Vorlesung
Software aus Komponenten

2. Grundlagen

Prof. Dr. Hans-Gert Gräbe
Wintersemester 2015/16

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 als Prinzip – drei wesentliche Facetten
  - Subtypen: Vererbung von Kontrakt-Fragmenten
    - Schnittstellen-Vererbung, Java: *implements*
    - wesentliche Technik, um Korrektheit über Komponentengrenzen hinweg zu garantieren
    - gängiger Weg zur Operationalisierung von Standards
  - Subklassen: Vererbung von Implementations-Fragmenten
    - Implementations-Vererbung, Java: *extends*
    - relativ schwierige Problematik über Komponentengrenzen hinweg
    - Problem der fragilen Basisklasse
  - Substituierbarkeit: Vererbung funktionaler Eigenschaften

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 (mglw. implizit) Bezug auf implementatorische (semantische) Details der Basisklasse
    - Mit einer neuen Version der Basisklasse kann diese Voraussetzung hinfällig sein.
    - Das ist 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]; // Textpuffer [0..(used-1)]
    private int used = 0; // Position nach dem letzten Textzeichen
    private int cursor = 0; // Position des Kursors

    void setCursor(int pos) { cursor=pos; }
    int cursorPos() { return cursor; }
    ...
    void write(int pos, char ch) { // füge Zeichen ch an Stelle pos im Puffer ein
            for (int i=used+1; i>pos; i--) { text[i]=text[i-1]; }
            used++; text[pos]=ch;
            if (cursorPos()>=pos) setCursor(cursor+1);
    }
    abstract int posToX(int pos);
    abstract int posToY(int pos);
    abstract int posFromCoords(int x, int y);
}
```

Subklasse implementiert View mit Kursor

```
class SimpleText extends Text {
    private int cacheX = 0; private int cacheY = 0;

    void setCursor(int pos) { // überschriebene Methode
            int old = cursorPos();
            if (old != pos) { hideCursor(); super.setCursor(pos); showCursor(); }
    }
    int posToX(int pos) {...}
    int posToY(int pos) {...}
    int posFromCoords(int x, int y) {...}
    void hideCursor() { ... }
    void showCursor() { ... }
    // Methode write wird unverändert geerbt und setzt den Kursor richtig.
}
```

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 (cursor >= pos) cursor++; // Position wird aktualisiert ohne setCursor!
    }
}
```

Effekt: *write* setzt nun den grafischen Kursor nicht mehr. Dieser steht an der falschen Position, weil die Implementierung von *SimpleText* sich darauf verlassen hat, dass *Text* die Variable *cursor* stets nur über die Methode *setCursor(...)* manipuliert.

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

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
- Historisch Strukturierungsinstrument alternativ zu Klassen, nicht instanziierbar
- Behauptung [Meyer 1988]: Klassen sind das bessere Modulkonzept, ein separates Modulkonzept wird nicht benötigt, denn Klassen sind mehr: Klasse = Schnittstelle + Polymorphie + Vererbung

Moderne Sprachen (Modula-3, Oberon, C#) trennen das dennoch wieder

- Moduln als semantische Strukturierungseinheit zur Code-Zusammenfassung **oberhalb** von Klassen
  - C# und .NET: Assemblies
  - Java Packages liefert als vergleichbares Konzept funktional deutlich weniger

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

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

„undokumentierte Features“

- Auf Komponenten-Verhalten wird oft jenseits der Spezifikation auch aus Beobachtung des laufenden Betriebs geschlossen.
  - Siehe Beispiel „fragile Basisklasse“
- Ist auch typisches Ergebnis eines „Debugging“-Prozesses bei Fehlersuche
- Kleine Fehler sind ökonomischer auf der Nutzerseite als auf der Anbieterseite zu beheben
- Open-Source-Ansatz