Vorlesung
Software aus Komponenten

2. Grundlagen

apl. Prof. Dr. Hans-Gert Gräbe
Wintersemester 2007/08

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

- Notwendigkeit, in diesem Universum von Namen und Konzepten **aufzuräumen**, Begriffe an definierte **Bedeutungen** zu binden und in ein gewisses **Ordnungsschema** zu bringen.

- Ansatz: Bedeutungen von Begriffen durch Angabe charakteristischer Eigenschaften fixieren

Komponentendefinition

Charakteristische Eigenschaften einer Komponente:

1. Einheit unabhängiger Packung
2. Einheit der Komposition durch Dritte
3. ohne (extern beobachtbaren) Zustand

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Objektdefinition

Charakteristische Eigenschaften eines Objekts:

1. Einheit der Instanziierung mit (eindeutiger) Identität
2. kann (extern beobachtbaren) Zustand besitzen
3. kapselt Zustand und Verhalten

Folgerungen:

Einheit der Instanziierung

- partielle Instanziierung nicht möglich

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

eindeutige Identität

- Zustand für jedes Objekt individuell, kann sich im Laufe des Objekt-Lebenszyklus ändern
- einzig garantiert persistente Eigenschaft eines Objekts ist dessen abstrakte Identität
  - Werner Brösels Auto bleibt W.B. Auto, auch wenn W.B. im Laufe der Zeit alle Teile daran ausgetauscht hat (und das Auto von seinem Original nichts mehr hat als eben diese abstrakte Identität)

Einheit der Instanziierung

- Plan muss Zustandsraum, Anfangszustand und (anfängliches) Verhalten eines neuen Objekts beschreiben
- Plan muss vor der Existenz der Objekte existieren
- Plan kann explizit sein = Klasse
- Plan kann implizit sein = Prototyp-Objekt
- Initialzustand muss valide sein, kann aber von weiteren Parametern abhängen
  - Code zur Erzeugung kann statisch sein = Konstruktor
  - Code zur Erzeugung kann selbst Objekt sein = factory object
  - Objekte können von anderen Objekten erzeugt werden = factory method

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Komponenten und Objekte

- Komponenten entfalten ihre Wirkung in der Regel über Objekte

  Eine Komponente besteht also meist aus mehreren Klassen oder unveränderlichen Prototyp-Objekten sowie weiteren Komponenten-Ressourcen
- Komponenten können auch vollkommen anders realisiert sein
- Export von Objekt(referenzen) bedeutet nicht, dass es intern auch OO zugeht und kann (allein über Objektreferenzen) von außen auch nicht erforscht werden
- Komponenten und Klassen

  Komponenten können mehrere Klassen umfassen

  eine Klasse kann nicht über mehrere Komponenten verteilt sein

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

- Komponenten und Vererbung
  - Vererbung ist über Komponentengrenzen hinweg möglich
    - muss aber in der Komponenten-Schnittstelle über eine Import-Deklaration explizit verankert sein
  - Vererbung von Spezifikationen (Schnittstellen-Vererbung)
    - wesentliche Technik, um Korrektheit über Komponentengrenzen hinweg zu garantieren
    - gängiger Weg zur Operationalisierung von Standards
  - Vererbung von Implementierungen
    - relativ schwierige Problematik über Komponentengrenzen hinweg
    - Thema greifen wir später noch mal auf

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Moduln

- Komponenten sind eher Moduln ähnlich. Wo sind die Unterschiede?
- Typische Eigenschaften von Moduln:
  - separate Compilierbarkeit
  - Mechanismen zur Typprüfung über Modulgrenzen hinweg
- Behauptung [Meyer 1988]: Klassen sind das bessere Modulkonzept
  - Sicht auf Moduln als ADT
  - Klasse = ADT + Polymorphie + Vererbung
  - Aber: Moduln unterstützen keine Instanziierung
    - statische Klassen als Modul-Imitation im Klassenkontext
- Moderne Sprachen (Modula-3, Oberon, C#) trennen das wieder
  - Moduln können mehrere Klassen umfassen
    - C#: Assemblies
    - Java: Imitation durch innere Klassen

- Im Gegensatz zu Klassen können Moduln die Grundlage für minimale Komponenten bilden

  Beispiel: math-Bibliotheken

  - sind eher funktionaler als objekt-orientierter Natur
- Moduln unterstützen keine Zuordnung persistenter unveränderlicher Ressourcen (jenseits hart im Code „verdrahteter" Konstanten)

  Komponenten werden mit lokal verfügbaren Ressourcen zu einer „lokalen Komponente" konfiguriert

  - die Beziehung zwischen einer Komponente und ihren Lokalisierungen ist komponenten-technologisch wichtig, im Modulkontext ausgeblendet
- Nicht jeder Modul geht als Komponente durch

  erlaubt: globale Variable, statische Abhängigkeit von Implementierungen in anderen Moduln (= zustandsbehaftet)
- Zusammenfassung: Modularität ist eine Voraussetzung für Komponenten-Technologie
  - Bindung im Modul so hoch wie möglich, Kopplung zwischen Moduln so gering wie möglich [Parnas 1972]
  - selbst das ist heute noch längst nicht Standard

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Whitebox und Blackbox

- Thema: Sichtbarkeit der Implementierung hinter der Schnittstelle
- Blackbox: Der Nutzer weiß nichts über die Interna

  Konzept der Komponente als Menge von Schnittstellen und deren Spezifikation
  - Bsp: API-Programmierung
- Whitebox: Die Schnittstelle kontrolliert die Zugriffsrechte, Interna sind aber prinzipiell bekannt

  Konzept der Komponente als Software-Fragment
  - schwache Form: durch Studium zusätzliche Informationen über das interne Verhalten der Komponente bekommen (glassbox)
  - starke Form: Implementations-Vererbung

  Entwickler studieren die Implementierung
  - Problem des Release-Wechsels

  Viele Klassenbibliotheken und Frameworks fallen unter diese Kategorie

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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Schnittstellen-Kontrakt

- Muss die Verwendung der Komponente in einem Produktiv-System genau beschreiben
- Aspekte:
  - Schnittstellen im engeren Sinne
  - Entpackung, Konfiguration, Installation der Komponente
  - Instanziierung und Beschreibung des Verhaltens dieser Instanzen durch ihre Schnittstellen (Laufzeitverhalten)
  - Beschreibung kollektiver Phänomene
    - Beispiel: Eine Stapel-Komponente hat Methoden `push` und `pop` und es muss klar sein, dass `pop(push(o))` wieder o zurückgibt
    - Das ist eine Bedingung an die Speicherschnittstelle, welche von der Komponente importiert wird

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

- Hier ausreichend: Schnittstelle als der Zugriffspunkt der Komponente

  Zugriffspunkt = spezieller Dienst samt Kontrakt, der von der Komponente angeboten wird

  Kontrakt als Interaktions-Basis der sonst unabhängigen Seiten Komponenten-Entwickler und Komponenten-Nutzer
- mehrere Schnittstellen = mehrere Zugriffspunkte = verschiedene Klienten

  ökonomischer Grund: Skalen-Effekt
- ein Zugriffspunkt oder die Summe der Zugriffspunkte kann über Markterfolg entscheiden

  wenn keiner der beiden Effekte erzielt werden kann, dann lohnt es nicht, die Software in Komponentenform zu bringen

  Komponenten ohne Markt können im Rahmen einer bereits eingesetzten Komponenten-Plattform als Lückenschluss sinnvoll sein

  ° Skaleneffekt indirekt: Komponente in Kombination mit der Plattform

- 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

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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

- 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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

- „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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

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

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

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.