

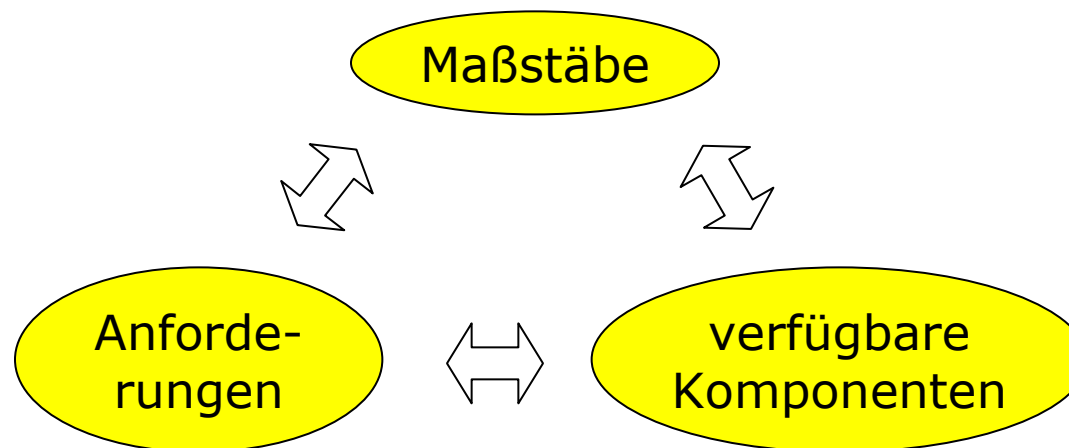
Vorlesung Software aus Komponenten

2. Grundlagen

Prof. Dr. Hans-Gert Gräbe
Wintersemester 2005/06

Komponenten abgrenzen

- Problem: Wie ist ein kompletter Entwurf in Komponenten zu partitionieren?



Prinzipien:

- Modularität und Kapselung
- Abhängigkeiten zwischen K. sind expliziert
- Mehrere Abstraktionsebenen, hierarchische Strukturierung
- natürliche Zuordnung von Verantwortlichkeiten
- Migrations-Erfordernisse vorab berücksichtigen

typische Betrachtungsansätze (führen zu unterschiedlicher Granularität)

- K. als Einheit der Abstraktion
 - Ansatz: white box → black box
 - Entwurfsexpertise kapseln (design expertise ready for use)
 - Theoretische Einschränkung der Vielfalt in der aktuellen Ebene ist Basis für größere Vielfalt in der nächsthöheren Ebene

2.4. Komponenten abgrenzen

typische Betrachtungsansätze

- K. als Einheit der Kostenrechnung
 - wichtig in größeren industriellen Kontexten, um Projektentwicklungskosten verfolgen zu können
- K. als Einheit des Managements
 - oft zu klein, Management auf der Ebene von Subsystemen, die mehrere Komponenten zusammenfassen
 - etwa auf der Ebene der Server
- K. als Einheit der Analyse
 - Analyse an vielen Stellen im Komponentenlebenszyklus erforderlich (Spezifikationsprüfung, Tests, Re-Engineering)
 - Kopplung zwischen Einheiten bestimmt, wie weit eine individuelle Analyse zweckmäßig bzw. überhaupt möglich ist
 - Regel: Einheiten für Analyse so klein wie möglich
 - Regel: streng statische hierarchische Grenzen (Moduln, Subsysteme) erleichtern die Analyse

2.4. Komponenten abgrenzen

typische Betrachtungsansätze

- K. als unabhängig compilierbare Einheit
 - Compilierbarkeit und Analyse sind eng verbunden
 - white box: Analyse
 - black box: Compilierbarkeit
 - sinnvolle Einheiten: Moduln oder Klassen
 - Globale Optimierung?
 - Antwort 1: Einheiten möglichst groß wählen
 - Antwort 2: Optimierung zwischen Komponenten, vielleicht sogar erst nach der Lokalisierung
 - muss im Komponentenkonzept und der Komponentenbeschreibung verankert sein
- K. als Auslieferungseinheit
 - Bündel der technischen und wirtschaftlichen Aspekte
 - Management (Service, Wartung, Schulung, Updates, ...) treibt den Preis in die Höhe
 - betriebswirtschaftliche Bedeutung jenseits der (geringen) Replikationskosten

- K. als Packungs-Einheit
 - Entpackung (deployment) = Prozess der Vorbereitung der Komponente auf den Einsatz in einer speziellen Umgebung (Lokalisierung)
 - wurde lange nicht als separater Schritt betrachtet
 - Prozess der Anbindung an eine spezielle Komponentenplattform
 - Konfiguration = Einstellung spezieller Eigenschaften der Komponente für den konkreten Einsatz
 - Installation = plattformspezifische Aktivität, mit der eine entpackte Komponente für die Nutzung in einer speziellen Hardware-Konfiguration verfügbar gemacht wird, die von der Plattform unterstützt wird.
 - Zeit, in der auch kritische Tests ausgeführt werden, die vor dem eigentlichen Betrieb erfolgen müssen (etwa Integritätstests)
 - für alle drei Aktivitäten müssen entsprechende Beschreibungen erstellt werden

- K. als Einheit der (Auseinandersetzung um) Fehlersuche
 - Problem: Was ist (u.a. wer haftet?), wenn ein aus Komponenten zusammengebautes Produktivsystem fehlerhaft arbeitet?
 - Problem der Lokalisierung von Fehlern (und damit Verantwortlichkeiten)
 - besonders schwierig wenn Objektreferenzen die Komponentengrenzen verlassen
 - vitale Regel: Fehler müssen in den verursachenden Komponenten bleiben (bug containment)
 - typische nicht-lokalisierbare Fehler: Speicherzugriffsfehler
 - Folge: Ausnahmebehandlungen müssen in der Regel innerhalb einer Komponente bleiben
 - Ausnahmen davon sind im Komponentenkontrakt zu fixieren
 - ➔ Komponente als Einheit der Fehlerbehandlung

2.4. Komponenten abgrenzen

typische Betrachtungsansätze

- K. als Einheit der Erweiterung
 - Beispiel: K. implementiert eine konkrete (standardisierte) Schnittstelle
 - Objekte in einer Erweiterungshierarchie sind gewöhnlich enger gekoppelt als andere
 - „friends“-Mechanismus, „protected“-Schnittstelle
 - sind Ersatz für Mechanismen der Kapselung mehrerer Objekte
 - unabhängige Erweiterungen und deren Koexistenz
 - eine Einheit der Analyse darf nicht in mehrere Erweiterungseinheiten zerlegt werden
 - gemeinsame Analyse hat oft kontextuelle Abhängigkeiten zwischen den Teilen zur Folge
- K. als Einheiten der Instanziierung
 - Einheit der Instanziierung sind Objekte
 - also: kein sinnvolles Kriterium für Komponentenabgrenzung

2.4. Komponenten abgrenzen

typische Betrachtungsansätze

- K. als Einheit des Ladens
 - oft wird beim ersten Einsatz einer Funktionen erst entsprechender Code geladen.
 - benötigt Mechanismen des dynamischen Ladens (etwa DLL)
 - typisch werden dabei gleich mehrere zusammenhängende Klassen geladen
 - → Komponente als Einheit des Ladens sinnvoll
 - Kontrolle der Version
 - Problem der fragilen Basisklasse
 - Maximale Flexibilität, wenn Versionskontrolle auf der Ebene von Schnittstellen oder sogar Methoden
 - Java: Versions-Check erst zur Laufzeit
 - ist problematisch, da Inkompatibilitäten erst mitten in der Programmausführung entdeckt werden
 - Kontrolle der Erfüllbarkeit der Importrelationen
 - Kontrolle von Namenskollisionen
 - Lösung: Hierarchisches Namensraum-Konzept
 - Problem der Namenskollision: Versionen derselben Komponente

- K. als Einheit des Ladens (Fortsetzung)
 - Konsistenz bereits geladener Komponenten muss erhalten bleiben
 - Konsistenz muss dazu lokale Eigenschaft sein
 - schließt z.B. globale Typprüfungsansätze aus
- K. als Einheit der Lokalität
 - Problem im Kontext verteilter Systeme: was befindet sich (lokal) auf welchem Rechner
 - typisch sind hierarchische Konzepte des Clusters von Rechnern
 - SAN, LAN, WAN, Internet (und weitere Zwischenstufen)
 - unterschiedliche Kommunikationszeiten und -kosten
 - Tradeoff: minimale Kommunikationskosten vs. maximale Ressourcennutzung
 - hohe Kopplung zwischen Objekten aus derselben Komponente
 - ➔ Komponente sollte nicht auf verschiedene Prozesse oder Maschinen verteilt sein
 - Bündelung von Zugriffen auf Objekte über Prozessgrenzen hinweg
 - ein komplexer statt mehrerer einfacher Zugriffe

Komponenten-Modelle

1. Grundlagen: Kommunikationskonzepte
2. OMG und CORBA – der geschäftsprozesszentrierte Ansatz
3. Sun und Java – der webzentrierte Ansatz
4. Microsofts und .NET – der dokumentenzentrierte Ansatz

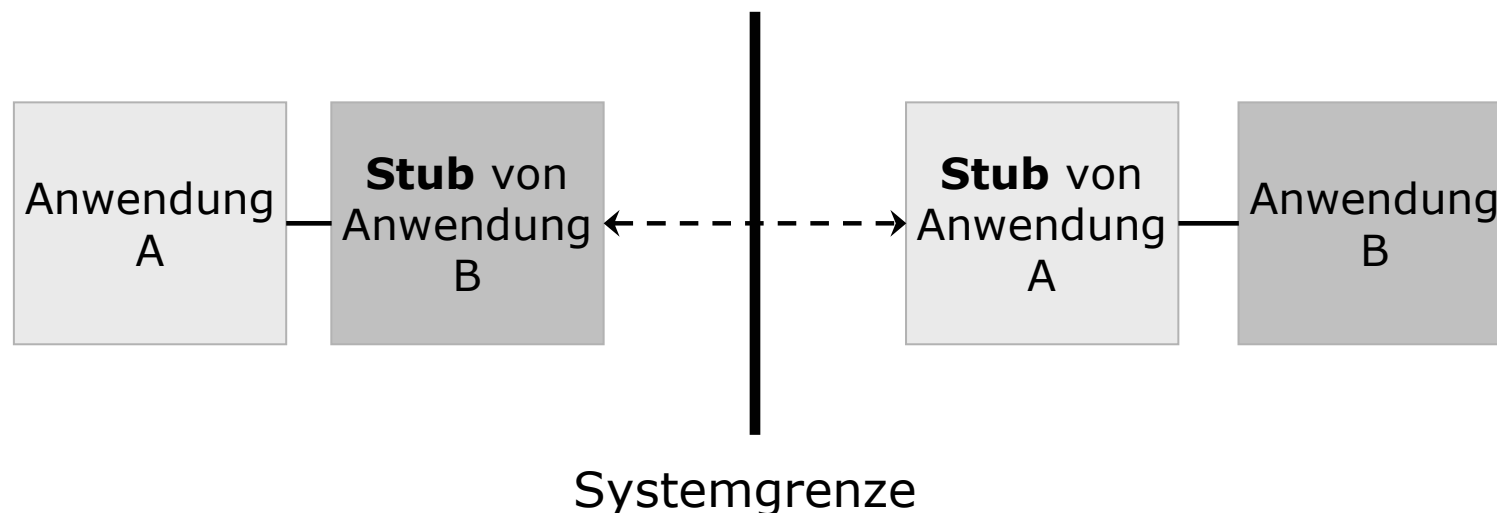
Interprozess-Kommunikation (IPC) auf OS-Ebene

- Charakteristika
 - kaum plattformübergreifend standardisiert
 - nicht Teil des von-Neumann-Modells
 - Prozess = virtueller Rechner auf physischem Host
- IPC-Modelle: Dateien, Pipes, Sockets, Semaphore, shared memory
 - außer Sockets keins so weit standardisiert, dass es plattformübergreifend eingesetzt werden könnte.
 - außer shared memory skalierbar und internetfähig
 - operiert auf Bitebene -> zu kompliziert für komplexe Anwendungen

IPC operiert auf Bitebene und ist deutlich zu kompliziert für komplexe Anwendungen.

Remote Procedure Calls (RPC, 1984)

- Ansatz: Stubs, die auch entfernte Prozeduraufrufe lokal aussehen lassen
 - Aufgabe des Stub: Serialisierung bzw. Deserialisierung des Prozeduraufrufs und der Aufrufparameter unter Beachtung von plattformabhängiger Byte-Kodierung, Zahldarstellung, ...



- Nutzung leichtgewichtiger RPC zur IPC auf derselben Maschine (Windows NT)
- Vorteil: einheitliches Abstraktionsniveau für alle Kommunikationserfordernisse (innerhalb eines Prozesses, zwischen Prozessen, zwischen Computern)
- Nachteile:
 - Versteckte Kommunikationskosten (Unterschied um Faktor $10...10^4$), Client kann nicht unterscheiden, ob lokaler oder entfernter Aufruf
 - blockierendes Konzept
 - Umgang mit Versionierung und Evolution von Komponenten vollkommen unklar

Das RPC-Konzept bildet zusammen mit dynamisch linkbaren Bibliotheken (DLL) die Basis für das einfachste Komponenten-Framework (und ist das heute am weitesten verbreitete).

Distributed Computing Environment (DCE)

- Standard der Open Software Foundation für RPC auf heterogenen Plattformen (<http://www.opengroup.org/dce>, aktuelle Version 1.2.2)
- **Interface Definition Language (IDL)**
 - zur Standardisierung des Absetzens von RPC
 - genaue Angabe von Architekturspezifika erforderlich
 - etwa: int = 32-bit low-endian Zweierkomplementdarstellung
- **Universally Unique IDentifiers (UUID)**
 - Standard zur global eindeutigen Bezeichnung und Identifikation von Computern, Prozessen, Prozeduren und Daten
 - maschinenorientierte Bitdarstellung, deshalb wird zur menschenlesbaren Beschreibung mit Aliasen gearbeitet

DCE ist ein Standard, um (u.a.) RPC zwischen heterogenen Plattformen konsistent einzusetzen.

Objekte und Methoden

- RPC ist ein statisches Aufrufkonzept
- Besonderheit von Methoden gegenüber Prozeduren:
 - werden dynamisch an Hand der Charakteristika des Objekts (=Instanz seiner Klasse) ausgewählt
 - erst nach dieser Auswahl kann der RPC-Mechanismus greifen
 - Klassen müssen dazu genügend (binär kodierte) Information bieten, die durch Introspektion zur Laufzeit abgefragt werden kann
 - Objektreferenzen als Aufrufparameter
 - Keine automatischen Objektkopien

Der RPC-Ansatz ist deutlich aufzubohren, wenn mit Objekten und Methoden mit laufzeitabhängigem Verhalten umgegangen werden soll.

Erweiterung des RPC-Konzepts

- Verwendung von Funktionsvariablen, die zur Laufzeit mit einem Funktionszeiger als Wert belegt werden
- Virtuelle Methodentabellen (VMT)
 - Beispiel: COM's dispatch tables (Microsoft)

oder Spezielle Laufzeitumgebung (virtual machine, VM)

- übernimmt Management von *Methoden*-Aufrufen
 - SOM (IBMs System Object Model)
 - Java (Java virtual machine)
 - .NET common language runtime (CLR)
- braucht spezielle Mechanismen, um **über die Grenzen der VM hinaus** mit Komponenten zu kommunizieren

Über RPC-Mechanismus hinaus gehende Fragen, die ein objektorientiertes Kommunikationskonzept beantworten muss

1. Wie werden Schnittstellen spezifiziert?
2. Wie werden Objektreferenzen behandelt, wenn der lokale Bereich verlassen wird?
3. Wie werden Dienste aufgefunden und bereitgestellt?
4. Wie wird die Evolution von Komponenten gehandhabt? (Versionsmanagement)