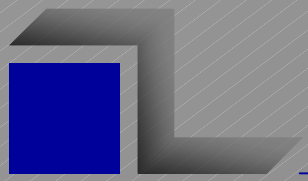
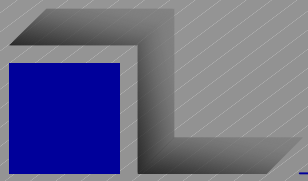


Design Pattern

- Motivation, Beispiel
- Definition
- "Das" Buch der *Gang of Four*
- Ausführliches Beispiel: Facade
- Beispiele
  - Singleton
  - Composite
  - Observer
- Aufgabe

© Prof. Dr. Peter Knauber
HS Mannheim



Motivation

- Wiederverwendung ist etwas Gutes ... !!!
- Wiederverwendung (auch: Verständlichkeit, Änderbarkeit, Portierbarkeit etc.) wird auf Design-Ebene ermöglicht (oder gar nicht)
- Wiederverwendbares (objektorientiertes) Design ist schwer
- Design-Erfahrung hilft ...

- Design-Pattern (Entwurfsmuster)
  - benennen,
  - erklären und
  - bewerten

  wichtige und wiederholt eingesetzte Design-Elemente objektorientierter Systeme in einer effektiv nutzbaren Form.

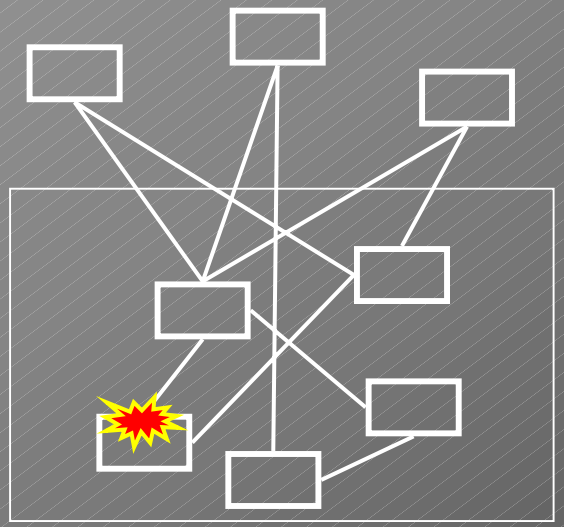
  Gamma, Helm, Johnson, Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, 1994

© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein Pattern: Facade

Design Pattern

- Motivation
- Definition
- "Das" Buch
- Facade
- Singleton
- Composite
- Observer
- Aufgabe



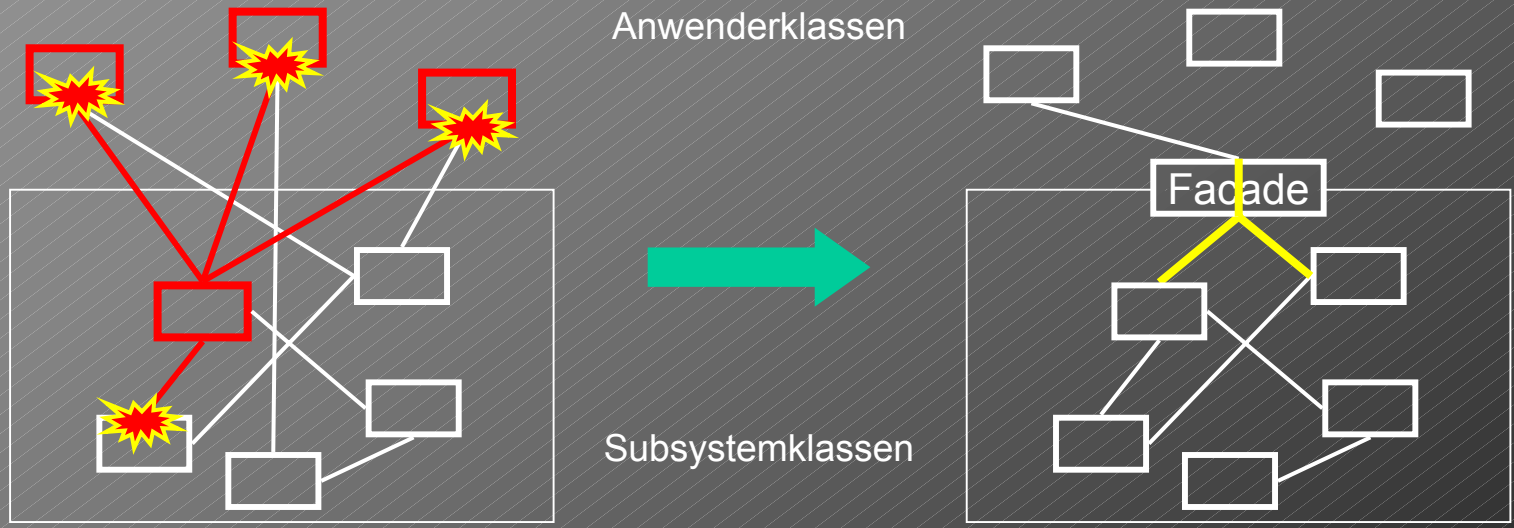
Anwenderklassen

Subsystemklassen

Ein Subsystem ist ein System, das auch ohne die Dienste (*Services*) anderer (Sub-)Systeme existieren und arbeiten kann

© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein Pattern: Facade

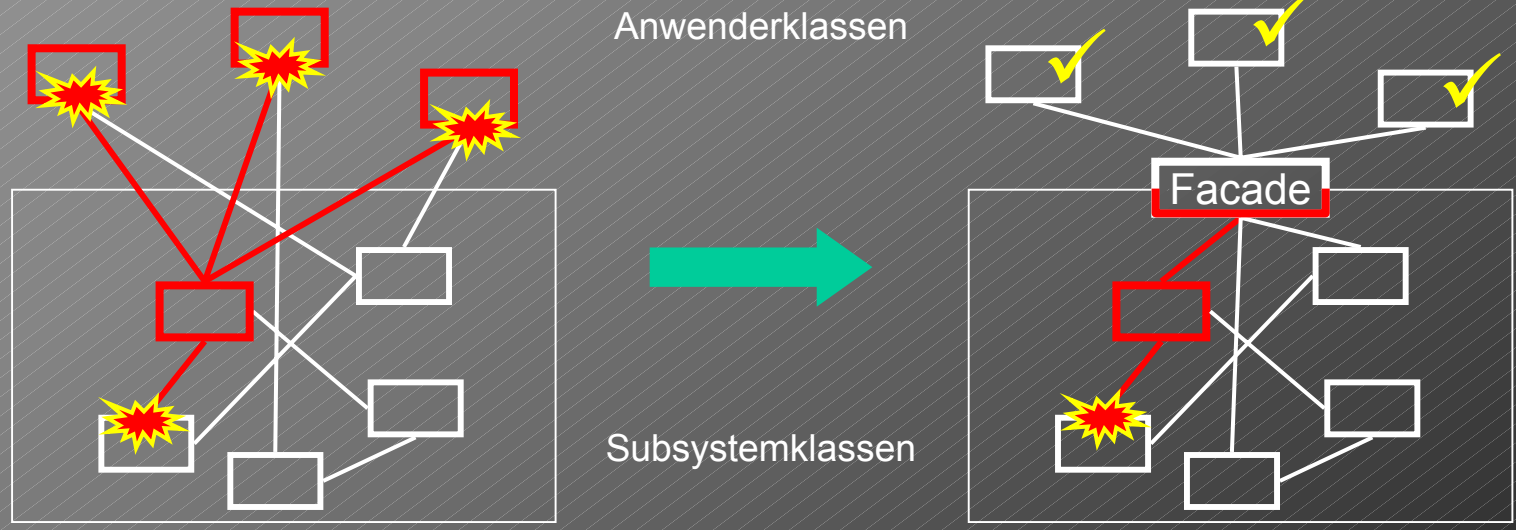


© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein Pattern: Facade

Design Pattern

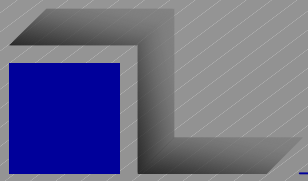
- Motivation
- Definition
- "Das" Buch
- Facade
- Singleton
- Composite
- Observer
- Aufgabe



Zweck der Facade:

- Bietet eine einheitliche, stabile Schnittstelle zu einer Menge von Schnittstellen eines Subsystems
- Macht die Benutzung des Subsystems einfacher

© Prof. Dr. Peter Knauber
HS Mannheim



Definition

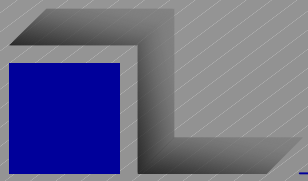
- Jedes Pattern (Muster) beschreibt ein Problem, das in unserem Umfeld immer wieder auftritt, und beschreibt dann den Kern einer Lösung für dieses Problem und einer Art und Weise, dass diese Lösung millionenfach wiederverwendet werden kann, ohne dass sie auch nur zweimal identisch ist.

  Alexander et al.: *A Pattern Language*, 1977

- Design-Pattern beschreiben eine erprobte Lösung für ein typisches Problem in einem gegebenen Kontext

  L. Bass, P. Clements, and R. Kazman: *Software Architecture in Practice*, 1998

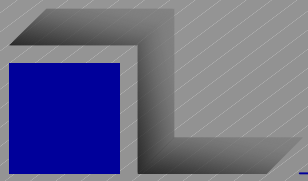
© Prof. Dr. Peter Knauber
HS Mannheim



(Mindest-)Dokumentation eines Patterns

- Name
  - Identifikation
  - Teil des "Design-Vokabulars"
- Problembeschreibung
  - Kontext für die Pattern-Anwendung
  - Beispiele
    - → Konkrete Algorithmen und ihre Repräsentation mittels Objekten
    - → Vorbedingungen für den Einsatz
- Lösungsbeschreibung
  - Klassen, Beziehungen, Verantwortlichkeiten, Zusammenarbeit
  - Kein konkretes Design!
- Konsequenzen
  - Beispiele
    - → Laufzeit- oder Speicherplatzbedarf
    - → Anzahl / Größe von Klassen / Objekten

© Prof. Dr. Peter Knauber
HS Mannheim



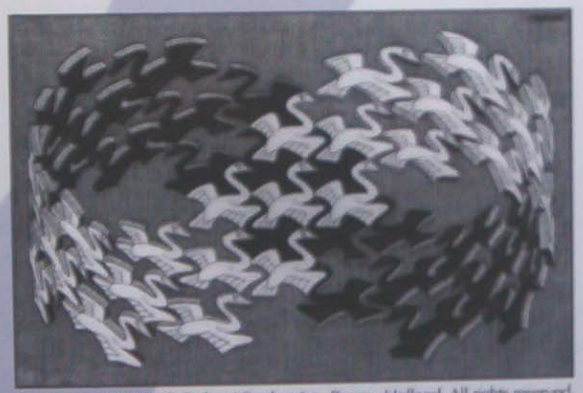
Fragen?

© Prof. Dr. Peter Knauber
HS Mannheim

Design Patterns

Elements of Reusable Object-Oriented Software

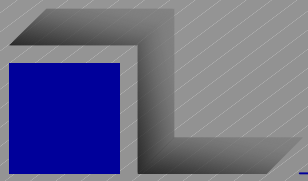
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Wichtig

Es werden keine neuen Design-Konstruktionen erfunden, sondern existierende (und von erfahrenen Designern eingesetzte) Lösungen für typische Probleme mit ihren Vor- und Nachteilen dokumentiert

© Prof. Dr. Peter Knauber
HS Mannheim

Verwendungszweck von Pattern

- Erzeugen (*creational*)
  - Abstraktion vom Instanziierungsprozess
  - Macht ein System unabhängig davon, wie seine Objekte instanziiert, zusammengesetzt und repräsentiert werden
- Struktur (*structural*)
  - Zusammensetzen von Klassen/Objekten zu größeren Strukturen, um neue Funktionalität zu realisieren

  → Wiederverwendbarkeit
- Verhalten (*behavioral*)
  - Zusammenarbeit, Arbeitsteilung, Verantwortlichkeiten zwischen verschiedenen Klassen oder Objekten

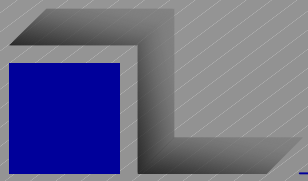
  → Flexibilität, Wiederverwendbarkeit

© Prof. Dr. Peter Knauber
HS Mannheim

Pattern-Dokumentation nach der *Gang of Four* 1/2

- Name und Klassifikation
  - Sprechender Name (erweitert das Vokabular)
  - Klassifikation: creational, structural, behavioral
- Absicht / Aufgabe
  - Was tut das Pattern?
  - Welches Problem löst es?
- Auch bekannt als
  - Alternative Namen
- Motivation
  - Beispielszenario für Problem und Lösung mittels Pattern
- Anwendbarkeit
  - Beispielsituationen
- Struktur
  - Graphische Darstellung der Struktur und der Interaktionen

© Prof. Dr. Peter Knauber
HS Mannheim



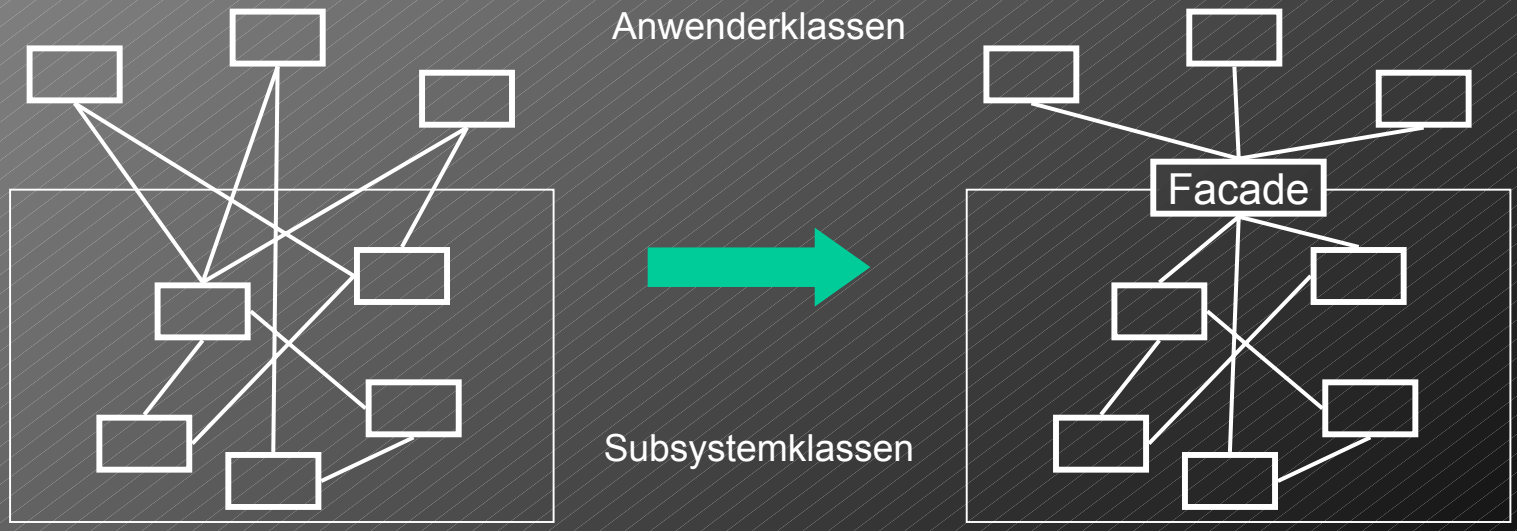
Pattern-Dokumentation nach der *Gang of Four* 2/2

- Beteiligte (Klassen und Objekte)
  - Klassen/Objekte, die zum Pattern gehören und ihre Aufgaben
- Zusammenarbeit
  - Art und Weise, wie die Beteiligten ihre Aufgaben erfüllen
- Konsequenzen
  - Positive und negative Konsequenzen der Pattern-Verwendung
- Implementierung
  - Techniken und Tipps für die Implementierung
  - Sprachspezifische Besonderheiten
- Code-Beispiel
  - In C++, Smalltalk
- Bekannte Anwendungen
  - Beispielsituationen, in denen das Pattern eingesetzt wird
- Pattern mit Bezug zu diesem Pattern
  - Gemeinsamkeiten mit und Unterschiede zu anderen Pattern

© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein Pattern: Facade

- Name: Facade
  Kategorie: Structural
- Aufgabe:
  Bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems; macht die Benutzung des Subsystems einfacher
- Motivation, Struktur:



© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein Pattern: Facade

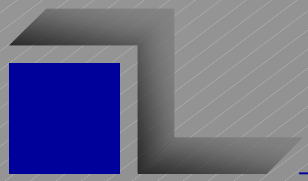
- Anwendbarkeit
  - Wenn eine einfache Schnittstelle zu einem komplexen Subsystem bestehen soll
  - Wenn viele Abhängigkeiten zwischen Anwendungsklassen und Subsystemklassen bestehen
  - Wenn Subsysteme in Schichten (Layern) angeordnet werden sollen
- Beteiligte
  - Facade
    - → Weiß, welche Subsystemklassen für eine Operation verantwortlich sind
    - → Delegiert Anfragen von Anwendern zu Objekten geeigneter Subsystemklassen
  - Subsystemklassen
    - → Implementieren die Funktionalität des Subsystems
    - → Bearbeiten die von der Facade zugewiesenen Anfragen
    - → Kennen die Facade nicht (halten keine Referenzen darauf)

© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein Pattern: Facade

- Zusammenarbeit
  - Anwendungsklassen kommunizieren mit der Facade, welche die Anfragen an die entsprechenden Subsystemobjekte weitergibt
  - Anwendungsklassen, welche die Facade nutzen, greifen nicht direkt auf die Subsystemobjekte zu
- Konsequenzen (Vorteile)
  - Das Subsystem wird einfacher zu benutzen, weil Anwendungsklassen nicht viele Subsystemklassen kennen und direkt darauf zugreifen müssen
  - Die Kopplung zwischen Anwendungsklassen und Subsystemklassen wird weniger stark; dadurch wird die Änderbarkeit der Subsystemklassen verbessert
  - Anwendungen, die das müssen, können nach wie vor direkt auf Subsystemklassen zugreifen

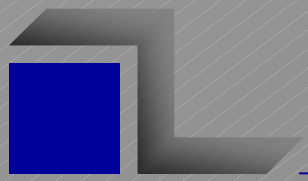
© Prof. Dr. Peter Knauber
HS Mannheim



Beispiel für ein Pattern: Facade

- Implementierung
  - Reduzieren der Kopplung (Coupling) zwischen Anwendungs- und Subsystemklassen: ...
  - Die Facade gehört zum public-Interface des Subsystems, die anderen Klassen können Subsystem-private sein (in Java gibt es dafür Packages): ...
- Code-Beispiel
  - Ausführliche Dokumentation eines Beispiels mit einer Compiler-Facade
- Bekannte Anwendungen
  - Das Compiler-System in ObjectWorks\Smalltalk
  - Browsen von Objekten zur Laufzeit im ET++ - Framework
- Pattern mit Bezug zu diesem Pattern
  - Abstract Factory ...
  - Mediator ...
  - Singleton ...

© Prof. Dr. Peter Knauber
HS Mannheim



Partner-Diskussion: Das Facade-Pattern

- Diskutieren Sie mit einem Partner
  - Finden Sie/konstruieren Sie zwei Anwendungsfälle für das Facade-Pattern
  - Einigen Sie sich darauf, welche Aufgaben die Facade dort übernehmen würde
  - In welchen Situationen bringt das Pattern dort Vorteile?
  - Gibt es in Ihren Anwendungsfällen auch Situationen, in denen das Pattern Nachteile bringt?
  - Warum ist es nicht möglich, Pattern "ein für alle mal" zu implementieren und als Bibliothek zur Verfügung zu stellen?

  Machen Sie sich Notizen!

- Dauer: 5 Minuten

© Prof. Dr. Peter Knauber
HS Mannheim

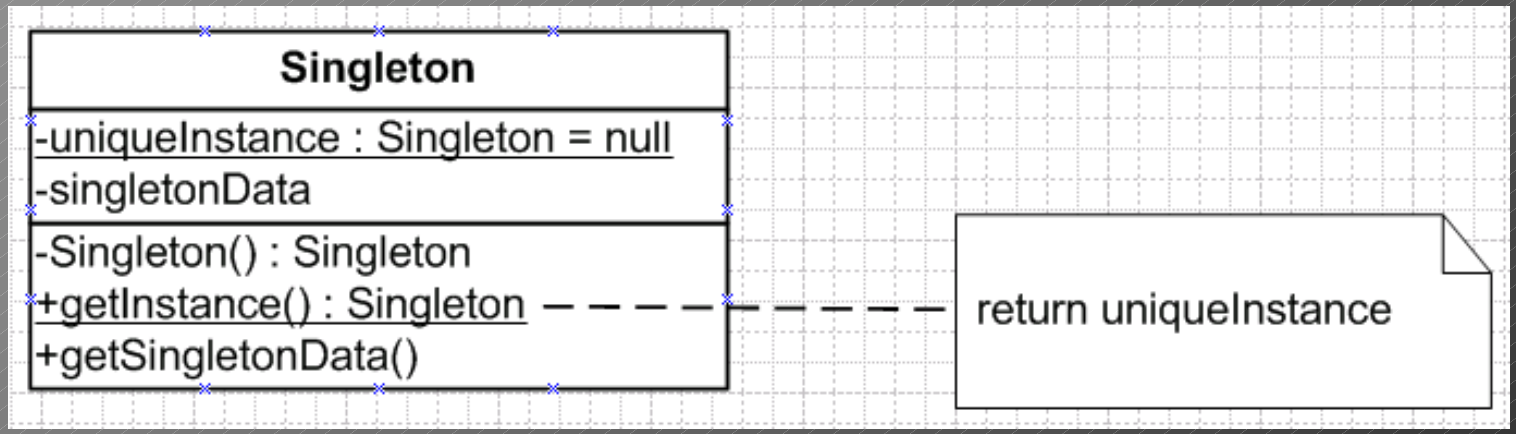
Beispiel für ein *creational* Pattern: Singleton

- Aufgabe
  Garantiert, dass eine Klasse genau eine Instanz hat und stellt den Zugriff darauf sicher
- Motivation
  - *Ein* Druckerserver auf einem Rechner, *ein* Window-Manager, *ein* Konfigurationsverwalter für ein Programm
  - Globale Variable stellen den Zugriff sicher, verhindern aber nicht die mehrfache Instanziierung
- Anwendbarkeit
  - Wenn es genau eine Instanz einer Klasse geben soll, auf die über einen einheitlichen Weg zugegriffen werden kann
  - Wenn die Instanz mittels Vererbung erweiterbar sein soll und eine erweiterte Instanz für Anwendungsklassen ohne Codeveränderung (in den Anwendungsklassen) nutzbar sein soll

© Prof. Dr. Peter Knauber
HS Mannheim

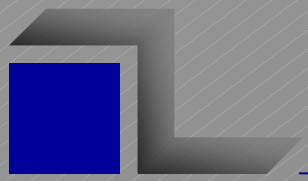
Beispiel für ein creational Pattern: Singleton

- Struktur



- Konsequenzen
  - Kontrollierter Zugriff auf die eine Instanz
  - "Schonen" des globalen Namensraumes
  - Erweiterung um Attribute und Methoden ist möglich
  - Anzahl erlaubter Instanzen ist einstellbar
  - Flexibler als Klassenmethoden (dynamische Bindung → erweiterbar, Anzahl kontrollierbar)

© Prof. Dr. Peter Knauber
HS Mannheim



Partner-Diskussion: Das Singleton-Pattern

- Diskutieren Sie mit einem Partner
  - Finden Sie/konstruieren Sie zwei Anwendungsfälle für das Singleton-Pattern
  - In welchen Situationen bringt das Pattern dort Vorteile?
  - Gibt es in Ihren Anwendungsfällen auch Situationen, in denen das Pattern Nachteile bringt?

  Machen Sie sich Notizen!

- Dauer: 3 Minuten

© Prof. Dr. Peter Knauber
HS Mannheim

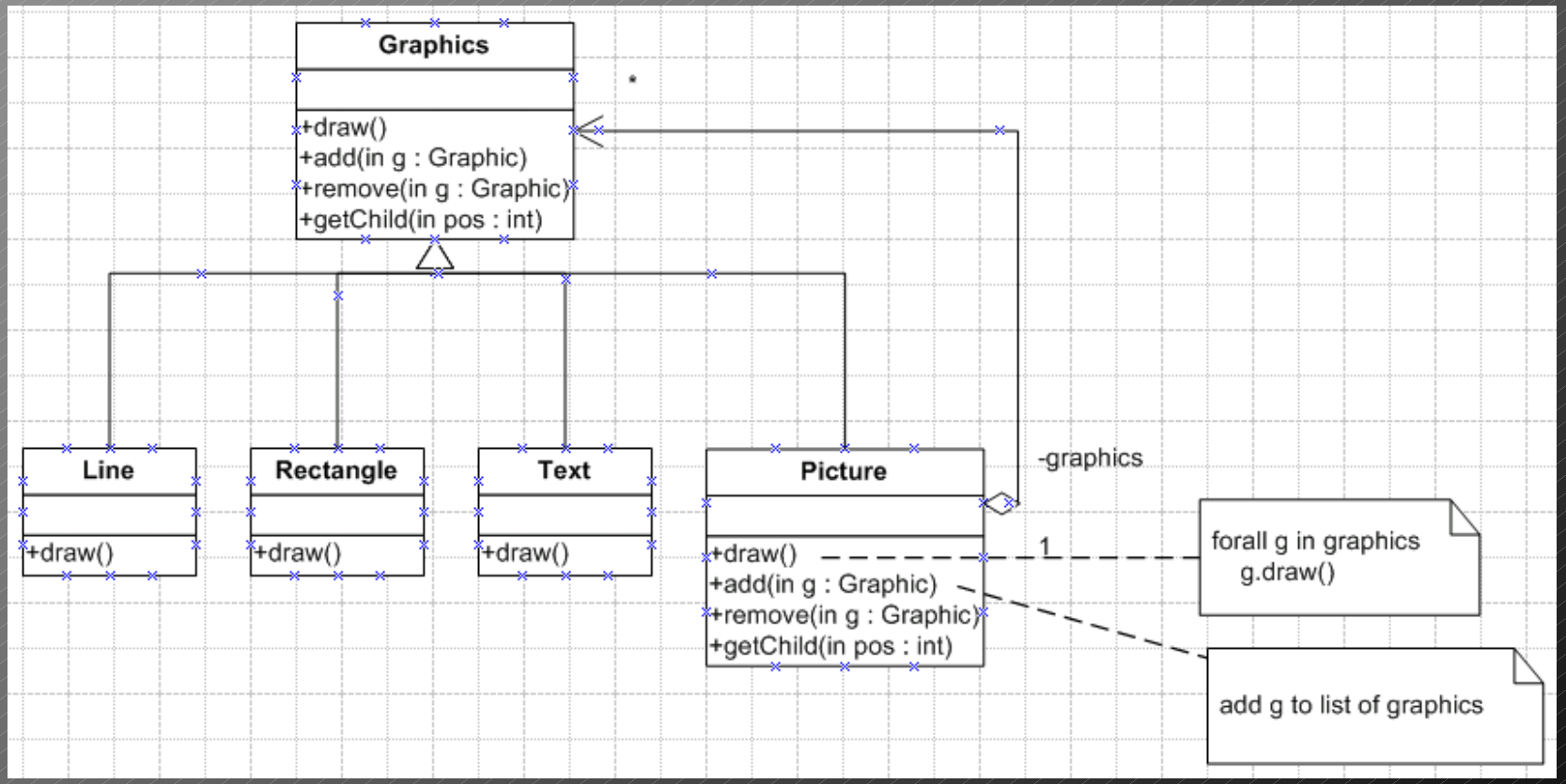
Beispiel für ein *structural* Pattern: Composite

- Aufgabe
  - Organisiert Objekte in Baumstrukturen für die Repräsentation von Teil-/Ganzes-Beziehungen
  - Erlaubt den gleichförmigen Zugriff auf atomare Einzelobjekte wie auf zusammengesetzte Objekte
- Motivation
  - Beispiel Graphikprogramm
    - → Einfache Objekte (*Primitive*) wie Linien und Texte sollen gruppierbar sein;
      der Benutzer möchte Gruppen genau wie Einzelobjekte behandeln
    - → Implementierungsidee: Klassen für Primitive + Klassen für Container
    - → Diese Unterscheidung macht den Programmcode sehr komplex
  - Lösung
    - → Eine abstrakte Oberklasse repräsentiert Primitive *und* Container
    - → Operationen von Primitiven werden von Containerobjekten an die enthaltenen Objekte delegiert

© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein *structural* Pattern: Composite

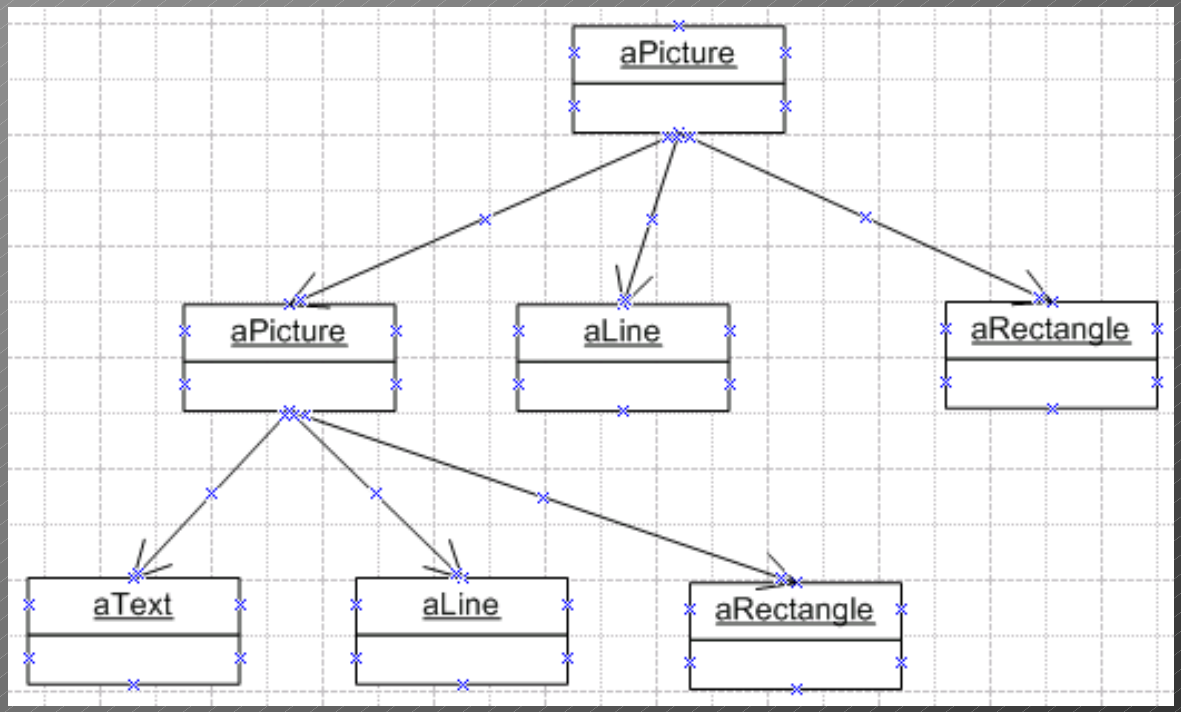
- Motivation (Fortsetzung): Klassen für das Graphik-Problem



© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein *structural* Pattern: Composite

- Motivation (Fortsetzung): eine typische Graphik

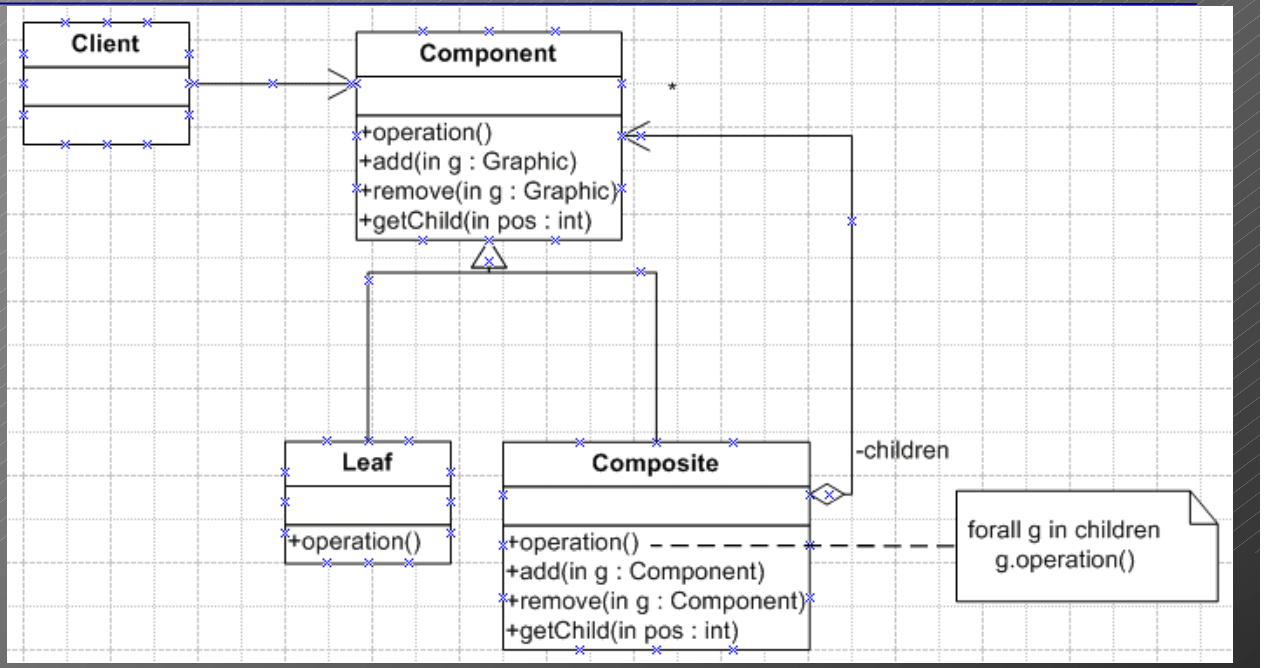


- Anwendbarkeit
  - Wenn Teil-/Ganzes-Hierarchien von Objekten zu repräsentieren sind
  - Wenn Anwendungsklassen den Unterschied zwischen atomaren und zusammengesetzten Objekten ignorieren sollen

© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein *structural* Pattern: Composite

- Struktur

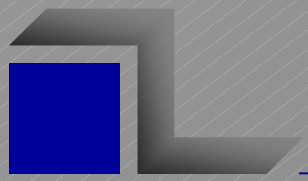


- Beteiligte
  - *Component*:
    - Definiert die gemeinsame Schnittstelle aller Objekte im Baum
    - Implementiert Default-Verhalten, wo möglich und sinnvoll
    - Definiert eine Schnittstelle für den Zugriff auf Kinder einer Komponente
  - *Leaf*:
    - Repräsentiert Blätter in der Baumhierarchie
    - Definiert das Verhalten von atomaren Objekten
  - *Composite*:
    - Repräsentiert Wurzel- und innere Knoten in der Baumhierarchie
    - Definiert das Verhalten dieser Knoten
    - Implementiert Kind-bezogene Operationen
  - *Client*: manipuliert Objekte mittels *Component*-Schnittstelle

Beispiel für ein *structural* Pattern: Composite

- Konsequenzen
  - Definiert Klassenhierarchien bestehend aus atomaren Objekten (*Primitiven*) und zusammengesetzten Objekten
    Verbirgt den Unterschied zwischen diesen Objekten vor Anwendungsklassen
  - Vereinfacht Anwendungsklassen (viele Fallunterscheidungen entfallen)
  - Macht es einfach, neue Arten von Komponenten hinzuzufügen; Anwendungsklassen funktionieren ohne Änderung
  - Macht das Design vielleicht "allgemeiner" als gewünscht;
    falls nur bestimmte Klassen in ein bestimmtes Composite aufgenommen werden sollen, sind dafür Laufzeitüberprüfungen notwendig

© Prof. Dr. Peter Knauber
HS Mannheim



Partner-Diskussion: Das Composite-Pattern

- Diskutieren Sie mit einem Partner
  - Finden Sie/konstruieren Sie zwei Anwendungsfälle für das Composite-Pattern
  - Welche Operationen würde die Klasse *Component* deklarieren?
  - In welchen Situationen bringt das Pattern Vorteile?
  - Gibt es in Ihren Anwendungsfällen auch Situationen, in denen das Pattern Nachteile bringt?

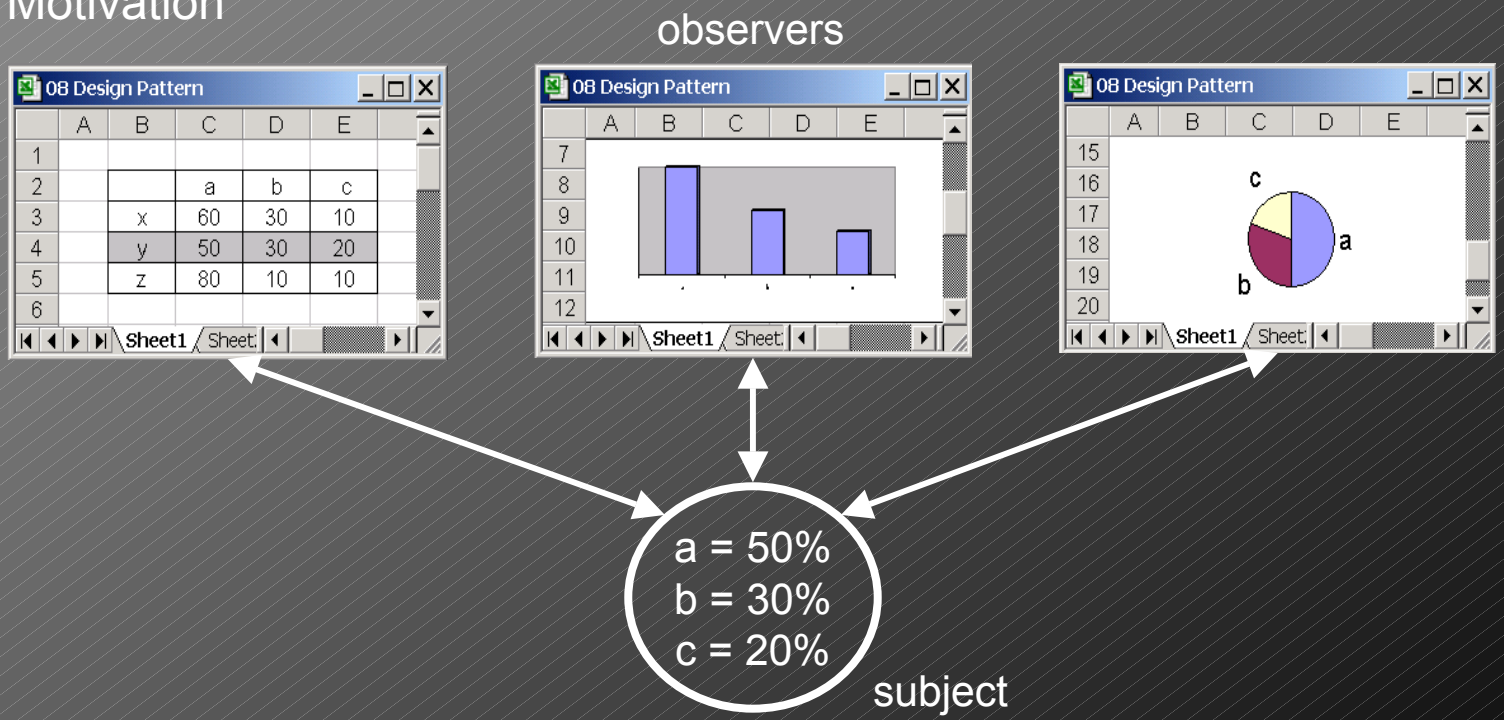
  Machen Sie sich Notizen!

- Dauer: 5 Minuten

© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein *behavioral* Pattern: Observer

- Aufgabe
  Definition einer 1 : n – Beziehung zwischen Objekten, so dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des Objekts ändert, von dem sie abhängen
- Motivation



© Prof. Dr. Peter Knauber
HS Mannheim

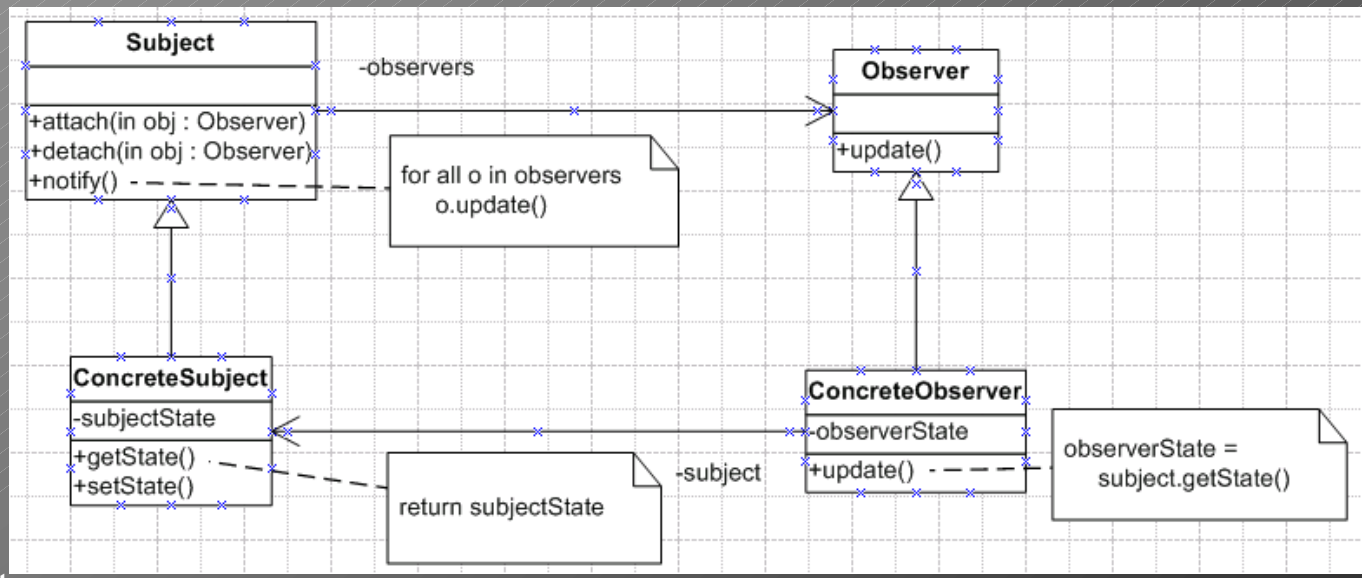
Beispiel für ein *behavioral* Pattern: Observer

- Motivation (Fortsetzung)
  - Daten und die verschiedenen Sichten darauf müssen konsistent gehalten werden
  - Ändern sich die Daten, so müssen die Tabelle und die Graphiken aktualisiert werden
- Anwendbarkeit
  - Wenn ein Modell zwei Aspekte hat, von denen der eine vom anderen abhängt; das Pattern ermöglicht es, diese Aspekte getrennt voneinander zu variieren und zu benutzen
  - Wenn eine Datenänderung eines Objekts Änderungen in anderen Objekten mit unbekannter Zahl nach sich zieht
  - Wenn ein Objekt andere Objekte benachrichtigen muss, ohne diese anderen Objekte zu kennen (geringe Kopplung)

© Prof. Dr. Peter Knauber
HS Mannheim

Beispiel für ein *behavioral* Pattern: Observer

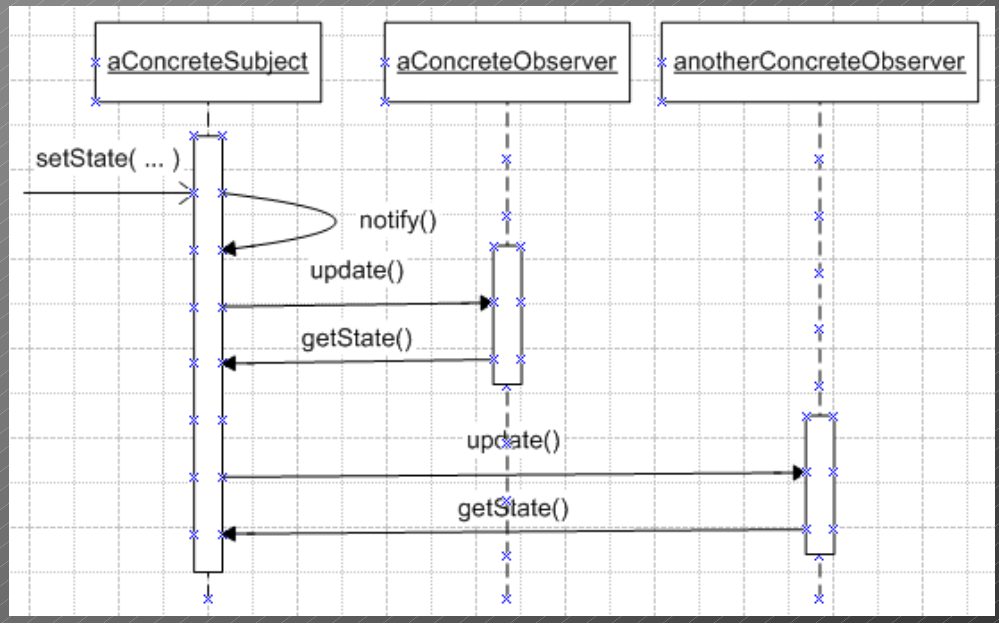
- Struktur



- Beteiligte
  - Subject
    - → Kennt seine Observer; eine beliebige Anzahl ist möglich
    - → Erlaubt es, Observer-Objekte anzuhängen (*attach*) und abzuhängen (*detach*)
  - Observer
    - → Definiert eine Schnittstelle für Objekte, die über Änderungen informiert werden sollen
  - ConcreteSubject
    - → Speichert den Zustand, der für *ConcreteObserver*-Objekte relevant ist
    - → Sendet eine Benachrichtigung (*notify*), wenn sich sein Zustand ändert
  - ConcreteObserver
    - → Enthält eine Referenz auf ein *ConcreteSubject*-Objekt
    - → Speichert den Zustandsanteil, der mit dem ConcreteSubject konsistent bleiben soll
    - → Implementiert die Update-Schnittstelle des *Observers*

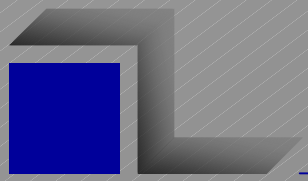
Beispiel für ein *behavioral* Pattern: Observer

- Zusammenarbeit



- Konsequenzen
  - Lose Kopplung zwischen *Subject* und *Observer*: Das *Subject* kennt nur die update-Schnittstelle zu seinen *Observern*
  - Broadcast-Kommunikation: Die update-Information wird an alle Objekte weitergeleitet, die sich zuvor registriert haben; es werden keine Objekte konkret angesprochen (jederzeit Hinzufügen/Wegnehmen möglich)
  - Update-Kaskaden:
    - → Änderungen im *Subject* können zu Kaskaden von *Observer*-Updates führen, kein (einzelnes) Objekt hat die Kontrolle
    - → Die update-Information ist unspezifisch

© Prof. Dr. Peter Knauber
HS Mannheim



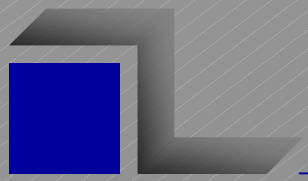
Partner-Diskussion: Das Observer-Pattern

- Diskutieren Sie mit einem Partner
  - Finden Sie/konstruieren Sie zwei Anwendungsfälle für das Observer-Pattern
  - Nennen Sie konkrete Subject-Daten und Observer dafür
  - In welchen Situationen bringt das Pattern Vorteile?
  - Gibt es in Ihren Anwendungsfällen auch Situationen, in denen das Pattern Nachteile bringt?

  Machen Sie sich Notizen!

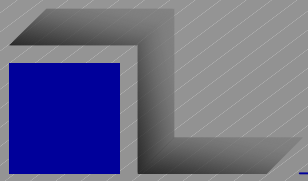
- Dauer: 5 Minuten

© Prof. Dr. Peter Knauber
HS Mannheim



Fragen?

© Prof. Dr. Peter Knauber
HS Mannheim



Aufgabe: Implementierung eines Design Pattern

- Implementieren Sie ein Design Pattern Ihrer Wahl (auch nicht vorgestellte Pattern sind erlaubt) in einem "sinnvollen Kontext"
- Beschreiben Sie genau, welchen *konkreten Nutzen* das Pattern in diesem Kontext bietet
- Dokumentieren Sie, warum Sie gerade dieses Pattern gewählt haben

Listing- und Doku-Abgabe: Dienstag, 20.12.05

© Prof. Dr. Peter Knauber
HS Mannheim