

Vorlesung Software aus Komponenten

2. Grundlagen

Prof. Dr. Hans-Gert Gräbe
Wintersemester 2013/14

Komponentendefinition

Charakteristische Eigenschaften einer Komponente:

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

Objektdefinition

Charakteristische Eigenschaften eines Objekts:

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

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

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.

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 caret = 0; // Position des Cursors  
  
    void setCaret(int pos) { caret=pos; }  
    int caretPos() { return caret; }  
    ...  
    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 (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() { ... }  
    // Methode write wird unverändert geerbt und setzt den Cursor 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 (caret >= pos) caret++; // Position wird aktualisiert ohne setCaret!  
    }  
}
```

Effekt: *write* setzt nun den grafischen Cursor nicht mehr. Dieser steht 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.

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 instanziiierbar
- 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

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

Komponenten als sozio-technische Artefakte

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 jeden tragfähigen Komponentenbegriff nicht nur im Software-Bereich

Schnittstellen-Kontrakt

- Muss die Verwendung der Komponente in einem Produktiv-System genau beschreiben
- Technische Aspekte:
 - Schnittstellen im engeren Sinne
 - Entpackung, Konfiguration, Installation der Komponente
 - Instanziierung und Beschreibung des Verhaltens der durch die Komponente erzeugbaren Objekte durch ihre Schnittstellen (Laufzeitverhalten)
 - Beschreibung der Ressourcenanforderungen der Komponente sowie 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

- 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 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

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

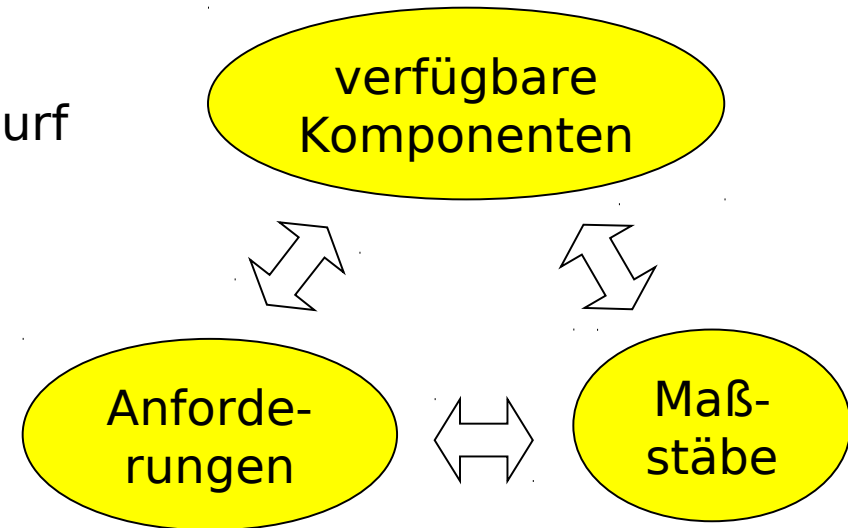
- Schnittstellenversionen müssen klar als kompatibel oder klar als veraltet (deprecated) deklariert werden können
- Ansatz der unveränderlichen Schnittstellen-Spezifikation (immutable interfaces)
 - Neue Versionen nur als neue Schnittstellen
 - Veraltete Schnittstellen werden einfach nicht mehr unterstützt
 - 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