Softwaretechnik 2015/2016

PST Lehrstuhl

Prof. Dr. Matthias Hölzl

Joschka Rinke

AGENDA

Übung 9:
10.12.2015

- **Fragen**
- **Anmeldung zur Klausur über UniWorX**
- **Besprechung Blatt08**

Aufgabe 1 – Composite Pattern

Composite Pattern:

- **Ideal zur Darstellung von Teil-Ganzes-Hierarchien**
- **Repräsentiert durch eine Baumstruktur und…**
- **…enthält Leaf- und Composite-Elementen, die beide Components sind**
- **Leaf-Elemente sind Blätter und haben keine Kind-Elemente**
- **Composite-Elemente haben Kinder, die wiederum Components sind**

Aufgabe 1 – Composite Pattern

Beispielsituationen:

Mitarbeiter → vorgesetzter Mitarbeiter
→ normaler Mitarbeiter
→ etc.

Mensch → Europäer
→ Deutscher
→ Bayer
→ Oberbayer
→ etc.

Aufgabe 1 – Composite Pattern

Beispielobjekte:

Bild → **geometrische Figuren**
→ **Kreis**
→ **Gerade**
→ **Punkt**

Benutzeroberfläche (GUIs)
→ **Frame**
→ **Panel**
→ **Button**

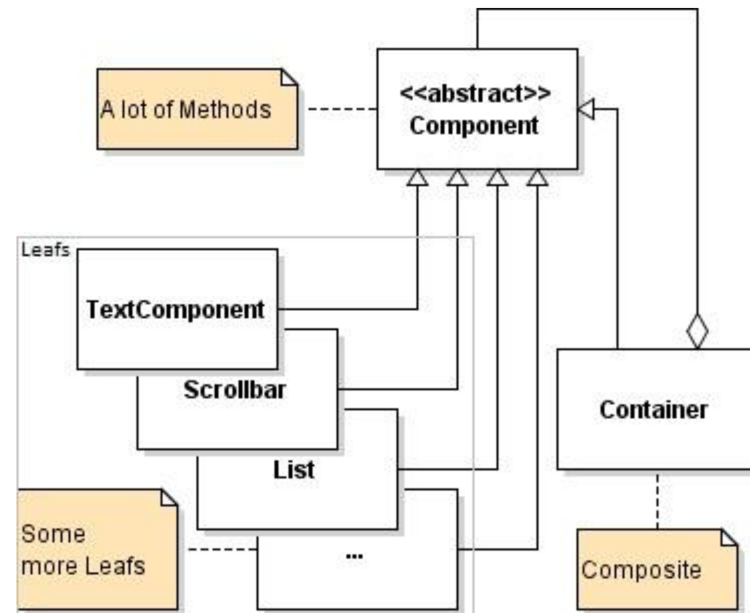
Aufgabe 1 – Composite Pattern

Bekannter Anwendungsfall:
JavaSwing

JavaSwing verwendet JavaAWT

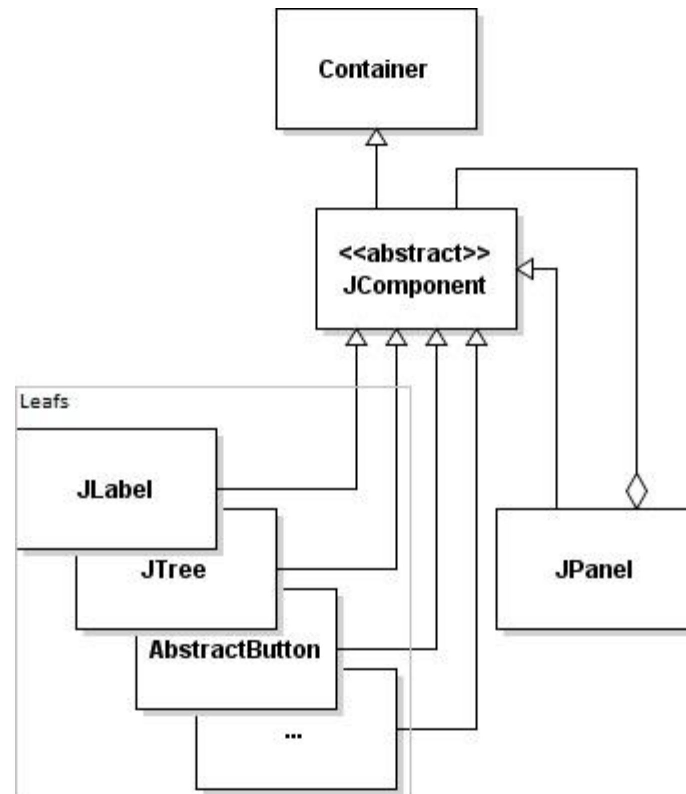
JavaSwing besteht aus:

- **Component**
- **Container**
- **…**



Anmerkung:
AWT verfolgt Strategie der Sicherheit, d.h.
dass Component nur die Methoden (dennoch sehr viele) enthält,
die für alle Componenten sinnvoll sind.

Aufgabe 1 – Composite Pattern



Bekannter Anwendungsfall:
JavaSwing

JavaSwing besteht aus:

- **…**
- **Container**
- **JComponent**

Anmerkung:
Swing setzt auf AWT-API auf und verfolgt die Strategie der Transparenz, d.h. dass JComponent als Schnittstelle zu allen Swing-Komponenten auch die Methoden zur Verwaltung von Componenten durch Composites enthält.

Aufgabe 1 – Composite Pattern

Zusammenspiel zwischen AWT und Swing:

A lot of Methods

<<abstract>>
Component

Leafs

TextComponent

Scrollbar

List

...

Some more Leafs

Container

Composite

+

Container

<<abstract>>
JComponent

Leafs

JLabel

JTree

AbstractButton

...

JPanel

Aufgabe 1 – Design Patterns

Wir betrachten die Strategie der Transparenz für die Implementierung der Methoden in Java Swing. Was passiert wenn eine Verwaltungsmethode, die für Composites gedacht ist, auf einem Leaf aufgerufen wird?

- **Es gibt einen Fehler**
- **Es passiert nichts**
- **Der Aufruf wird an das Parent-Composite delegiert**
- **Das geht nicht, da solche Methoden nur in den Composites verfügbar sind**

Aufgabe 1 – Design Patterns

Was ist Leaf, was ist Composite, was ist Component?
Component: Employee
Leaf: Assistant
Composite: Supervisor, Manager

Implementierung abhängig davon, welche Strategie verfolgt wird...
Sicherheit:
→ Nur die Methoden, die alle Components benötigen
→ Implementierung der Methoden auch in der Component-Klasse möglich

Transparenz:
→ Alle Methoden, egal ob sie von den Leafs nicht benötigt werden
→ Implementierung der Component-Klasse z.B. als Interface

Aufgabe 1 – Design Patterns

Vorteile:

- **Einfache Repräsentation von verschachtelten Strukturen, d.h. es können entweder primitive (Leafs) oder zusammengesetzte Objekte (Composites) a.d. gleichen Stelle stehen**
- **Vereinfachter Clientcode**
- **Elegantes Traversieren dank Baumstruktur**
- **Sehr leicht erweiterbar → neue Elemente können hinzugefügt werden OHNE dass an bestehenden Elementen etwas beachtet oder geändert werden muss**

Nachteile:

- **Evtl. Übergeneralisierung**
- **Leaf-Klasse muss evtl. Methoden implementieren, die sie nicht braucht (je nach Architektur)**
- **Man muss sich relativ früh auf ein Schema für die allgemeine Componentschnittstelle festlegen**

Aufgabe 2 – Reflection Pattern

Reflection Pattern (Wiederholung):
Architekturmuster, das zwei Ebenen definiert:

- **Meta-Level**
- **Base-Level**

Durch Verwendung des Reflection Pattern ist es möglich, dass Änderungen an einem laufenden System vorgenommen werden können, ohne dieses herunterfahren zu müssen.

Metaobject-Protocol (MOP) als Interface zwischen Meta- und Base-Level

Aufgabe 2 – Reflection Pattern

Meta-Level: Informationen über Systemeigenschaften (Struktur, Verhalten, Architektur)

Base-Level: Anwendungslogik → Änderungen (Intercession) in der Meta-Ebene beeinflussen das Base-Level (Logik)

Dynamische Änderungen:

- **Änderungen auf Metaebene am Datenmodell zur Laufzeit**

Aufgabe 2 – Reflection Pattern

Metaobject:
Ein Objekt existiert nur zur Laufzeit und repräsentiert ein Programmiersprachenkonstrukt (Klasse, Methode, Field, etc.)

Metaobject-Protocol (MOP):
Interface für Änderungen auf der Meta-Ebene
➜ Verantwortung für den korrekten Umgang mit Änderungen liegt bei MOP
➜ Klassen hinzufügen/ändern
➜ Methoden hinzufügen/ändern

Aufgabe 2 – Reflection Pattern

Fortsetzung Metaobject-Protocol (MOP):
Änderungen durch MOP haben Auswirkungen auf Base-Level und somit auf mögliches Systemverhalten
Achtung: Reflection ist sehr mächtig
- ➔ **Zugriff kann über MOP reguliert werden**
- ➔ **Zugriff kann auch vereinfacht werden, indem über das MOP zusammengehörige Meta-Level Objekte verändert werden**

MOP als Singleton um Zugriffskonflikte auf Objekte der Meta-Ebene zu vermeiden

Aufgabe 2 – Reflection Pattern

Intercessions sind Änderungen an der Objektstruktur der Meta-Ebene zur Laufzeit.
Das heißt, dass eine Klasse X im laufenden System existiert, dass die Klasse X verändert wird und dass dann durch eine MOP-Funktion die veränderte Klasse X dem Metamodell hinzugefügt wird und die alte Klasse X ersetzt.
Ist das in Java ohne weiteres möglich?

Aufgabe 2 – Reflection Pattern

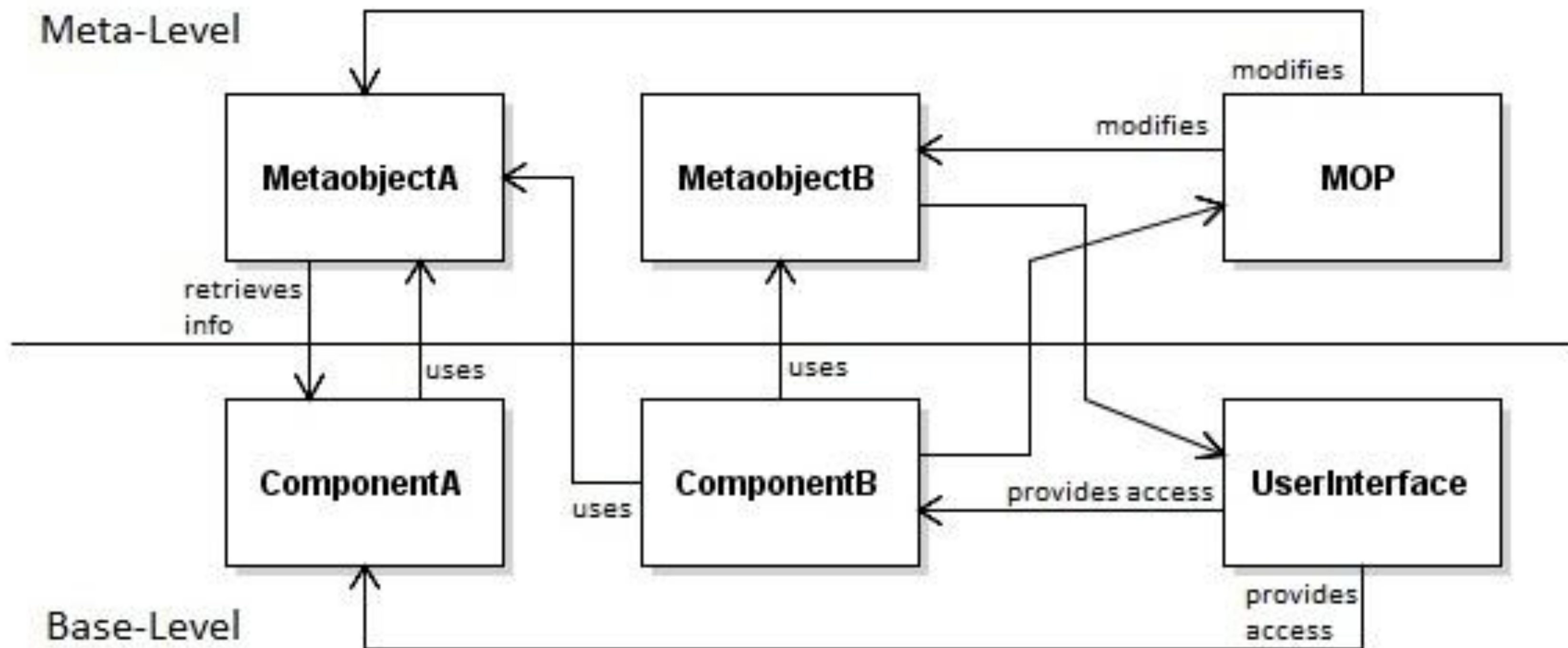
Reflection in Java:

- **Inspection von Klassen, Methoden und Fields zur Laufzeit möglich, ohne dass die Namen zur Compilezeit klar sind**
- **Instantiierung neuer Objekte und Aufrufen von Methoden**

→ Verändern (Incession) existierender Objekte auf Meta-Ebene ist nicht vorgesehen

Aufgabe 2 – Reflection Pattern

Skizze für Zusammenhänge zwischen Meta- und Base-Level (kein UML):



Aufgabe 2 – Reflection Pattern

Abhängig von verwendeter Sprache

Vorteile:

- **kann Speicher sehr effizient nutzen (z.B. durch geschicktes MOP)**
- **kann Robustheit erhöhen**

Nachteile:

- **Ist ohne Optimierung durch MOP speicherintensiver…**
- **…und langsamer**
- **kann zu sehr komplizierten Fehlern führen**

➔ stark Abhängig von der Qualität des Codes