

Vorlesung Software aus Komponenten

2. Grundlagen

Prof. Dr. Hans-Gert Gräbe
Wintersemester 2010/11

Direkte und indirekte Schnittstellen

- Direkte Schnittstelle: Schnittstelle der Komponente selbst
 - meist prozeduraler Natur
- Indirekte Schnittstelle: Schnittstelle von Objekten, die in der Komponente erzeugt werden
 - meist objektorientierter Natur
- Vereinheitlichung durch Einführung eines statischen Objekts möglich
 - typischer Ansatz von OO Sprachkonzepten
- Überladung indirekter Schnittstellen und späte Bindung
 - Provider des Dienstes hängt vom Objekt ab
 - Derselbe Dienst kann über dieselbe Schnittstelle innerhalb desselben Komponenten-Kontexts von unterschiedlichen Anbietern kommen

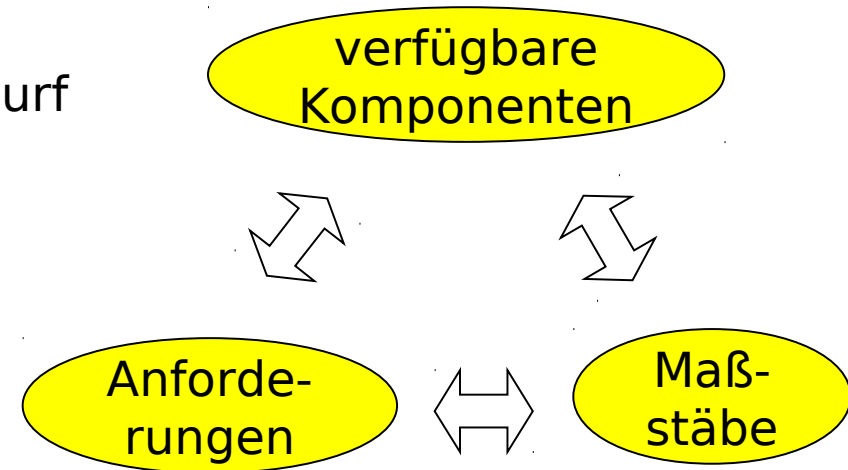
Schnittstellen und Versionen

- Problem: Schnittstellen können ihr Verhalten zwischen Versionen wechseln
- Management traditionell über Versionsnummern
 - Komponente als unteilbare Einheit => Versionsnummern nur für ganze Komponenten
- Versions-Information in Import- und Exportschnittstellen
 - direkte Schnittstellen: Abfrage zur Bindungszeit, also (nur) vor dem ersten Schnittstellenaufruf
 - indirekte Schnittstellen: Abfrage vor jedem Aufruf erforderlich
 - Alternative: Integration ins Management der Objektidentität
- Problem der Versions-Information, wenn eine Objektreferenz Komponentengrenzen überschreitet
 - Objekt bietet eigene Dienste an
 - etwa Rückgabe in einem bestimmten Format
 - Objekt nutzt Dienste anderer => dynamische Versionskontrolle

- Schnittstellenversionen müssen klar als kompatibel oder klar als veraltet (deprecated) deklariert werden können
- Ansatz der unveränderlichen Schnittstellen-Spezifikation (immutable interfaces)
 - Neue Versionen nur als neue Schnittstellen
 - Veraltete Schnittstellen werden einfach nicht mehr unterstützt
 - Beispiel: COM = Component Object Model von MicroSoft
- Veränderbare Schnittstellen-Spezifikationen:
 - klares Kompatibilitäts-Konzept erforderlich
 - Installation verschiedener Komponentenversionen in derselben Umgebung kann erforderlich sein.
 - Unterscheidung zwischen Komponenten, die immer in der aktuellsten Version verwendet werden können und Komponenten, die nur in der ursprünglich installierten Version verwendet werden können
 - wird im Rahmen der CLR = Common Language Runtime verfolgt

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

- Wie „fett“ soll eine Komponente sein?
 - optimal, aber unreal: „richtige Schnittstellenmenge“ und keine Kontext-Abhängigkeit
 - maximal: „fette“ Komponente, die alle benötigten Dienste mitbringt (=Applikation, grobkörnig)
 - minimal: Auslagern aller bis auf die zentrale Funktionalität (=Klasse, feinkörnig)
 - „Maximizing reuse minimizes use.“
 - Grund: Explodierende Kontext-Abhängigkeit
 - würde nur unter statischen Entwicklungsbedingungen funktionieren
 - Beispiel: Linux-Probleme mit Bibliotheksversionen
 - praktisch ist hier ein je ausgewogenes Mittel zu finden
- Je detaillierter Normierung und Standardisierung, desto schlankere Komponenten sind möglich
 - Standardisierung ist in vertikalen Marktsegmenten (funktional) eher möglich als in horizontalen, aber wegen der geringen Marktgröße schwieriger

Komponenten und Entwurf

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

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

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

Komponenten aus technischer Anbietersicht

K. als unabhängig compilierbare Einheit

- Compilierbarkeit und Analyse sind eng verbunden
 - white box: Analyse
 - black box: Compilierbarkeit
- sinnvolle Einheiten: Klassen oder Pakete?
- Globale Optimierung?
 - Antwort 1: Einheiten möglichst groß wählen
 - Antwort 2: Optimierung zwischen Komponenten, vielleicht sogar erst nach der Lokalisierung
- muss im Komponentenkonzept und in der Komponentenbeschreibung verankert sein

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

Komponenten aus technischer Nutzersicht

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
 - Maximale Flexibilität, wenn die Versionskontrolle auf der Ebene von Schnittstellen oder sogar Methoden erfolgt
 - Erfordert eine Zuordnung von Methodenversionen zu Komponentenversionen

K. als Einheit des Ladens (Fortsetzung)

- 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
- 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 und Management

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 wird es, 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 Komponenten-Kontrakt zu fixieren
 - Komponente als Einheit der Fehlerbehandlung

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 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