbetriebssystem) hinzugefügt bzw. übernimmt selbst die Aufgaben eines Betriebssystems (verteiltes Betriebssystem).

Wozu braucht man das?
Um die Verteilung transparent zu machen und um eine Infrastruktur für die Anwendungskomponenten zu bieten und die Interaktionen der beteiligten Komponenten zu unterstützen.
Die Verteilungsplattform isoliert die Anwendungsprogramme vom BS und Kommunikationsprotokollen. Die *middleware* benutzt die Funktionen (Dienste) des BS über Systemaufrufe. Anwendungsprogramme benutzen die *middleware* über eine API.
Verteilungsplattformen sind selbst verteilte Client/Server-Systeme. Man unterscheidet zwischen *Anwendungs-Server* und *System-Server*. Anwendungs-Server sind Teil der verteilten Anwendungen. Ihr Dienst ist anwendungsspezifisch. System-Server gehören zur Verteilungsplattform und erbringen allgemeine, systemnahe Dienste (z.B. Verzeichnisdienst und Sicherheitsdienst).