Design in Patterns in C#

Zitat:
„Jedes Muster beschreibt in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen“
A Pattern Language von Christopher Alexander, Oxford University Press 1977.

Agenda

1) Ziel DesignPatterns
2) Warum Design Patterns?
3) Warum sollte man sich mit D.S. beschäftigen?
4) Einführung Pattern anhand Singleton
5) Vorstellung Facade Pattern anhand Beispiel
6) Vorstellung Adapter Pattern anhand Beispiel
7) Vergleich Facade und Adapter
8) Vorstellung Bridge Pattern anhand Beispiel
9) Vorstellung Abstract Factory anhand Beispiel

1 Ziel von Design Patterns

- Vermitteln von Design Pattern und deren Absicht
- Abstraktion und Wiederverwendbarkeit von gutem OO-Design
- Festhalten von Design Pattern
  - Dokumentationsmittel
  - Erklären Entwurfsentscheidungen
- Bennenung und Katelogisierung von häufig auftretenden Entwurfsstrukturen
  - Gemeinsames Vokabular und Kommunikationsmedium
- Ergänzung zu bestehenden Designmethoden (Kein Ersatz)

Warum Design Patterns?

- Objektorientierte Programme bestehen aus Objekten
  - Objekte fassen Prozeduren und Daten zusammen
  - Objekte führen Operationen aus nach Anfrage vom Client
  - Schnittstelle bestimmt vollständige Anfragen an Objekte
- Verbindung zwischen Anfrage an ein Objekt und seiner Operation = dynamisches Binden
- Dynamisches Binden ermöglicht Objekte mit identischen Schnittstellen zur Laufzeit zu ersetzen.
- Polymorphie
  - Vereinfacht Definition von Clients
  - Entkoppelt Objekte voneinander
  - Variation von Objektbeziehungen

Schlussfolgerung

- **Design Patterns** helfen Schnittstellen zu definieren durch
  - Identifizierung der Arten von Daten und zentralen Elemente
- **Design Patterns** sagen was nicht in eine Schnittstelle verpackt werden darf
- **Design Patterns** definiern die Beziehungen der Schnittstelle

Wiederholung

- Klassen beschreiben, wie Objekte definiert werden
  - Klasse definiert internen Zustand vom Objekt und Implementierung seiner Operationen
- Klassenvererbung definiert Implementierung eines Objektes mit Hilfe Implementierung eines anderen Objektes
- Schnittstellenvererbung wann ein Objekt anstselle eines anderen verwendet werden.

Wiederholung

- Abstrakte Klassen definieren das Verhalten einer Menge von Klassen
  - Abstrakte Klassen ermöglichen eine Familie von Objekten mit identischen Schnittstellen zu definieren.
  - Alle konkreten Klassen teilen Schnittstelle der abstrakten Klassen.

Wiederholung

- Vorteile, wenn Objekte ausschließlich über Schnittstellen der abstrakten Klassen manipuliert werden:
  1. Klienten wissen nichts über die Klassen der Objekte, die sie benutzen, solange sie die Objekte über Schnittstelle erhalten
  2. Klienten kennen nur die Schnittstellen
- Reduzierung Implementierungsabh. zwischenSubsystemen
- *Programmiere immer auf die Schnittstelle nicht gegen die Implementierung*

Design Patterns vs. Framework

- Design Patterns sind abstrakter als Framework
  - Framework können als Code dargestellt werden
  - Design Patterns erläutern Vor-und Nachteile und Konsequenzen eines Entwurfs
- Design Patterns sind kleiner als Framework.
  - Framework enthält mehrere Design Patterns
- Design Patterns sind weniger spezialisierter als Framework. Framework haben immer bestimmten Anwendungsbereich.

Definition Design Patterns

- **Design Patterns**
  - Design Patterns sind Lösungsideen zu immer wiederkehrenden Entwurfsproblemen
  - Beschreibung zusammenarbeitender Objekte und Klassen
  - Beschreiben Lösungen für Probleme im Feinentwurf
- **Unterschiedliche Arten von** Design Patterns
  - Erzeugungsmuster (Abstraktion der Erzeugung)
  - Strukturmuster (Komposition von Klassen und Objekten, für größere Strukturbildung)
  - Verhaltensmuster (befassen sich mit Algorithmen und der Zuweisung von Zuständigkeiten zu Objekten und der Interaktion von Muster)
- **Gemeinsamkeiten**
  - **Abstrahieren von konkreten Objekten**
  - **Objektifizieren abstrakter Konzepte**
  - **Flexible Objektkomposition, -Konfiguration**

Warum soll man sich Design Patterns beschäftigen?

- **Design Patterns** helfen bei der Kommunikation und Wiederverwendung
- **Design Patterns** bieten eine abstrakte Sicht auf Analyse und Design
- **Verbessern Kommunikation im Team und das individuelle Lernen**
- **Code kann leichter modifiziert werden**
- **Design Pattenrs verdeutlichen objektorientierte Prinzipien**
- **Alternative zu umfangreichen Vererbungshierarchien**

Façade Patterns

- Vorstellung des Façade-Patterns
  - Zweck: eine einheitliche, abstrakte Schnittstelle
- Laut GoF ist das Anwendungsgebiet wie folgt:
  - Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstelle eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.

Facade Muster

- Facade soll den einfacheren Umgang mit komplexen Systemen ermöglichen:
  - Indem man Untermenge eines Gesamtsystems nutzt
  - Oder System auf eine bestimmte Art und Weise nutzt
  - Nur ein Teil des komplexen Systems wird genutzt

Façade-Pattern

- Wesentliche Merkmale:
  - **Zweck**: Man will die Handhabung eines existierenden Systems vereinfachen durch definition eigener Schnittstellen
  - **Problem**: man benötigt nur einen Teil des komplexen Systems. Oder man muss nur auf eine bestimmte Art mit dem System interagieren
  - **Lösung**: Die Fassade bietet einem Client eine spezialisierte Schnittstelle zu einem bestehenden System
  - **Teilnehmer:** Ein Client bekommt eine spezialisierte Schnittstelle mit einfacher Handhabung

Façade-Pattern

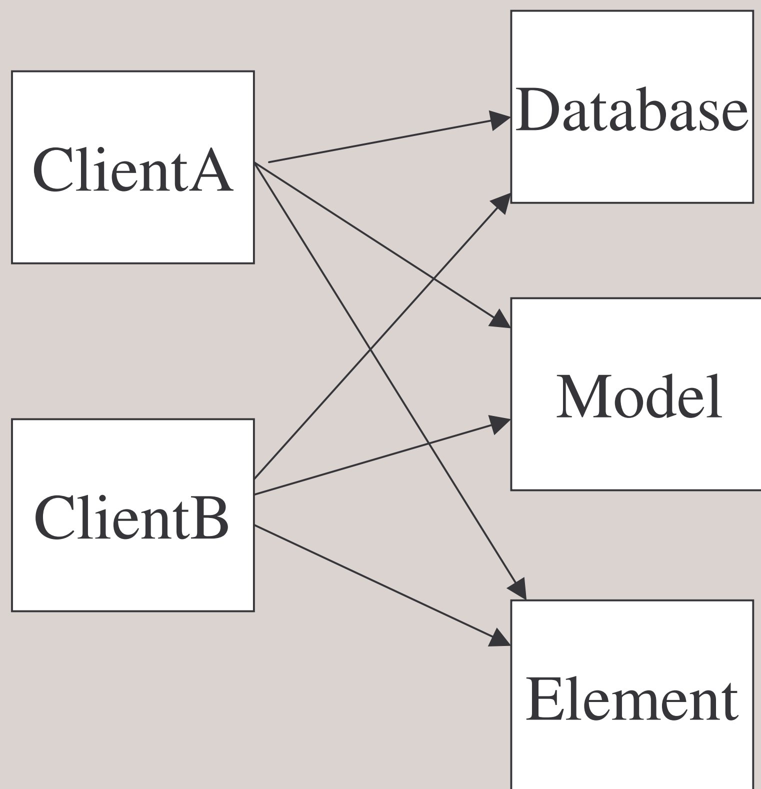
  - **Konsequenzen**: Facade vereinfacht Verwendung eines bestimmten Subsystems.
- **Implementierung:**
  - Definition einer neuen Klasse(n) mit der gewünschten Schnittstelle.
  - Neue Klasse verwendet bestehende System zur Erfüllung der Aufgaben
  - GoF-Referenzen: Seiten 185-193

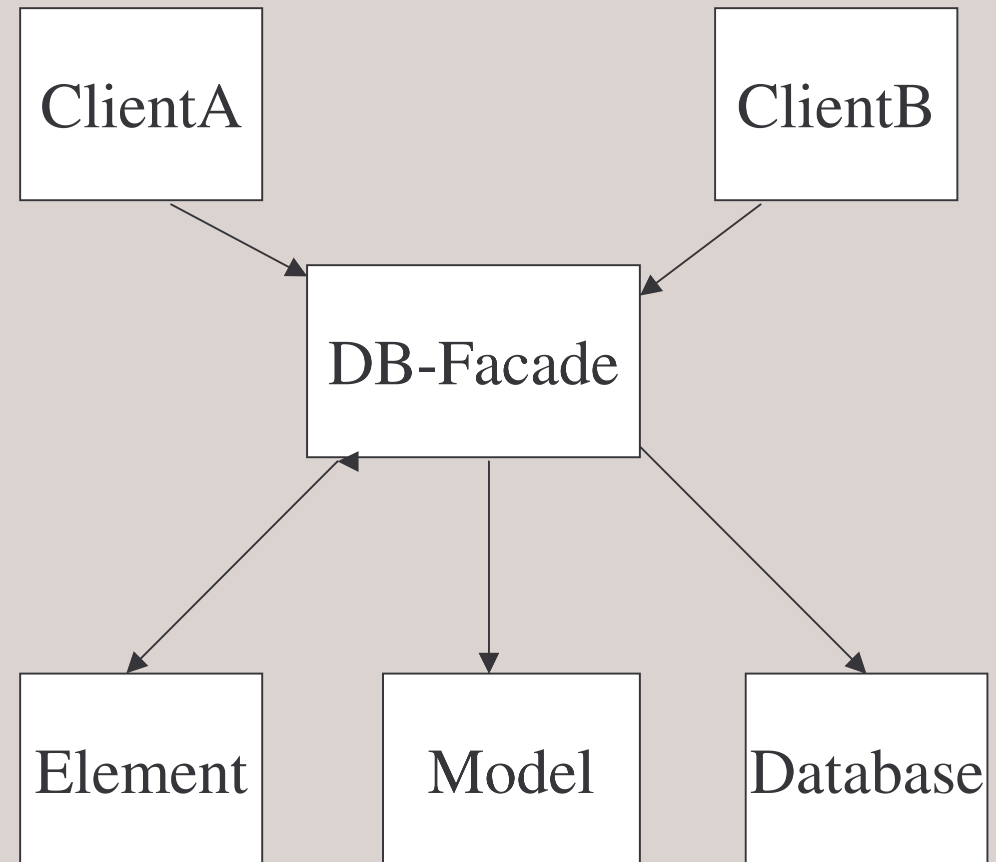
Façade-Pattern

- Variationen der Facade:
  - Ergänzung bestehender Funktionen durch neuen Routinen
- Pattern bieten einen allgemeingültigen Ansatz
  - Ein Pattern bietet nur die Richtung für eine Vorgehensweise an. Ob man neue Methoden hinzufügen muss oder nicht, hängt von der gegebenen Situationen ab. Pattern bieten so etwas wie einen Entwurf an, der den Einstieg erleichtert. Es sind keine festen, in Stein gehauene Regeln.

Anmerkungen zum Facade-Muster

- Variationen der Fassade: Verminderung der Anzahl der Objekte, mit denen ein Client arbeiten muss.





Anmerkungen zum Pattern

- Variationen der Facade: Eine einkapselnde Schicht:
  - Eine Fassade kann dazu dienen ein System zu verbergen, ein System zu kapseln oder zu verbergen, z.B. System kann als private Attribut der Fassade-Klasse enthalten sein.
  - Somit System als fester Bestandteil der Façade-Klasse.
- Vorteile durch Systemkapselung:
  - Systemanwendung überwachen
  - Systeme austauschen

Singleton

- Zweck gemäß GoF: Man soll sicherstellen, dass von einer Klasse nur eine Instanz existiert und das Muster bietet einen allgemeinen Zugangspunkt zu dieser Instanz

Wie funktioniert das Singleton-Muster?

- Wird Methode aufgerufen, prüft das Singleton, ob Objekt bereits instanziert ist.
- Ist dies der Fall, so gibt die Methode eine Referenz auf das existierende Objekt zurück.
- Wenn nicht, wird es neu erzeugt und eine neue Referenz zurückgegeben.
- Wichtig: Konstruktor des Singleton wird als private oder protected definiert

Wesentliche Merkmale

- Zweck: Man möchte nur eine Instanz des Objekts, es gibt jedoch kein globales Objekt, dass Instanzierung zentral steuert
- Problem: Verschiedene Client-Objekte müssen auf ein gemeinsam genutztes Objekt zugreifen.
- Lösung: garantiert genau eine Instanz
- Teilnehmer: Client-Objekte erzeugen ausschließlich durch Methode getInstance() eine Instanz der Singleton-Klasse

Wesentliche Merkmale

- Konsequenzen: Client-Objekte müssen sich nicht darum kümmern, ob bereits Instanz des Singleton existiert, Kontrolle durch das Singleton-Objekts selbst
- Implementierung:
  - Füge ein private oder static Attribut von Typ der Klasse hinzu, welches auf gewünschte Objekt verweist
  - Füge eine public static Methode hinzu, die Objekt der Klasse instanziert, falls obige Attribut NULL hat. Methode setzt Wert der Variable auf Referenz des neu erzeugten Objekts und liefert Wert dieses Attributes als Rückgabewert

Wesentliche Merkmale

- Implementierung: Konstruktor sollte protected oder private markiert werden, so dass niemand Instanziierung durch einen statischen Konstruktor umgehen kann.

Singleton
-theInstance : Singleton
+getSingleton() : Singleton

Adapter-Muster

- Zweck: Erzeugen neuer Schnittstellen
- Laut GoF:
  - Passe die Schnittstellen einer Klasse an eine andere von Ihren Client erwartete Schnittstelle an. Das Adapter-Muster lässt Klassen zusammen arbeiten, die wegen inkompatiblen Schnittstellen ansonsten, nicht in der Lage wären

  Kurz gesagt: Man benötigt eine neue Schnittstelle für ein Objekt zu erzeugen, das zwar das gewünschte leisten kann, aber über eine falsche Schnittstelle verfügt

Wesentliche Merkmale

- Zweck: : Man passt eine existierende Objekt, das außerhalb des eigenen Einflussbereiches liegt, an eine andere Schnittstelle an.
- Problem: **Ein System hat die richtigen Daten und das richtige Verhalten, jedoch die falsche Schnittstelle. Oft verwendet, wenn man von einer bestehenden oder zu definierenden abstrakten Klasse ableiten muss.**
- Lösung: Ein Adapter umwickelt(Wrapper) eine Klasse oder ein Objekt mit der gewünschten Schnittstelle.

Wesentliche Merkmale

- Teilnehmer: Der Adapter passt die Schnittstelle des adaptierenden Objekts so an, dass es der Schnittstelle des Zielobjekts Target des Adapters entspricht (der Klasse von der geerbt wird). So kann ein Client das adaptierte Objekt so verwenden, als wäre es vom Typ des üblichen Zielobjekts(Target).
- Konsequenzen: Das Adapter-Muster erlaubt es, bestehende Klassen in neue Klassenhierarchien einzufügen, ohne das deren unpassende Schnittstelle sich störend auswirkt.

Wesentliche Merkmale

- Implementierung:
- Die bestehende Klasse wird in eine andere Klasse eingebunden. Die umschließend Klasse muss eine passende Schnittstelle definieren und die Methoden der beinhalteten Klasse aufrufen.
- GoF-Referenz: Seiten 139-150

Variationen des Adapter Muster

- Das Objektadapter-Muster- Das Adapter-Muster, so wie ich es bisher verwendet habe, wird als Objektadapter bezeichnet, da dessen Anwendung sich auf ein adaptierende Objekt bezieht, welches ein adaptierte Objekt enthält.
- Klassenadapter-Muster: Eine andere Möglichkeit, das Adapter-Muster zu realisieren, besteht in der Awendung von Mehrfachvererbung. (Klassenadapter-Muster) bezeichnet.
- Je nach Problemstellung kann eins der o.g. Adapter verwendet werden.

Vergleich zwischen Fassade und Adapter

- In beiden Fällen gibt es eine bereits existierende Klassen oder Klassen, die keine passende Schnittstelle hat. In beiden Fällen wird ein neues Objekt erzeugt, das die gewünschte Schnittstelle aufweist.

Vergleich

- Beide Muster umhüllen (Wrapper)
- Populär vor allem bei der Umhüllung von Legacy Systemen mit Hilfe von Objekten
- Aber beide Muster beinhalten unterschiedliche Wrapper
- Unterschiede:

	Fassade	Adapter
Gibt es bereits existierende Klassen?	Ja	Ja
Gibt es Vorgaben für eine Schnittstelle?	Nein	Ja
Muss Objekt polymorph verwendet werden können?	Nein	Möglich
Ist eine vereinfachte Schnittstelle notwendig?	Ja	Nein

Fazit: Eine Fassade vereinfacht eine Schnittstelle, während ein Adapter bereits bestehende Schnittstelle in eine gewünschte konvertiert.

Brücken-Muster

- Zweck: Entkopplung der Abstraktion von der Implementierung

Laut GoF dient das Brücken-Muster dazu, die Abstraktion von Ihrer Implementierung zu entkoppeln, so dass beide unabhängig voneinander variiert werden können.

Wie bitte?

Brücken-Muster

- Entkoppeln bedeutet, dass sich Dinge unabhängig voneinander verhalten oder explizit angeben, welche Art von Beziehung zwischen Objekten besteht.
- Abstraktion zeigt an, wie Dinge konzeptionell miteinander in Beziehung stehen
- Brücken-Muster ist ein sehr gutes Beispiel für Umsetzung folgender Kernaussagen:
  - Finde Unterschiede und kapsele sie ein.
  - Komposition von Objekten ist der Vererbung vorzuziehen

Verstehen Brücken-Muster

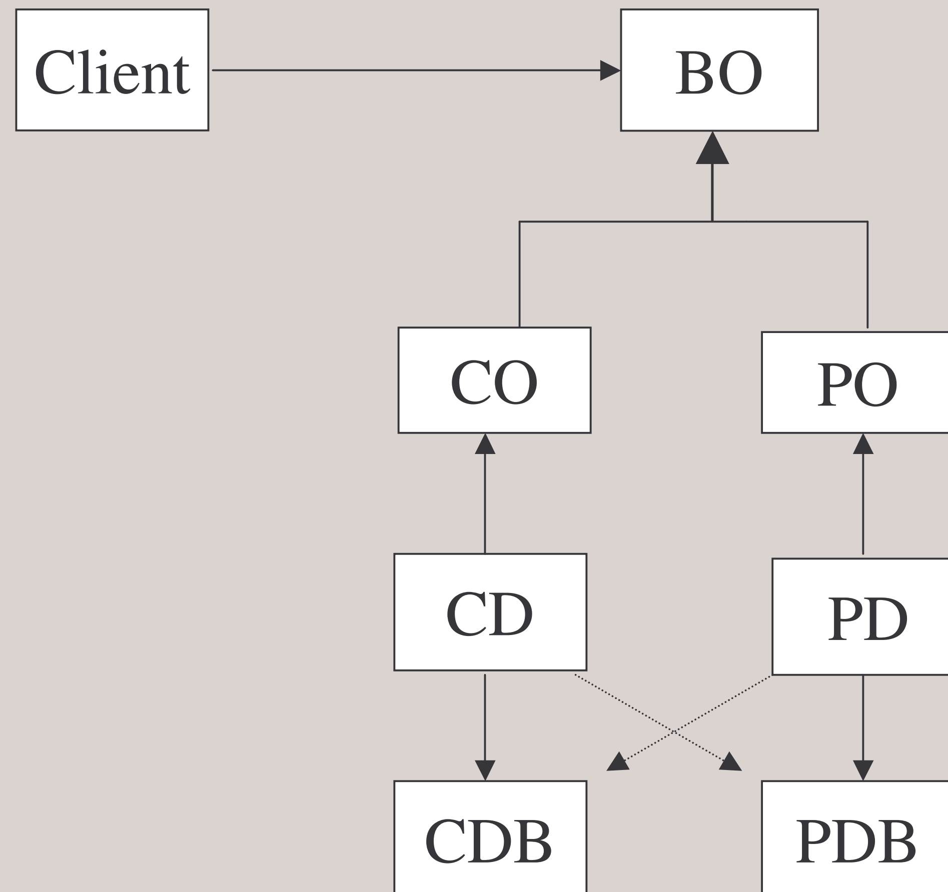
- Brücken Muster anhand eines Beispiels verstehen
  - Verstehen, warum es das Muster gibt, und es dann herleiten

Anforderung:

Erstellen einer OLAP-Anwendung, die BusinessObjekte wie Kunden von Datenbanken liefert

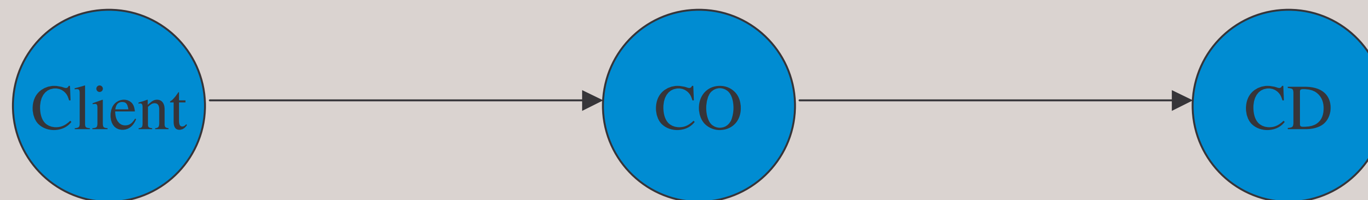
Praxisbeispiel

1. Anwendung Vererbung



Brücken-Muster

- Design durch Vererbung



- Diese Lösung leidet unter kombinatorischer Explosion
  – Hinzufügung einer neuen DB, d.h. eine weitere Variation der Implementierung.
  – Hinzufügung einer neuen Variation im Konzept z.B. TimeObject
  – Anzahl der Klassen kann schnell explodieren, aufgrund enger Kopplung zwischen Abstraktion und Implementierung

Beobachtung zur Verwendung Design Pattern

- Eine neue Sicht auf Design Pattern
  - Zunächst Konzentration auf Lösung.
  - Design Pattern werben damit, dass sie gute Lösung für vorliegende Probleme bieten
- Damit „zäumt man das Pferd von der falschen Seite auf".
  - Schwer zu identifizieren, wann ein Muster zu einer Problemstellung passt.
  - Man erfährt nur was zu tun ist, nicht aber, wann es angemessen ist das Muster zu nutzen
- Lösung: Konzentration auf den Kontext eines Design Pattern, das Problem, welches es zu lösen versucht

Bridge Pattern verstehen

- Was für ein Problem soll das Bridge Pattern lösen:
  - Das Bridge Pattern ist sinnvoll einsetzbar, wenn man eine Abstraktion mit verschiedenen Implementierungen hat. Es ermöglicht es der Abstraktion und der Implementierung unabhängig voneinander zu variieren.

Herleiten Brückenmuster

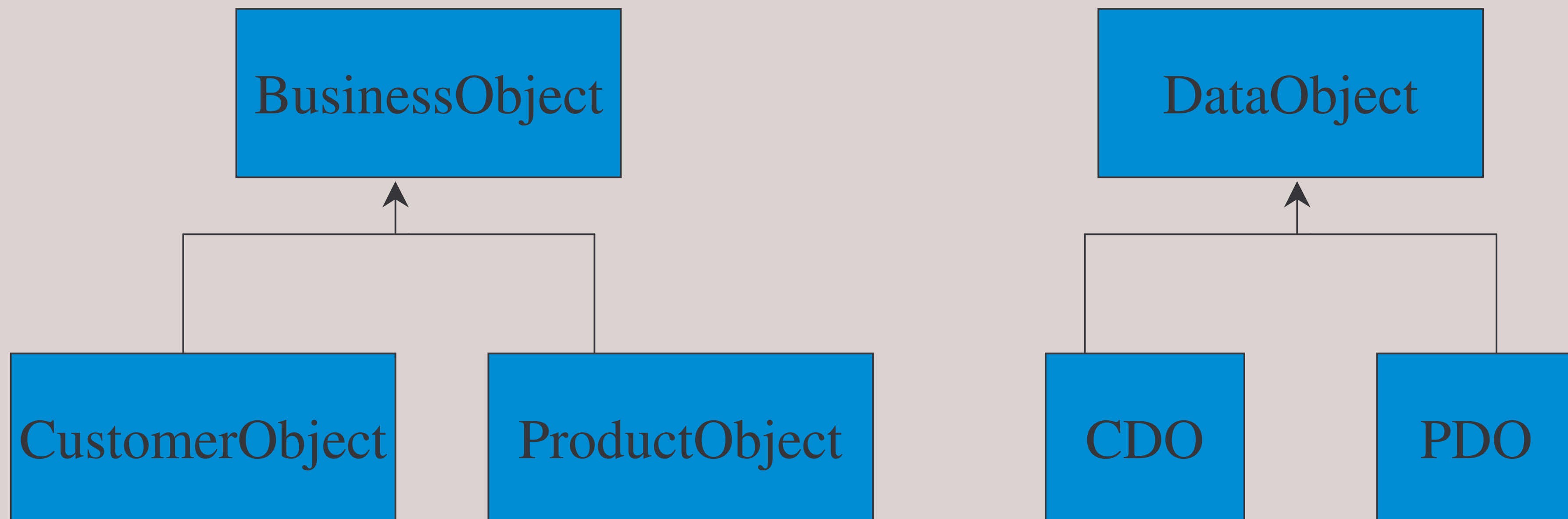
- Identifizierung der Unterschiede
  1. Kapselung der Unterschiede
  2. Komposition der Vererbung vorziehen

BusinessObject

DataObject

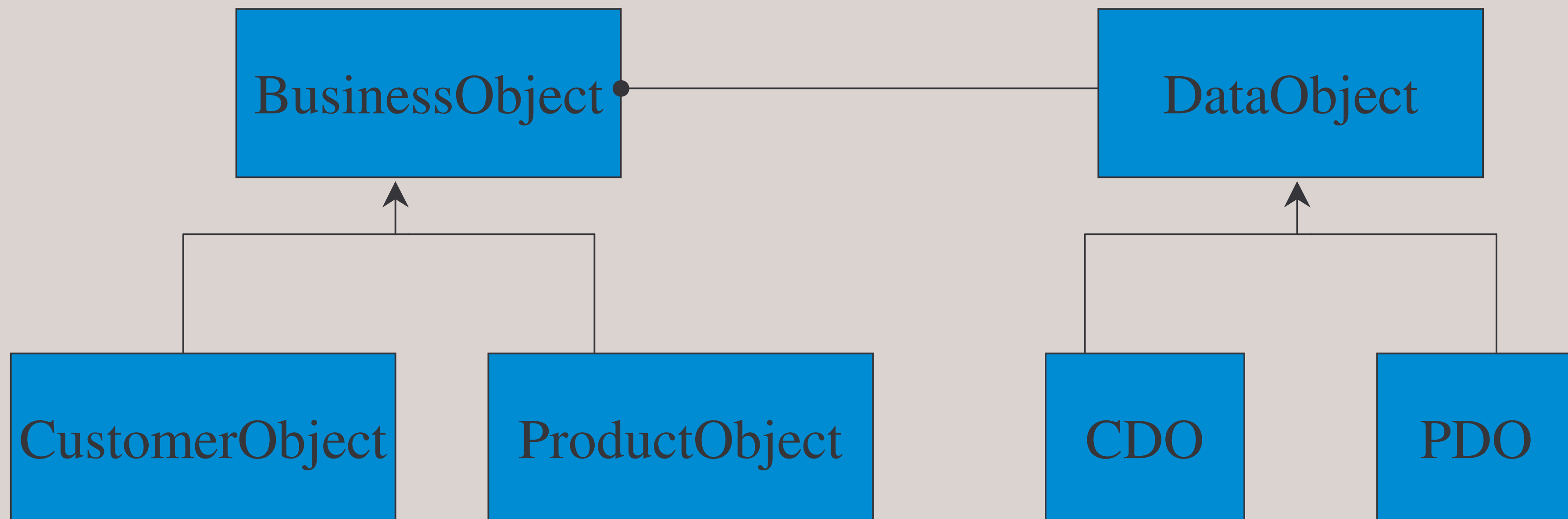
Brückenmuster Herleiten

- Unterschiede darstellen



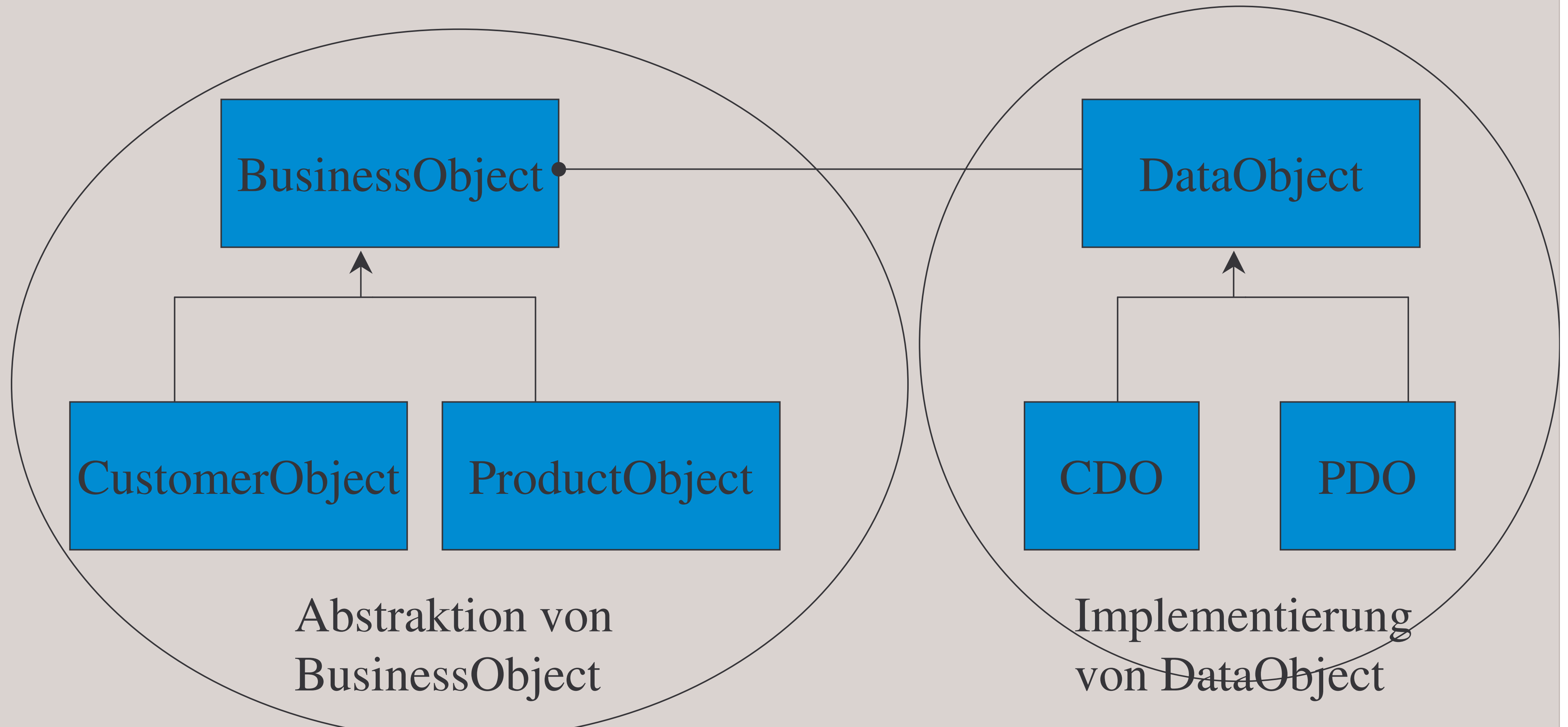
Brückenmuster Herleiten

- Die Klassen verbinden: Wer benutzt wen?



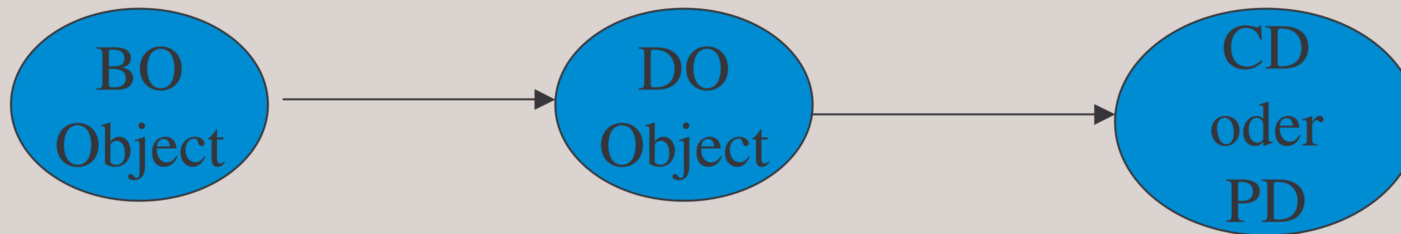
Bridge Pattern Herleiten

- Den Entwurf erweitern



Bridge Pattern herleiten

- Aus der Sicht von Objekten



Entweder CO
oder PO,
Client kann
Unterschied
nicht festhalten

Instanz von Customer Data
oder Product Data,
BO Object kann nicht
unterscheiden
da beide identisch sind.

- Fazit: Es sind nur drei Objekte zur selben Zeit beteiligt

Bridge Pattern im Rückblick

- Wichtig: Die Schnittstelle der Implementation sollte so gestaltet werden, dass sie die Erfordernisse der von der abstrakten Klasse abgeleiteten Klasse erfüllt.
- Das Bridge-Pattern beinhaltet oft das Adapter-Muster

Merkmale

- Zweck: Entkopplung von Implementierungen und den Objekten, die diese Implementierungen verwenden.
- Problem: Die von einer abstrakten Klasse abgeleiteten Klassen müssen verschiedene Implementierungen nutzen können, ohne dass diese zu einer sehr schnell wachsenden Anzahl von Klassen führt.
- Lösung: Definition einer Schnittstelle, die von allen Implementierungen verwenden werden muss. Die Ableitungen der abstrakten Klasse verwenden dann diese Schnittstelle.

Merkmale

- Teilnehmer: Die Abstraktion definiert die Schnittstelle für die zu implementierenden Objekten. Der Implementator definiert die Schnittstellen für die konkreten Implementierungsklassen. Von der abstraktion abgeleitete Klassen nutzen Klassen, die vom Implementator abgeleitet sind, ohne zu wissen, welcher KonkreteImplementator verwendet wird.

Merkmale

- Konsequenzen: Die Entkopplung der Implementierung von den Klassen, die diese Implementierung nutzen, macht eine Lösung erweiterbar. Client-Objekte wissen nichts über die Details der Implementierung.
- Implementierung: Die Implementierung wird durch eine abstrakte Klasse gekapselt.
- In der Basisklasse der Abstraktion, die implementiert wird, muss eine Referenz auf die Implementierung enthalten sein.
- GoF-Referenz: S. 151-162

Abstract Factory Pattern

- Zweck: koordinierte Instanziierung von Objekten
- Nach GoF: Das Abstract Factory bietet eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.
- Abstract Factory sorgt dafür, dass das System immer die für die Situation passende Objekt erhält

Verstehen
Abstract Factory

- Beispiel.: Auswahl eines Gerätetreibers in Abhängigkeit der Systemkapazität

Vier Treiber	Schwächeren Systemen	Leistungsstarken Systemen
Bildschirm	LRDD	HRDD
Drucker	LRPD	HRPD

Verstehen Abstract Factory

- Definition von Familien auf der Basis eines vereinigten Konzepts
  - Eine Familie für geringe Auflösungen –LRDD und LRPD
  - Eine Familie für hohe Auflösungen HRDD und HRPD

Verstehen Abstract Factory

- Alternative 1: Verwendung einer switch-Anweisung zur Auswahl des Treibers

```
class AppControl
{
void doDraw()
{
switch (RESOLUTION){
case LOW:
//verwende lrdd
case HIGH:
//verwende hrdd
}
void doPrint(){…}
}
```

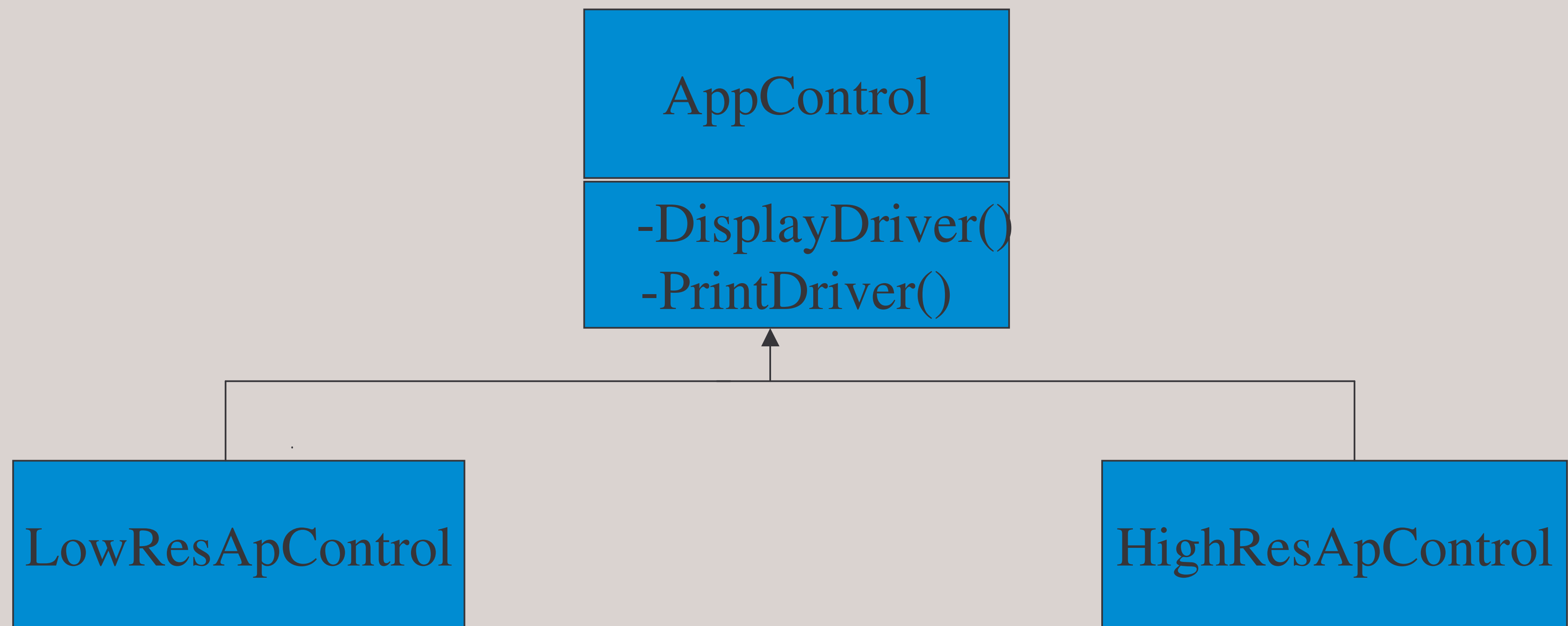
Verstehen Abstract Factory

- Problem enge Kopplung bei Alternative 1
- Geringe Kohäsion

Momentan stellen beide o.g. Nachteile kein großes Problem. Nur später kann dies zu hohen Wartungskosten führen.

Verstehen Abstract Factory

- Alternative 2: Anwendung Vererbung



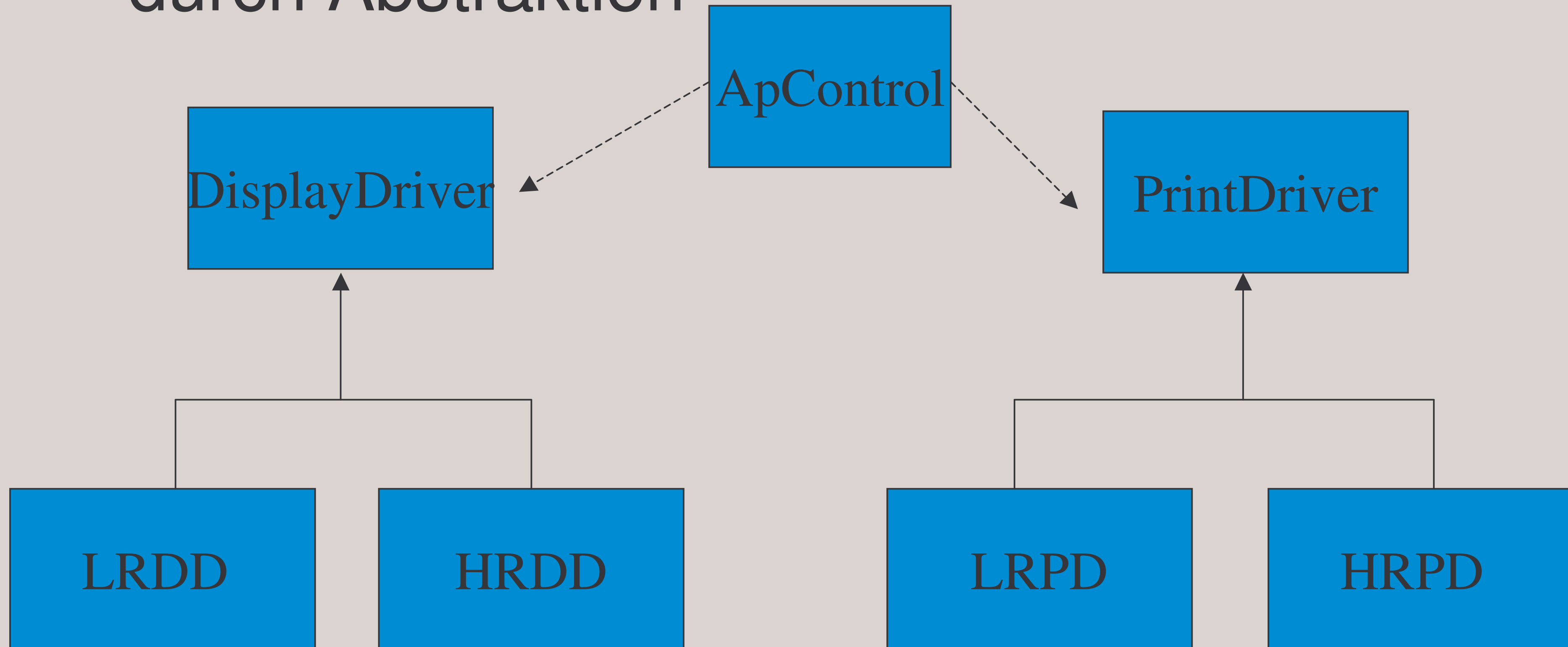
Konstruktor instanziiert Treiber für entsprechende Auflösung und weist diese an Display-Driver und PrintDriver zu.

Verstehen Abstract Factory

- Kombinatorische Explosion: Für jede der verschiedene Familien und jede neue zukünftige Familie muss eine neue, konkrete Klasse erzeugt werden. (also neue Version von ApControl)
- Unklare Bedeutung: Erzeugte Klassen helfen nicht klarzustellen, was eigentlich passiert. Jede Klasse wurde an ein Spezialfall angepasst.
- Verletzung der Regel Komposition der Vererbung vorzuziehen

Fabrik-Muster verstehen

- Alternative 3: Ersetzen der switch-Anweisung durch Abstraktion



Verstehen Abstract Factory

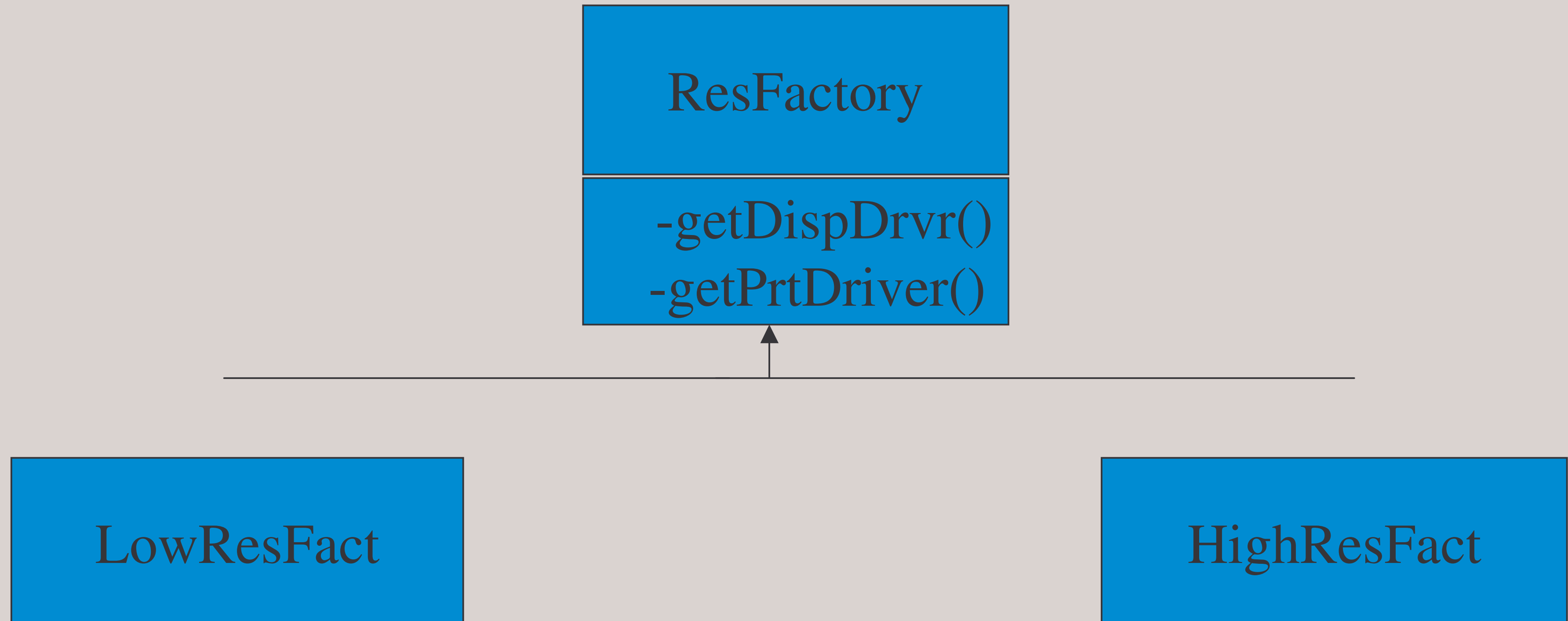
- Klasse ApControl ist leichter zu verstehen,
  - Da sie sich nicht drum kümmern muss, welche Art von Treiber zu verwenden ist.
- Fabrik-Objekte
  - Wie erzeugt man die richtigen Objekte?
  - Antwort: Nicht ApControl kümmert sich um diese Aufgabe, sondern ein Fabrik-Objekt.
  - Das Objekt ApControl wird ein andere Objekt verwenden, das Fabrik-Objekt, um für das aktuelle Rechnersystem die passenden Treiber für Bildschirm und Drucker zu erhalten.
  - ➢ Fabrik ist verantwortlich…

Verstehen Abstract Factory

- Fabrik-Objekt= ResFactory-Objekt.
  - ResFactory kümmert sich darum, die passenden Treiber zu erzeugen, AppControl tut nichts weiter als die Treiber zu nutzen
  - Variationen werden in einer Klasse gekapselt
- Für die Klasse ResFactory hätte man zwei Variationen im Verhalten (Methoden):
  - Liefere mir den Bildschirmtreiber, den ich verwenden kann
  - Liefere mir den Druckertreiber, den ich verwenden kann

Verstehen Abstract Factory

- ResFactory kann aus einer der beiden Klassen instanziiert und von einer abstrakten Klasse abgeleitet werden, die über öffentliche Methoden verfügen



Strategien für Brückenschlag zwischen Analyse und Design

- Identifiziere Unterschiede und kapsele sie ein.
  - Die Wahl, welche Treiber zu verwenden ist, viel auf verschiedene. Kapselung durch ResFactory
- Komposition ist der Vererbung vorzuziehen.
  - Verlagerung der Variation in ein anderes Objekt, ResFactory, das durch ApControl verwendet wird. Sonst zwei verschiedene ApControl
- Entwurf soll sich an Schnittstelle und nicht an Implementierung orientieren
  - ApControl weiß, wie es ein Objekt vom Typ ResFactory nach einem Treiber fragt. Es weiß nicht, wie ResFactory dies eigentlich tut.

Abstract Factory Verstehen

- Rollen der Objekte in der Abstract Factory:
  - Client-Objekt weiß nur ,wen es nach den benötigten Objekten fragen kann und wie es diese verwenden kann.
  - Abstract Factory legt fest, welche Objekte instanziiert werden können, indem sie für jede Art von Objekten eine Methode definiert.
  - Konkreten Fabrik-Objekte legen fest, welche Objekte tatsächlich erzeugt werden.

Merkmale Abstract-Factory

- Siehe GoF-Referenz S. 87-96
- Anmerkungen zur Abstract Factory:
  - Wie man das richtige Fabrik-Objekt erhält:
  - Config-Datei, die die notwendigen Infos enthält, um auf Grundlage diese Konfiginfos das richtige Factory-Objekt zu instanziieren.
  - Verwendung AbstractFactory, so dass ich Teilsysteme für verschiedene Anwendungen verwenden kann. In diesem Fall ist es üblicherweise dem Hauptteil des Systems bekannt, welche Familie von Objekten ein Subsystem benötigen wird.

Bücherliste

- Entwurfsmuster verstehen von Alan Shalloway, James R. Trott mitp Verlag ISBN:3-8266-1345-7
- Design Patterns GoF von Adison-Wesley Verlag