

Vorlesung Software aus Komponenten

2. Grundlagen

Prof. Dr. Hans-Gert Gräbe
Wintersemester 2006/07

Definition Software-Komponente

Eine Software-Komponente ist eine Einheit der Komposition mit durch Kontrakt spezifizierten Schnittstellen und nur expliziten Kontext-Abhängigkeiten. Eine Software-Komponente kann unabhängig verteilt werden und zur Komposition durch Dritte verwendet werden.

- Definition wurde erstmals so 1996 auf der „European Conf. on OO Programming“ gegeben [Szyperski, Pfister]
- technische Seite: Unabhängigkeit, Schnittstellen-Kontrakt, Zusammenbau
- soziale Seite: Dritte, Verteilung
- Diese Verbindung ist typisch für den Komponentenbegriff nicht nur im Software-Bereich

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

Schnittstellen-Kontrakt

- Schnittstellen-Spezifikation als Kontrakt zwischen
 - Nutzer der Funktionalität einer Schnittstelle und
 - Anbieter der Implementierung dieser Schnittstelle
- verbreiteter Zugang auf technischer Ebene: durch Vor- und Nachbedingungen (Hoare-Kalkül: $\{V\} P \{N\}$)
 - Nutzer sichert die Vorbedingungen V
 - Anbieter sichert dann Nachbedingung N
 - Problem: Sichert funktionale Eigenschaften, aber weder Performanz von P noch Termination überhaupt
- heute üblich: auch nicht-funktionale Aspekte im Kontrakt erfassen
 - Beispiel: Service Level Agreement
 - enthält Qualitätsaussagen für den Betrieb wie Verfügbarkeit, Fehlerrate, Datensicherheit etc.
 - Konsequenzen im Einsatz sind ähnlich gravierend wie funktionale Fehler

- „undokumentierte Features“
 - Auf Komponenten-Verhalten kann jenseits der Spezifikation auch aus Beobachtung des laufenden Betriebs geschlossen werden
 - typisches Ergebnis eines „Debugging“-Prozesses bei Fehlersuche
 - kleine Fehler sind ökonomischer auf der Nutzerseite als auf der Anbieterseite zu beheben
 - Open-Source-Ansatz

Vererbung als Prinzip

- Drei wesentliche Facetten
 - Subklassen: Vererbung von Implementations-Fragmenten
 - Implementations-Vererbung
 - Java: *extends*
 - Subtypen: Vererbung von Kontrakt-Fragmenten
 - Schnittstellen-Vererbung
 - Java: *implements*
 - Substituierbarkeit: Vererbung funktionaler Eigenschaften
- Mehrfach-Vererbung
 - Kein Problem auf Schnittstellenebene
 - Diamanten-Problem auf Implementationsebene
 - auf der Ebene der Zustände (Attribute – Referenz oder Kopie?)
 - Referenz bricht Kapselung
 - auf der Ebene der Methoden

Das Problem der fragilen Basisklasse

- Fragestellung: Was passiert in einer Vererbungsrelation, wenn die Basisklasse durch eine neue Version ersetzt wird?
 - syntaktische Dimension: Wie sieht es mit der Binärkompatibilität von neuer Basisklasse und alten Kindklassen aus?
 - Muss die Kindklasse recompiliert werden?
 - Wenn nur Methoden vererbt werden: im Prinzip nein
 - Selbst bei Verschiebungen in der Vererbungshierarchie (Restrukturierung) oder Schnittstellen-Erweiterungen nicht
 - Grund: Methodenbindung erfolgt zur Laufzeit über die Dispatch-Tabellen der Klassen
 - semantische Dimension: Die Implementierung der Subklasse nimmt Bezug auf implementatorische (semantische) Details der Basisklasse
 - mit einer neuen Version der Basisklasse kann diese Voraussetzung hinfällig sein.
 - auch durch Recompilieren nicht aus der Welt zu schaffen

Basisklasse implementiert read/write auf abstrakter Ebene

```
abstract class Text {
    private char[] text = new char[1000]; // Textbuffer
    private int used = 0; // Position des letzten Textzeichens
    private int caret = 0; // Position des Cursors

    void setCaret(int pos) { caret=pos; }
    int caretPos() { return caret; }
    ...
    void write(int pos, char ch) {
        for (int i=used; i>pos; i--) { text[i]=text[i-1]; }
        used++; text[pos]=ch;
        if (caretPos()>=pos) setCaret(caret+1);
    }
    abstract int posToX(int pos);
    abstract int posToY(int pos);
    abstract int posFromCoords(int x, int y);
}
```

Subklasse implementiert View mit Cursor

```
class SimpleText extends Text {  
    private int cacheX = 0; private int cacheY = 0;  
  
    void setCaret(int pos) { // überschriebene Methode  
        int old = caretPos();  
        if (old != pos) { hideCaret(); super.setCaret(pos); showCaret(); }  
    }  
    int posToX(int pos) {...}  
    int posToY(int pos) {...}  
    int posFromCoords(int x, int y) {...}  
    void hideCaret() { ... }  
    void showCaret() { ... }  
}
```

Neue Version der Klasse Text:

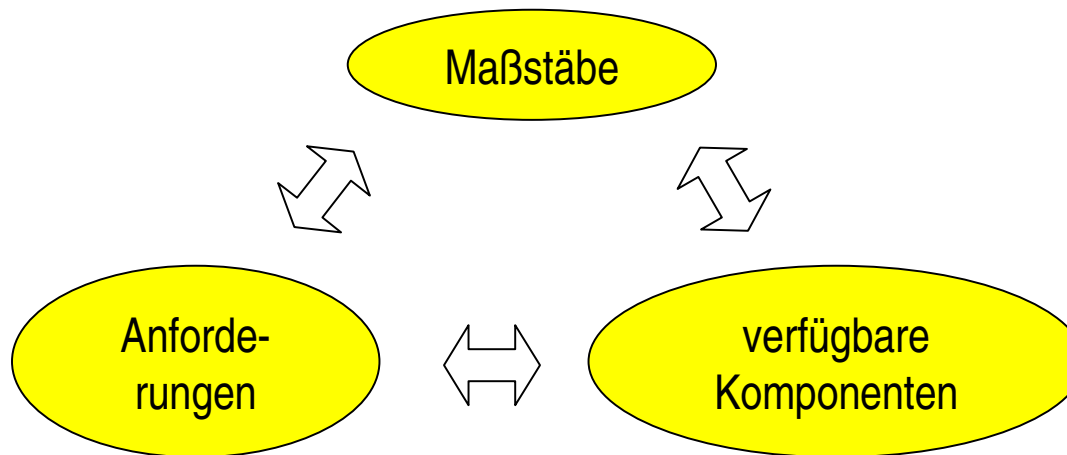
```
abstract class Text {  
    ...  
    void write(int pos, char ch) {  
        for (int i=used; i>pos; i--) { text[i]=text[i-1]; }  
        used++; text[pos]=ch;  
        if (caret >= pos) caret++;  
    }  
}
```

Effekt: Nach *write* steht der grafische Cursor an der falschen Position, weil die Implementierung von *SimpleText* sich darauf verlassen hat, dass *Text* die Variable *caret* stets nur über *setCaret* manipuliert.

Wechsel der inneren Implementierung des Anbieters kann den Nutzer korrumpieren,
ohne den Kontrakt zu verletzen.

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

- 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

- 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