UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Vorlesung
Software aus Komponenten

2. Grundlagen

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

- Neben den Schnittstellen muss für Komponenten auch klar sein, welche Umgebungsbedingungen für ihre Entfaltung erforderlich sind
  - Spezifikation der Anforderung an die Lokalisierungs-Umgebung
    - auch als Kontext-Abhängigkeit bezeichnet
  - enthält:
    - Komponenten-Modell = Spezifikation der Kompositions-Regeln
    - Komponenten-Plattform = Spezifikation der Regeln für Entpackung, Installation und Aktivierung von Komponenten
- heute existieren mehrere Komponentenwelten nebeneinander
  - sind selbst intern wieder fragmentiert nach Computer- und Netzwerk-Plattformen

Komponenten-Gewicht

- 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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

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 Kursors

    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);
}
```

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Subklasse implementiert View mit Kursor

```
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() { ... }
}
```

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

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++;
    }
}
```

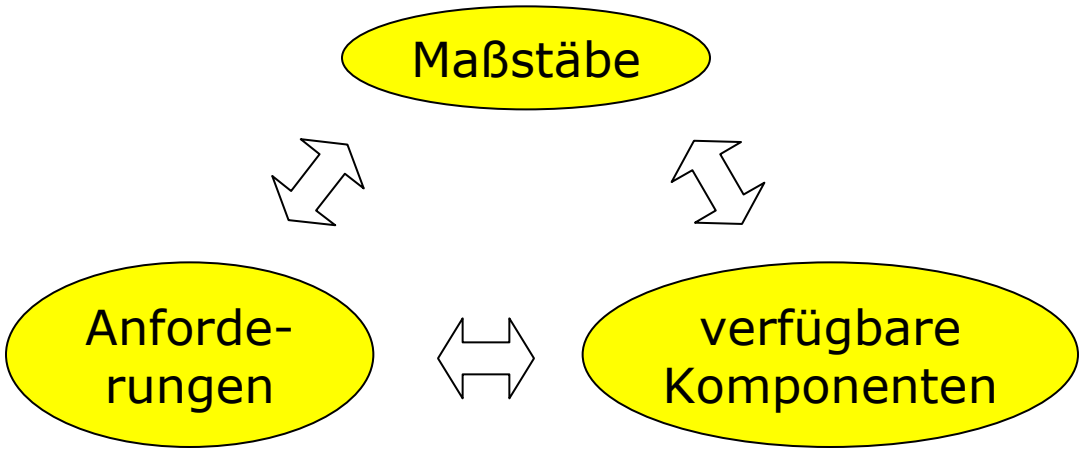
Effekt: Nach *write* steht der grafische Kursor an der falschen Position, weil die Implementierung von *SimpleText* sich darauf verlassen hat, dass *Text* die Variable *caret* stets nur über *setCaret* manipuliert.

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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Komponenten abgrenzen

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



UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

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