

Kapitel 7

Entwurfsmuster (“Design Patterns”)

Stand: 23.11.2010

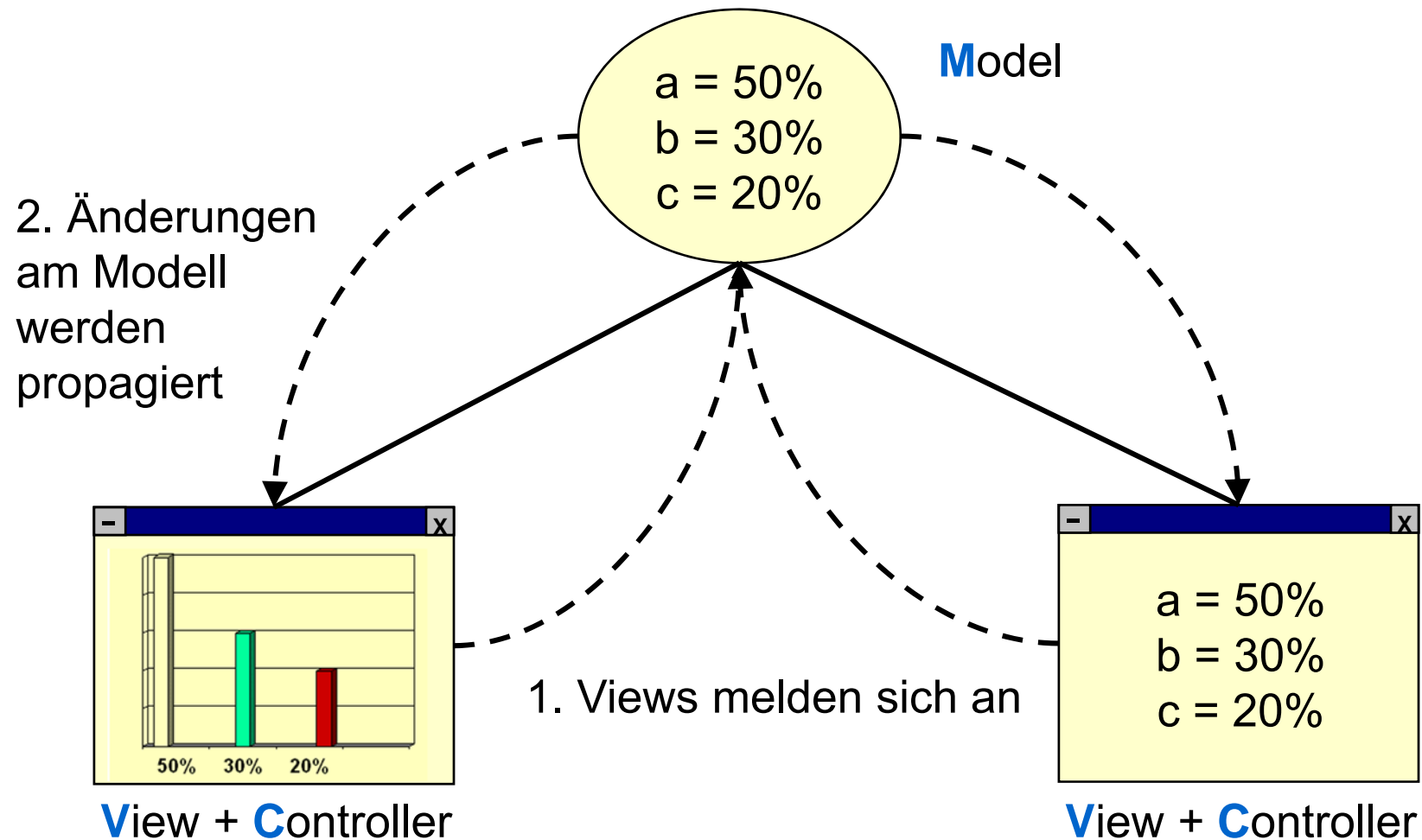
Einführung: Die Grundidee von Pattern

- Grundlegende Idee
 - ◆ Alle Ingenieurdisziplinen benötigen eine **Terminologie** ihrer wesentlichen Konzepte sowie eine **Sprache**, die diese Konzepte in Beziehung setzt.
 - ◆ Pattern (Muster) wurden zuerst im Bereich der Architektur beschrieben.
- Ziele von Pattern in der Welt der Software:
 - ◆ **Dokumentation** von Lösungen wiederkehrender Probleme, um Programmierer bei der Softwareentwicklung zu unterstützen.
 - ◆ Schaffung einer gemeinsamen **Sprache**, um über Probleme und ihre Lösungen zu sprechen.
 - ◆ Bereitstellung eines standardisierten **Katalogisierungsschemas** um erfolgreiche Lösungen aufzuzeichnen.
- Bei Pattern handelt es sich weniger um eine Technologie (wie z.B. bei UML), als um eine Kultur der Dokumentation und Unterstützung guter Softwarearchitektur.

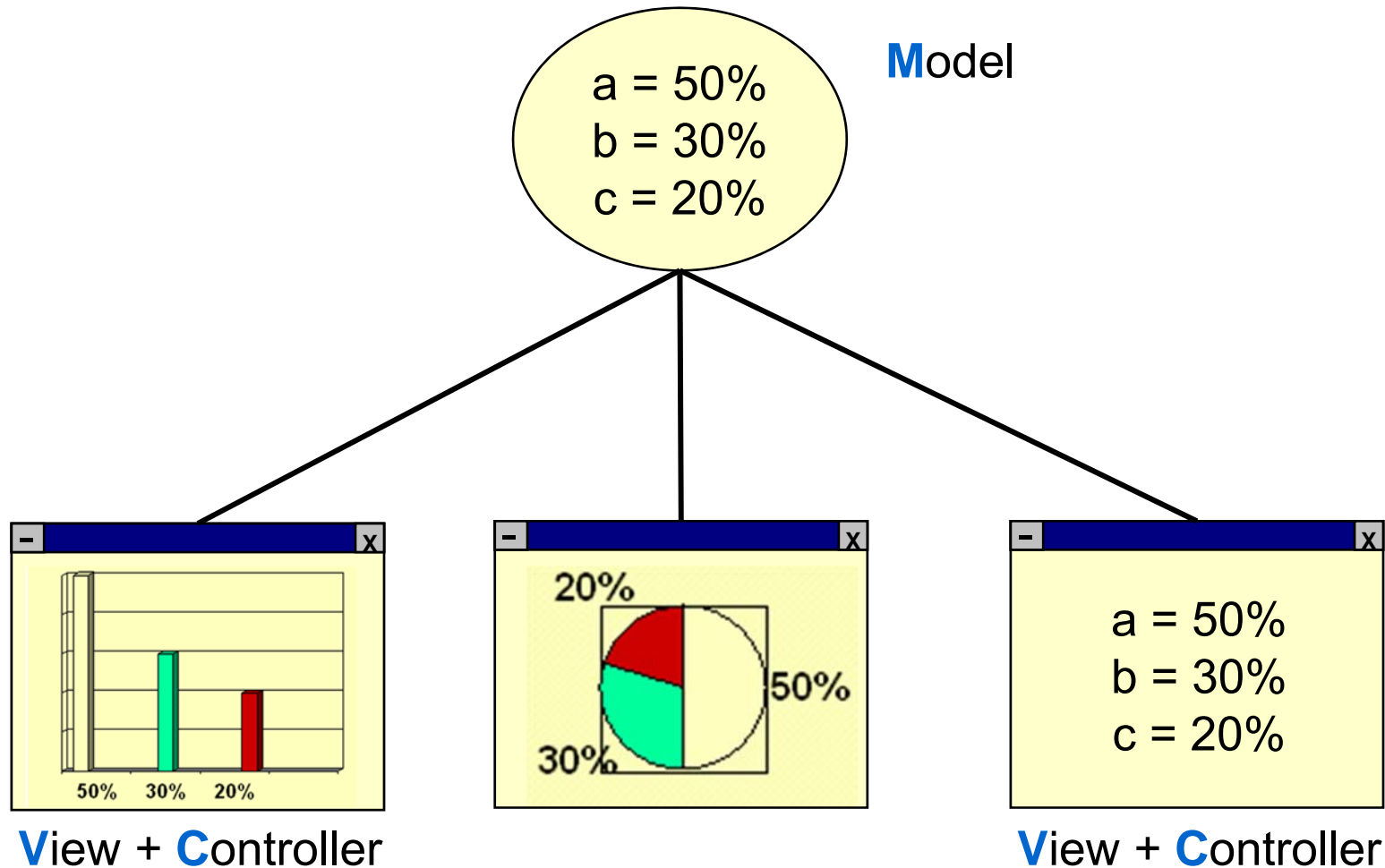
Einführung: Literatur zu Pattern und Software

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): "Design Patterns - Elements of Reusable Object-Oriented Software" Addison-Wesley, 1995
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal (Gang of Five): "Pattern-Oriented Software Architecture - A System of Patterns" John Wiley & Sons, 1996
- Serge Demeyer, Stephane Ducasse, Oscar Nierstrasz: "Object Oriented Reengineering Patterns", Morgan Kaufman, 2002
- Patterns im WWW
 - ◆ Portland Pattern Repository: <http://c2.com/ppr/>
 - ◆ Hillside Group Patterns Library: <http://www.hillside.net/patterns/>
 - ◆ Brad Appleton: „Patterns and Software: Essential Concepts and Terminology“ <http://www.bradapp.net/docs/patterns-intro.html>
 - ◆ Doug Lea, Patterns-Discussion FAQ, <http://gee.oswego.edu/dl/pd-FAQ/pd-FAQ.html>
 - ◆ Buchliste: <http://c2.com/cgi/wiki?PatternRelatedBookList>

Einführung: Das MVC-Framework in Smalltalk als Beispiel



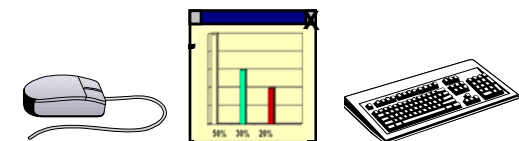
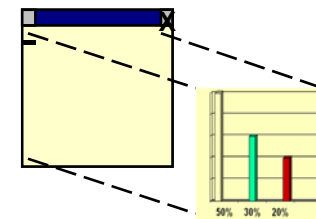
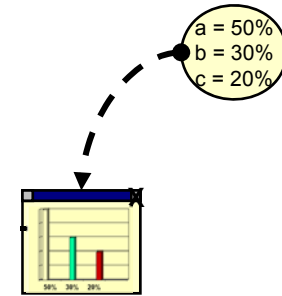
Einführung: Das MVC-Framework in Smalltalk als Beispiel



Neue Views können ohne Änderung des Modells oder der anderen Views hinzugefügt werden

Einführung: Das MVC-Framework in Smalltalk als Beispiel

- Propagierung von Änderungen: **Observer Pattern**
 - ◆ Kommt z.B. auch bei Client/Server-Programmierung zur Benachrichtigung der Clients zum Einsatz
- Geschachtelte Views: **Composite Pattern**
 - ◆ View enthält weitere Views, wird aber wie ein einziger View behandelt.
 - ◆ Kommt z.B. auch bei Parsebäumen im Compilerbau zum Einsatz (Ausdrücke).
- Reaktion auf Events im Controller: **Strategy Pattern**
 - ◆ Eingabedaten können validiert werden (Daten, Zahlen, etc.).
 - ◆ Controller können zur Laufzeit gewechselt werden.
 - ◆ Kommt z.B. auch bei der Codeerzeugung im Compilerbau zum Einsatz (Code für verschiedene CPUs).



1. Einführung
2. Einstiegsbeispiel: "Observer" im Detail
3. Was also sind Patterns?
4. Wichtige Patterns
5. Zusammenspiel verschiedener Patterns
6. Fazit: Nutzen von Patterns
7. Zusammenfassung und Ausblick

Das Observer Pattern

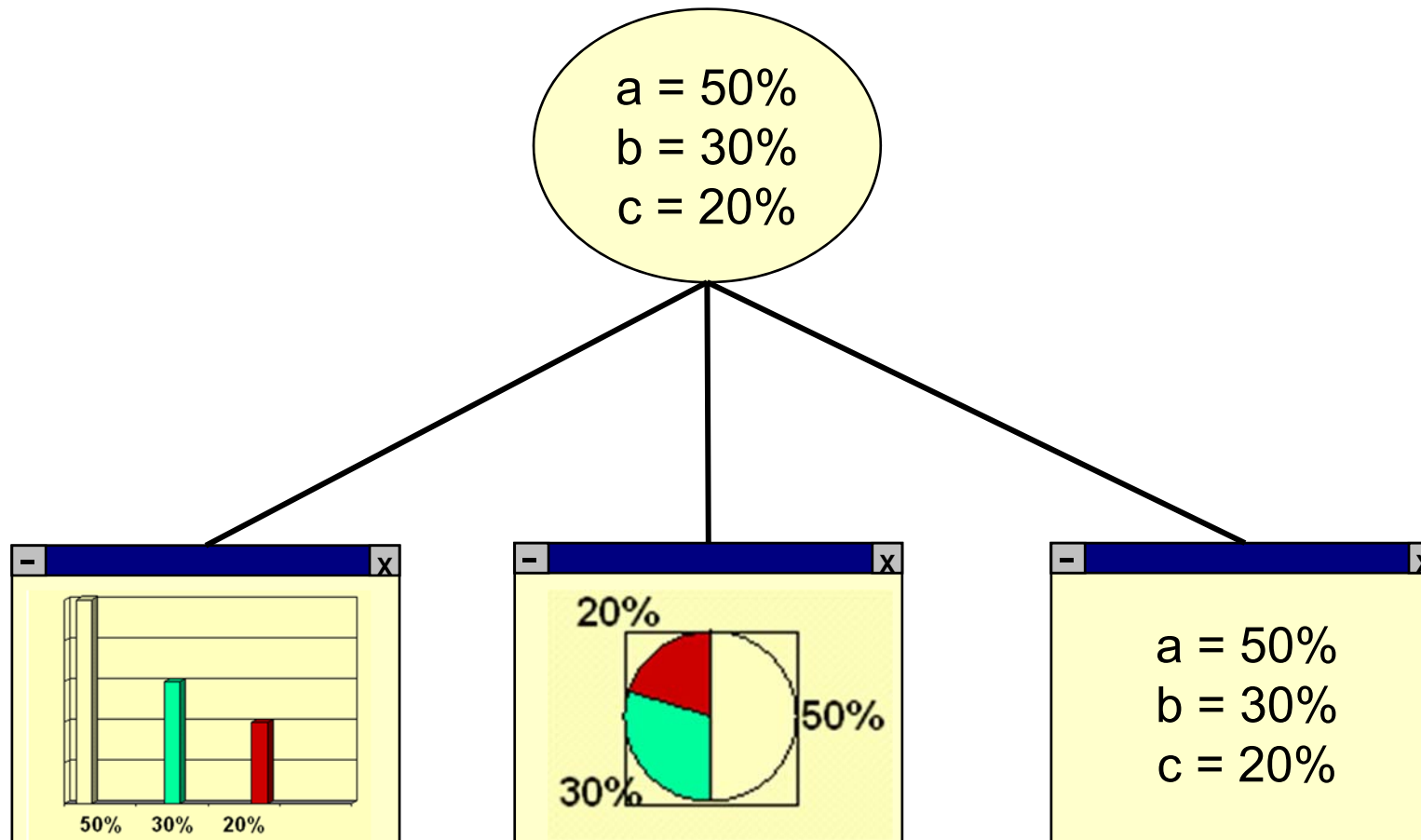
Das Observer Pattern: Einführung

- Absicht
 - ◆ Stellt eine 1-zu-n Beziehung zwischen Objekten her
 - ◆ Wenn das eine Objekt seinen Zustand ändert, werden die davon abhängigen Objekte benachrichtigt und entsprechend aktualisiert
- Andere Namen
 - ◆ "Dependents", "Publish-Subscribe", "Listener"
- Motivation
 - ◆ Verschiedene Objekte sollen zueinander konsistent gehalten werden
 - ◆ Andererseits sollen sie dennoch nicht eng miteinander gekoppelt sein. (bessere Wiederverwendbarkeit)

 - ◆ Diese Ziele stehen in einem gewissen Konflikt zueinander. Man spricht von „*conflicting forces*“, also gegenläufig wirkenden Kräften.

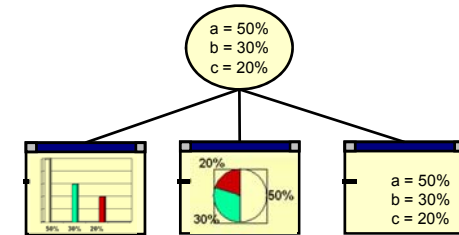
Das Observer Pattern: Beispiel

- Trennung von Daten und Darstellung
 - ◆ Wenn in einer Sicht Änderungen vorgenommen werden, werden alle anderen Sichten aktualisiert – Sichten sind aber unabhängig voneinander.



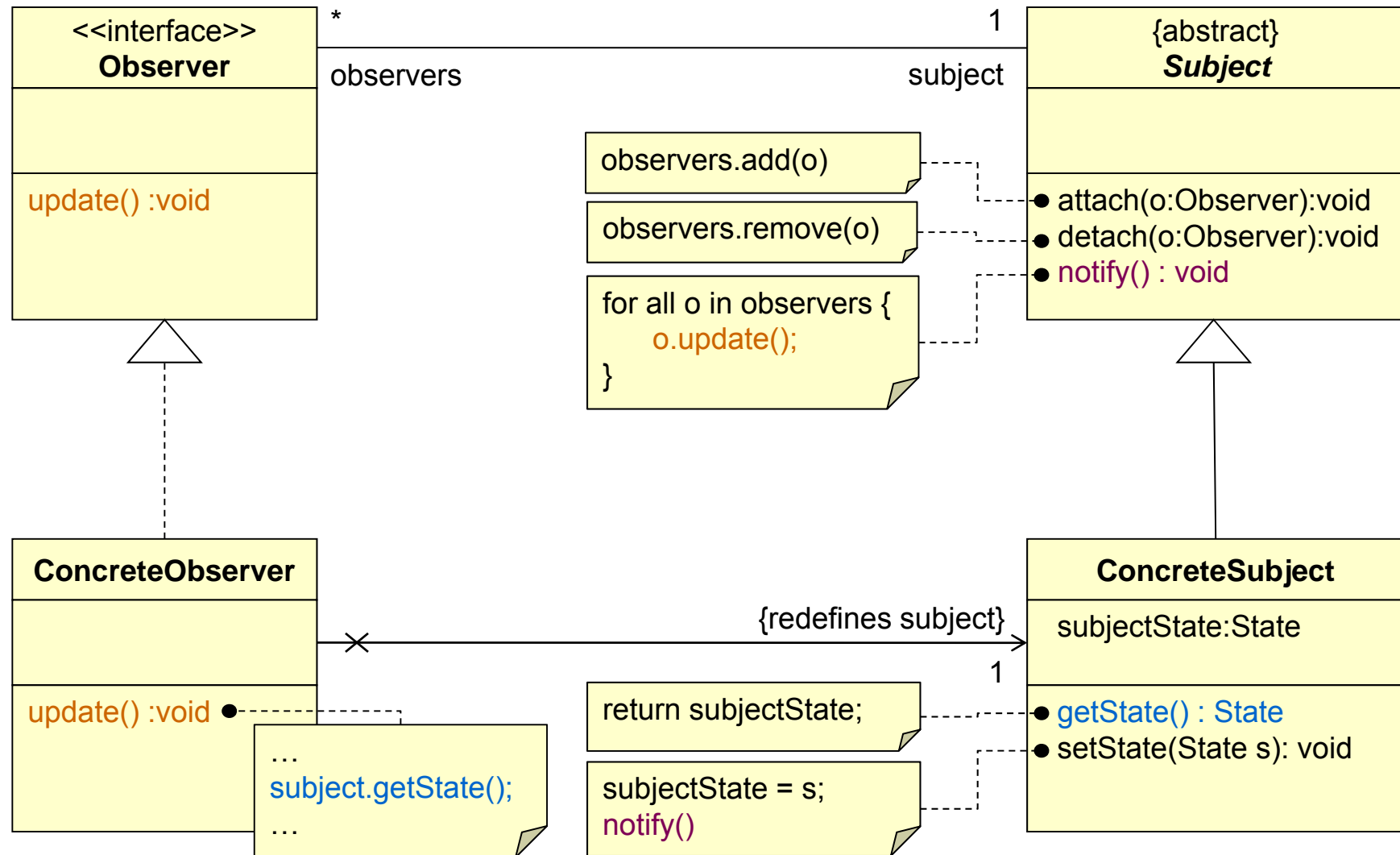
Das Observer Pattern: Anwendbarkeit

Das Pattern ist im folgenden Kontext anwendbar:



- **Abhängigkeiten**
 - ◆ Ein Aspekt einer Abstraktion ist abhängig von einem anderen Aspekt.
 - ◆ Aufteilung dieser Aspekte in verschiedene Objekte erhöht Variationsmöglichkeit und Wiederverwendbarkeit.
- **Folgeänderungen**
 - ◆ Änderungen an einem Objekt erfordert Änderungen an anderen Objekten.
 - ◆ Es ist nicht bekannt, wie viele Objekte geändert werden müssen.
- **Lose Kopplung**
 - ◆ Objekte sollen andere Objekte benachrichtigen können, ohne Annahmen über die Beschaffenheit dieser Objekte machen zu müssen.

Das Observer Pattern: Struktur (N:1, Pull-Modell)



Rollenaufteilung / Verantwortlichkeiten (Pull Modell)

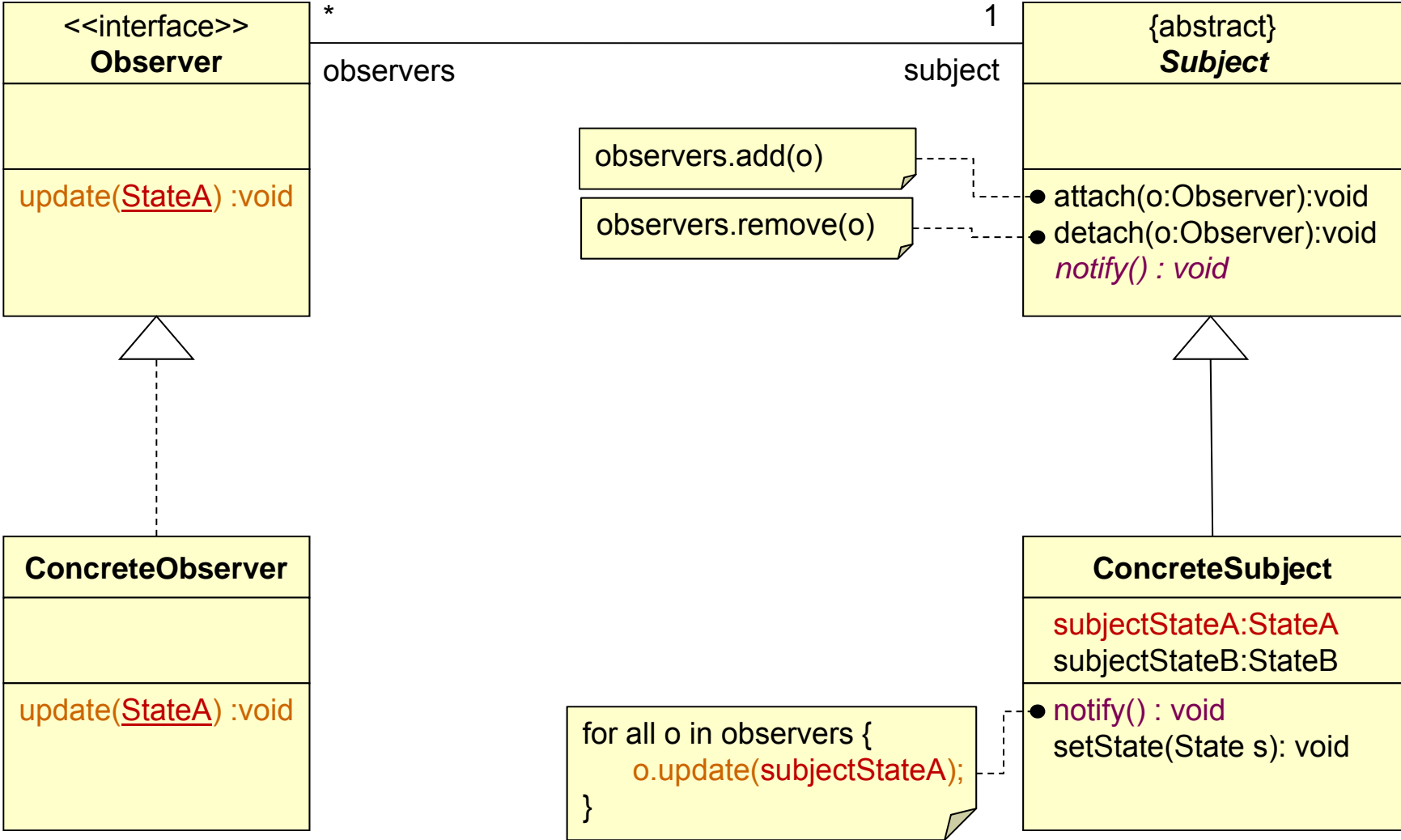
- Observer („Beobachter“) -- auch: Subscriber, Listener
 - ◆ `update()` -- auch: `handleEvent`
 - ⇒ Reaktion auf Zustandsänderung des Subjects
- Subject („Subjekt“) -- auch: Publisher
 - ◆ `attach(Observer o)` -- auch: `register`, `addListener`
 - ⇒ Observer registrieren
 - ◆ `detach(Observer o)` -- auch: `unregister`, `removeListener`
 - ⇒ registrierte Observer entfernen
 - ◆ `notify()`
 - ⇒ `update`-Methoden aller registrierten Observer aufrufen
 - ◆ `setState(...)`
 - ⇒ zustandsändernde Operation(en)
 - ⇒ für internen Gebrauch und beliebige Clients
 - ◆ `getState()`
 - ⇒ abfrage des aktuellen Zustands
 - ⇒ damit Observer feststellen können was sich wie geändert hat

Das Observer Patterns: Implementierung

Wie werden die Informationen über eine Änderung weitergegeben:
„push“ versus „pull“

- **Pull:** Subjekt übergibt in „update()“ keinerlei Informationen, aber die Beobachter müssen sich die Informationen vom Subjekt holen
 - ◆ + Geringere Kopplung zwischen Subjekt und Beobachter.
 - ◆ – Berechnungen werden häufiger durchgeführt.
- **Push:** Subjekt übergibt in Parametern von „update()“ detaillierte Informationen über Änderungen.
 - ◆ + Rückaufrufe werden seltener durchgeführt.
 - ◆ – Beobachter sind weniger wiederverwendbar (Abhängig von den Parametertypen)
- Zwischenformen sind möglich

Das Observer Pattern: Struktur (N:1, Push-Modell)



Das Observer Pattern: Implementierung

- Vermeidung irrelevanter Notifikationen durch Differenzierung von Ereignissen
 - ◆ Bisher: Notifikation bei jeder Änderung
 - ◆ Alternative: Beobachter-Registrierung und Notifikation nur für spezielle Ereignisse
 - ◆ Realisierung: Differenzierung von `attach()`, `detach()`, `update()` und `notify()` in jeweils ereignisspezifische Varianten
 - ◆ Vorteile:
 - ⇒ Notifikation nur für relevante Ereignisse → höhere Effizienz
 - ⇒ Weniger „Rückfragen“ pro Ereignis → höhere Effizienz
 - ◆ Nachteil: Mehr Programmieraufwand, wenn man sich für viele Ereignistypen interessiert
 - ⇒ Aber: Werkzeugunterstützung möglich

Das Observer Patterns: Implementierung

- ChangeManager

- ◆ Verwaltet Beziehungen zwischen Subjekt und Beobachter. (Speicherung in Subjekt und Beobachter kann entfallen.)
- ◆ Definiert die Aktualisierungsstrategie
- ◆ Benachrichtigt alle Beobachter. Verzögerte Benachrichtigung möglich
 - ⇒ Insbesondere wenn mehrere Subjekte verändert werden müssen, bevor die Aktualisierungen der Beobachter Sinn macht

- Wer ruft notify() auf?

- ◆ a) "setState()" -Methode des Subjekts:
 - ⇒ + Klienten können Aufruf von "notify()" nicht vergessen.
 - ⇒ – Aufeinanderfolgende Aufrufe von "setState()" führen zu evtl. überflüssigen Aktualisierungen.
- ◆ b) Klienten:
 - ⇒ + Mehrere Änderungen können akkumuliert werden.
 - ⇒ – Klienten vergessen möglicherweise Aufruf von "notify()".

Das Observer Pattern: Implementierung

- Konsistenz-Problem

- ◆ Zustand eines Subjekts muss vor Aufruf von "notify()" konsistent sein.
- ◆ Vorsicht bei Vererbung bei Aufruf jeglicher geerbter Methoden die möglicherweise „notify()“ -aufrufen :

```
public class MySubject extends SubjectSuperclass {  
    public void doSomething(State newState) {  
        super.doSomething(newState); // ruft "notify()" auf  
        this.modifyMyState(newState); // zu spät!  
    }  
}
```

- Lösung

- ◆ Dokumentation von „notify()“-Aufrufen erforderlich (Schnittstelle!)
- ◆ Besser: In Oberklasse „Template-Method Pattern“ anwenden um sicherzustellen, dass „notify()“-Aufrufe immer am Schluss einer Methode stattfinden → s. nächste Folie.

Das Observer Pattern: Implementierung

Verwendung des „Template Method Pattern“

```
public class SubjectSuperclass {  
    ...  
    final public void doSomething(State newState) {  
        this.doItReally(newState);  
        this.notify();           // notify immer am Schluß  
    }  
    public void doItReally(State newState) {  
    }  
}
```

Template Method

Hook Method

```
public class MySubject extends SubjectSuperclass {  
    public void doItReally(State newState) {  
        this.modifyMyState(newState);  
    }  
}
```

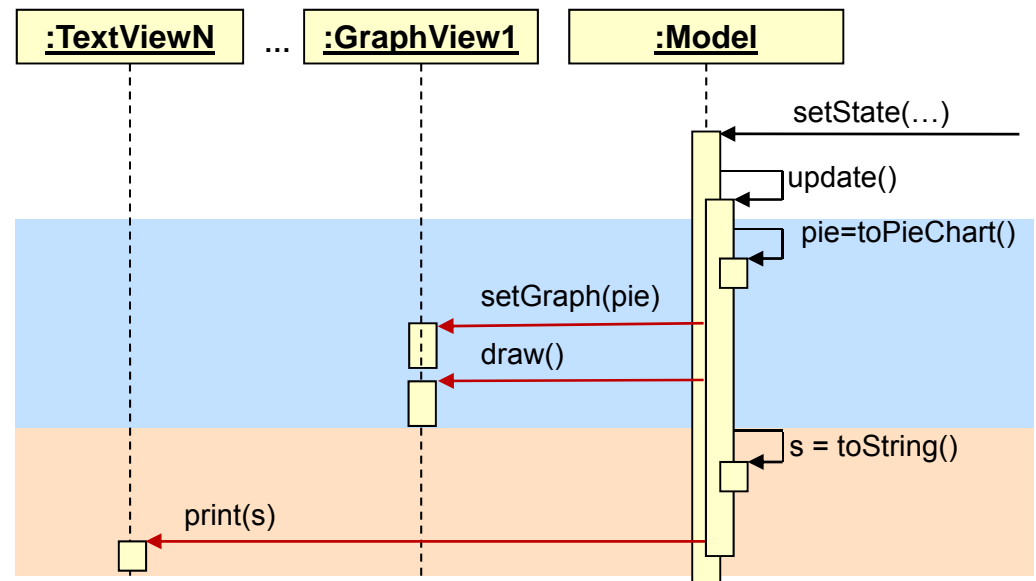
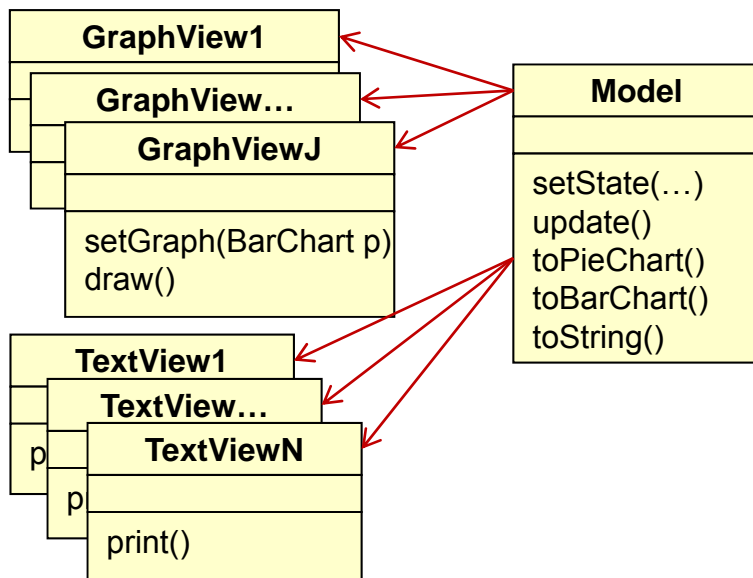
Konsequenzen von Observer für Abhängigkeiten

Für Abhängigkeitsreduzierung allgemein
Für Presentation-Application-Data (PAD)
Für Boundary-Controller-Entity (BCE)
Für Model-View-Kontroller (MVC)
PAD versus BCE versus MVC

Abhängigkeiten mit und ohne Observer

Ohne Observer: Abhängigkeit View ← Model

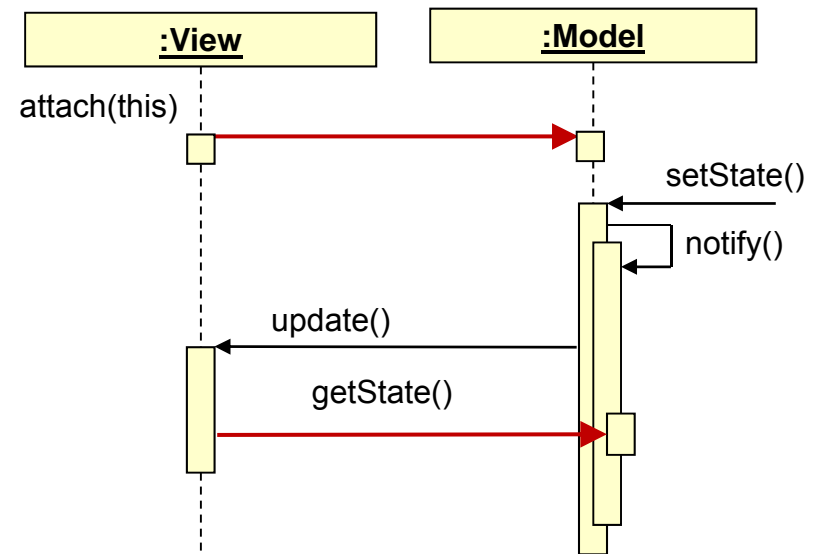
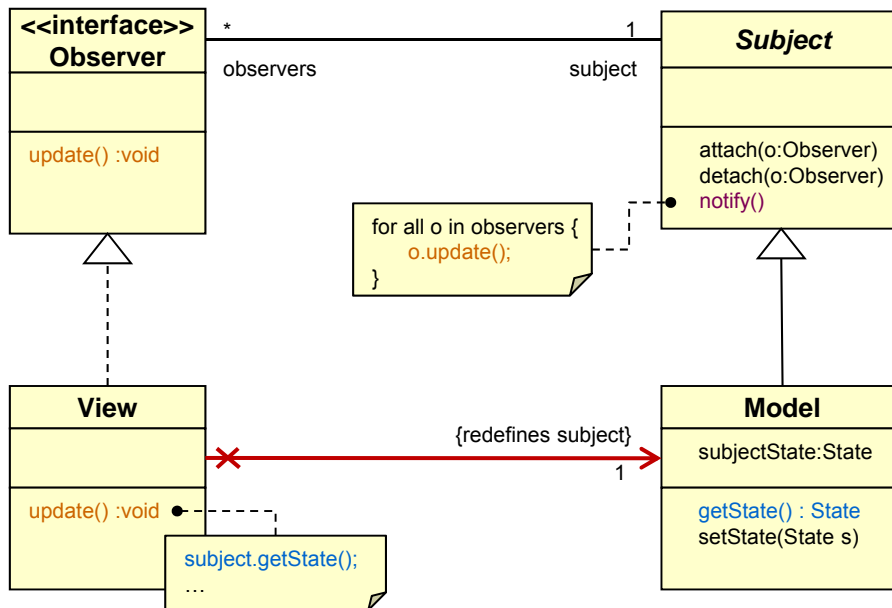
- Ein konkretes Model kennt das Interface eines jeden konkreten View und steuert was genau jeder der Views nach einem update tun soll:
 - ◆ `myGraphView1.setGraph(this.toPieChart()); myGraphView1.draw();`
 - ◆ `myGraphView2.setGraph(this.toBarChart()); myGraphView2.draw();`
 - ◆ ...
 - ◆ `myTextViewN.print(myState.toString());`



Abhängigkeiten mit und ohne Observer

Mit Observer: Abhängigkeit View → Model

- Ein konkretes Model kennt nur das abstrakte Observer-Interface
- Ein konkreter View kennt die zustandsabfragenden Methoden des konkreten Models
 - ◆ `model.getState1();`
 - ◆ `model.getState2();`



Nettoeffekt: Abhängigkeits- und Kontrollumkehrung

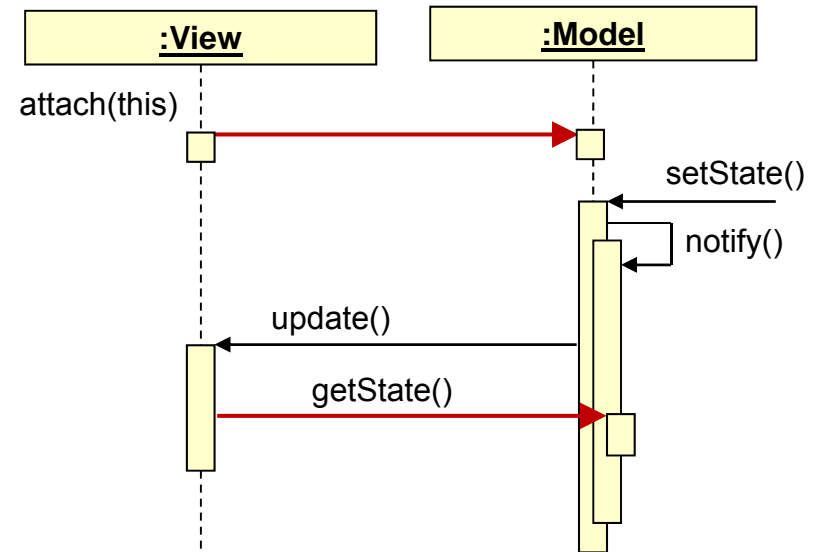
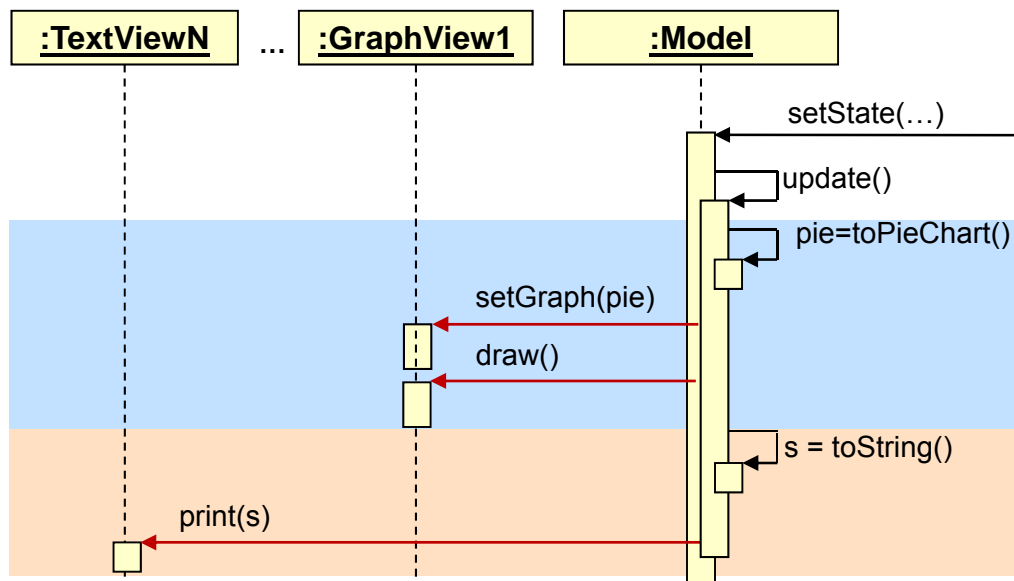
Abhängigkeitsumkehrung („Dependency Inversion“)

- Ohne Observer
 - ◆ Abhängigkeit Model → Views
- Mit Observer
 - ◆ Abhängigkeit Model ← Views

Kontrollumkehrung („Inversion of Control“)

- Ohne Observer
 - ◆ Model steuert alle updates
- Mit Observer
 - ◆ Jeder View steuert sein update

Vergleich Ablauf ohne / mit Observer



Das Observer Pattern: Konsequenzen

- Unabhängigkeit
 - ◆ Konkrete Beobachter können **hinzugefügt** werden, ohne konkrete Subjekte oder andere konkrete Beobachter zu ändern.
 - ◆ Konkrete Subjekte können unabhängig voneinander und von konkreten Beobachtern **variiert** werden.
 - ◆ Konkrete Subjekte können unabhängig voneinander und von konkreten Beobachtern **wiederverwendet** werden.
- „Broadcast“-Nachrichten
 - ◆ Subjekt benachrichtigt alle angemeldeten Beobachter
 - ◆ Beobachter entscheiden, ob sie Nachrichten behandeln oder ignorieren
- Unerwartete Aktualisierungen
 - ◆ Kleine Zustandsänderungen des Subjekts können komplexe Folgen haben.
 - ◆ Auch uninteressante Zwischenzustände können unnötige Aktualisierungen auslösen.

Das Observer Pattern: Bekannte Anwendungen

- Bekannte Anwendungen
 - ◆ Model-View-Controller-Framework in Smalltalk
 - ◆ Java Foundation Classes / Swing
 - ◆ Oberon System
 - ◆ Diverse Client/Server-Modelle, z.B. Java RMI

Hier geht's weiter am Di, 30.11.2010

Was also sind Patterns?

Definition: Was ist jetzt also ein Pattern?

- Definitionsvorschlag

"Each pattern is a three-part rule,
which expresses a relation between a certain **context**,
a certain system of **forces** which occurs repeatedly in that context, and
a certain **software configuration** which allows these forces to resolve
themselves."

Richard Gabriel, on the Patterns Home Page

Bestandteile eines Patterns: Kontext, Problem, Randbedingungen

- Anwendbarkeit (**Context**) – Applicability
 - ◆ Die Vorbedingungen unter denen das Pattern benötigt wird.
- Problem
 - ◆ Beschreibung des Problems oder der Absicht des Patterns
 - ◆ Ziel und gewünschte Eigenschaften die in einem bestimmten Kontext mit bestimmten Anforderungen erreicht werden sollen.
 - ◆ Die Kräfte und die Ziele scheinen sich zu widersprechen.
- Anforderungen (**Forces**)
 - ◆ Die relevanten Anforderungen und Einschränkungen, die berücksichtigt werden müssen.
 - ◆ Wie diese miteinander interagieren und im Konflikt stehen.
 - ◆ Die daraus entstehenden Kosten.
 - ◆ Typischerweise durch ein motivierendes Szenario illustriert.

Bestandteile eines Patterns: Lösung

Die Lösung (**Software Configuration**) wird beschrieben durch:

- Rollen

- ◆ Funktionen die Programmelemente im Rahmen des Patterns erfüllen.
- ◆ Interfaces, Klassen, Methoden, Felder / Assoziationen
- ◆ Methodenaufrufe und Feldzugriffe
- ◆ Sie werden bei der Implementierung auf konkrete Programmelemente abgebildet („Player“)

- Statische + dynamische Beziehungen

- ◆ Klassendiagramm, dynamische Diagramme, Text
- ◆ Meistens ist das Verständnis des dynamischen Verhaltens entscheidend
 - ⇒ Denken in Objekten (Instanzen) statt Klassen (Typen)!

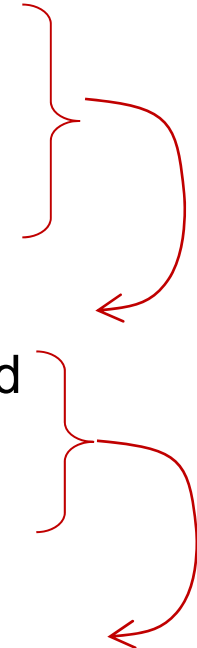
- Teilnehmer – Participants

- ◆ Rollen auf Typebene (Klassen und Interfaces)

Bestandteile einer Pattern-Beschreibung

Name(n) des Patterns

- **Problem**, das vom Pattern gelöst wird
- **Kontext**, in dem das Pattern angewendet werden kann
- **Anforderungen**, die das Pattern beeinflussen
- **Lösung**. Beschreibung, wie das gewünschte Ergebnis erzielt wird
 - ◆ **Varianten** der Lösung
 - ◆ **Beispiele** der Anwendung der Lösung
- **Konsequenzen** aus der Anwendung des Patterns
- **Bezug** zu anderen Patterns
- **Bekannte Verwendungen** des Patterns



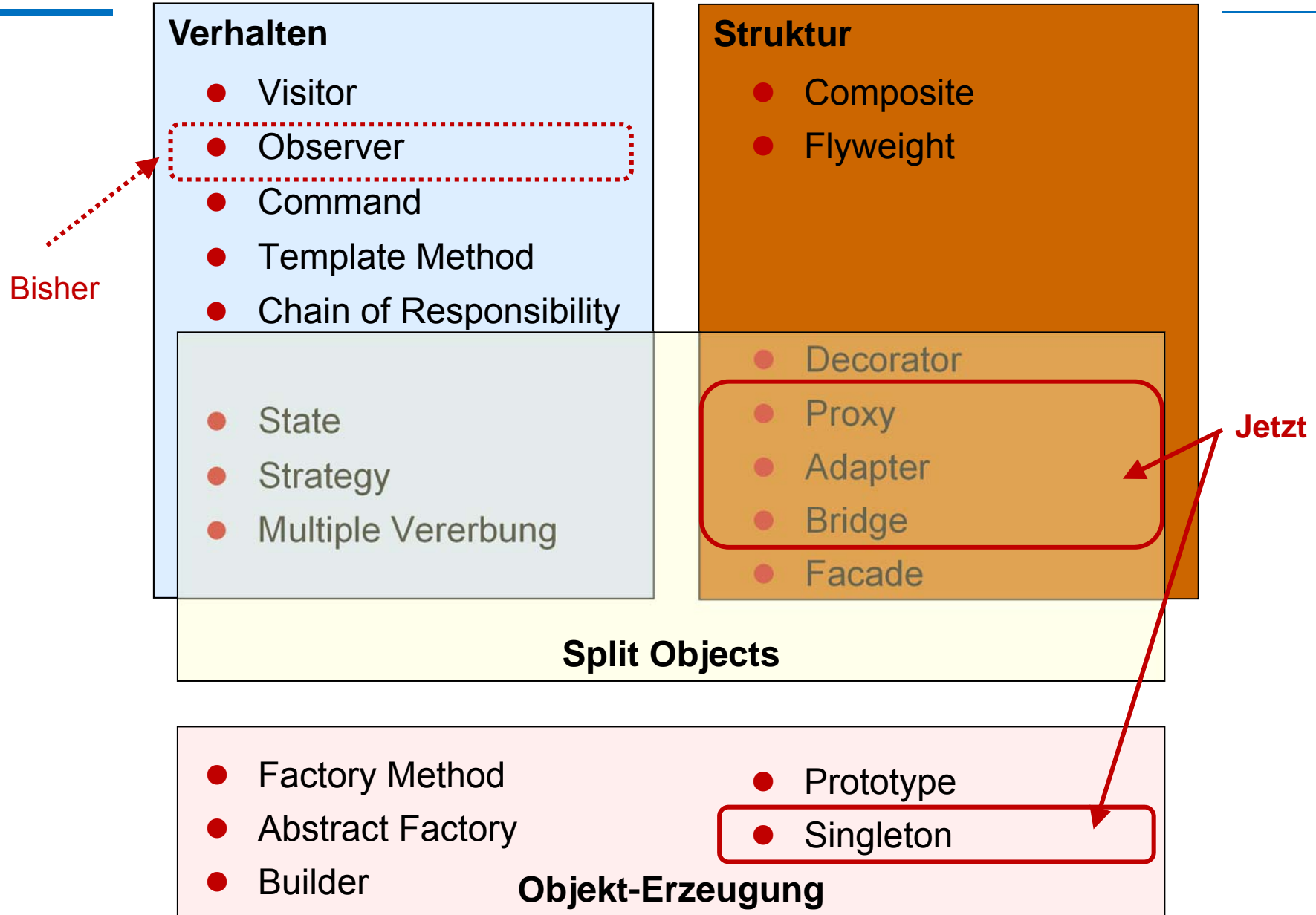
Klassifikation

Danach, **was** sie modellieren

Danach, **wie** sie es modellieren

Danach, **wo** sie meist eingesetzt werden

"Gang of Four"-Patterns: Überblick und Klassifikation



Klassifikation: „Was“ und „Wie“

Was?

- Verhaltens-Patterns
 - ◆ Verhalten leicht erweiterbar, komponierbar, dynamisch änderbar oder explizit manipulierbar machen
- Struktur-Patterns
 - ◆ Objekte mit fehlendem Zustand, rekursive Strukturen
 - ◆ Verschiedenste Formen der Entkopplung (Schnittstelle / Implementierung, Identität / physikalische Speicherung, ...)
- Erzeugungs-Patterns
 - ◆ Flexibilität indem die Festlegung von konkret zu erzeugenden Objekte (new XYZ()) so weit wie möglich verzögert wird und evtl. sogar zur Laufzeit immer noch verändert werden kann.

Wie?

- Split Object Patterns
 - ◆ Ziel: Dynamisch änderbares Verhalten, gemeinsame Verwendung oder Entkopplung von Teilobjekten
 - ◆ Weg: Aufteilung eines konzeptionellen Gesamtobjektes in modellierte Teilobjekte die kooperieren um das Verhalten des konzeptionellen Gesamtobjektes zu realisieren

Klassifikation: „Was“ und „Wie“ (Fortsetzung)

- Klassifikation danach **was** modelliert wird (vorherige Folien)
 - ◆ Struktur, Verhalten, Objekterzeugung
- Klassifikation danach, **wie** etwas modelliert wird (vorherige Folien)
 - ◆ **Whole Objects**: 1:1-Entsprechung von konzeptuellen Objekten und Implementierungs-Objekten
 - ◆ **Split Objects**: Aufteilung eines konzeptuellen Objektes in verschiedene Implementierungs-Objekte, dynamisch veränderbare Anteile oder Gemeinsamkeiten verschiedener konzeptueller Objekte darstellen

Klassifikation: „Wo“

- Klassifikation danach, **wo** sie meist eingesetzt werden
 - ◆ Systementwurf, zur Verbindung / Abgrenzung von Subsystemen
 - ⇒ Facade, Adapter, Bridge, Proxy, Singleton (zusammen mit Facade) , Observer
 - ◆ Objektentwurf
 - ⇒ Observer, Command, Factory Method, Abstract Factory, Composite, Visitor

Wichtige Entwurfsmuster, Teil 1

- System Design Patterns -

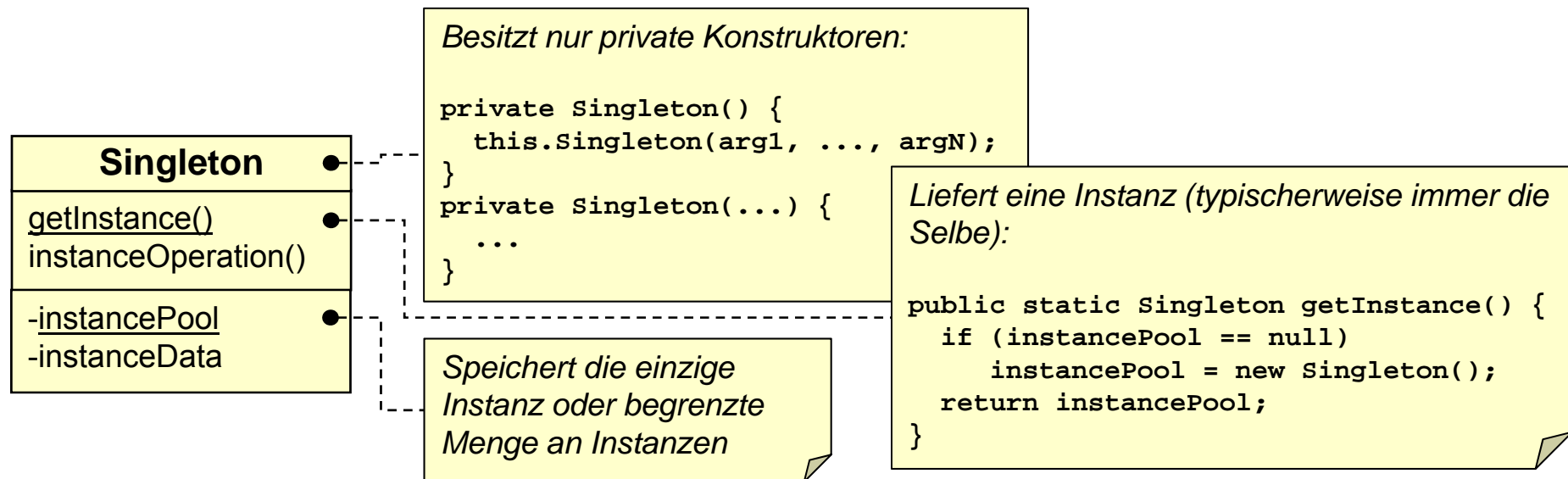
Facade
Singleton
Adapter
Proxy
Bridge

Das Singleton Pattern

Singleton: Motivation

- Beschränkung der Anzahl von Exemplaren zu einer Klasse
- Meist: nur ein einzelnes Exemplar
 - ◆ Motivation: Zentrale Kontrolle → Z.B. Facade, Repository, Abstract Factory
- Aber auch: feste Menge von Exemplaren
 - ◆ Motivation 1: begrenzte Ressourcen (z.B. auf mobilen Geräten)
 - ◆ Motivation 2: Teure Objekterzeugung durch „Object Pool“ vermeiden
→ z.B. 1000 Enterprise Java Beans vorhalten, nach Nutzung zurück in den Pool

Singleton: Struktur + Implementierung

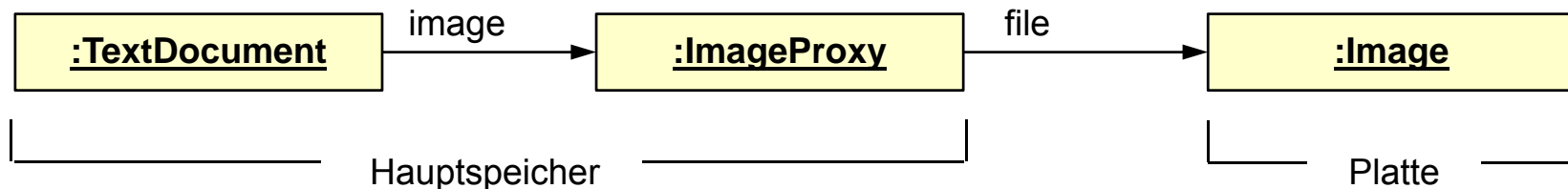


- Nur private Konstruktoren
 - ◆ dadurch wird verhindert, dass Clients beliebig viele Instanzen erzeugen können
 - ◆ in Java muss **explizit ein privater** Konstruktor mit leerer Argumentliste implementiert werden, damit **kein impliziter öffentlicher** Konstruktor vom Compiler erzeugt wird
- `instancePool` als Registry für alle Singleton-Instanzen
 - ◆ lookup-Mechanismus erforderlich um gezielt eine Instanz auszuwählen

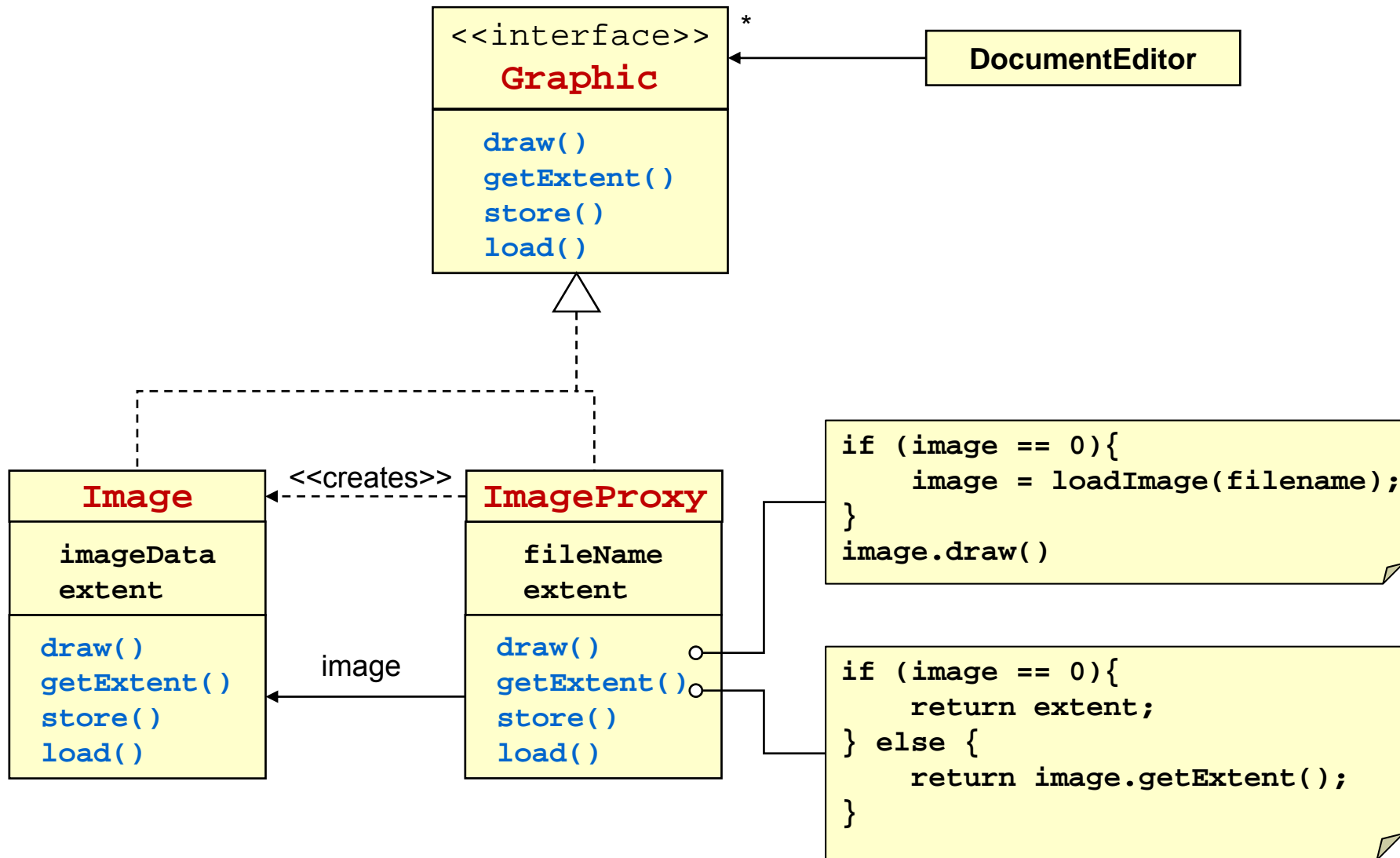
Das Proxy Pattern

Proxy Pattern (auch: Surogate, Smart Reference)

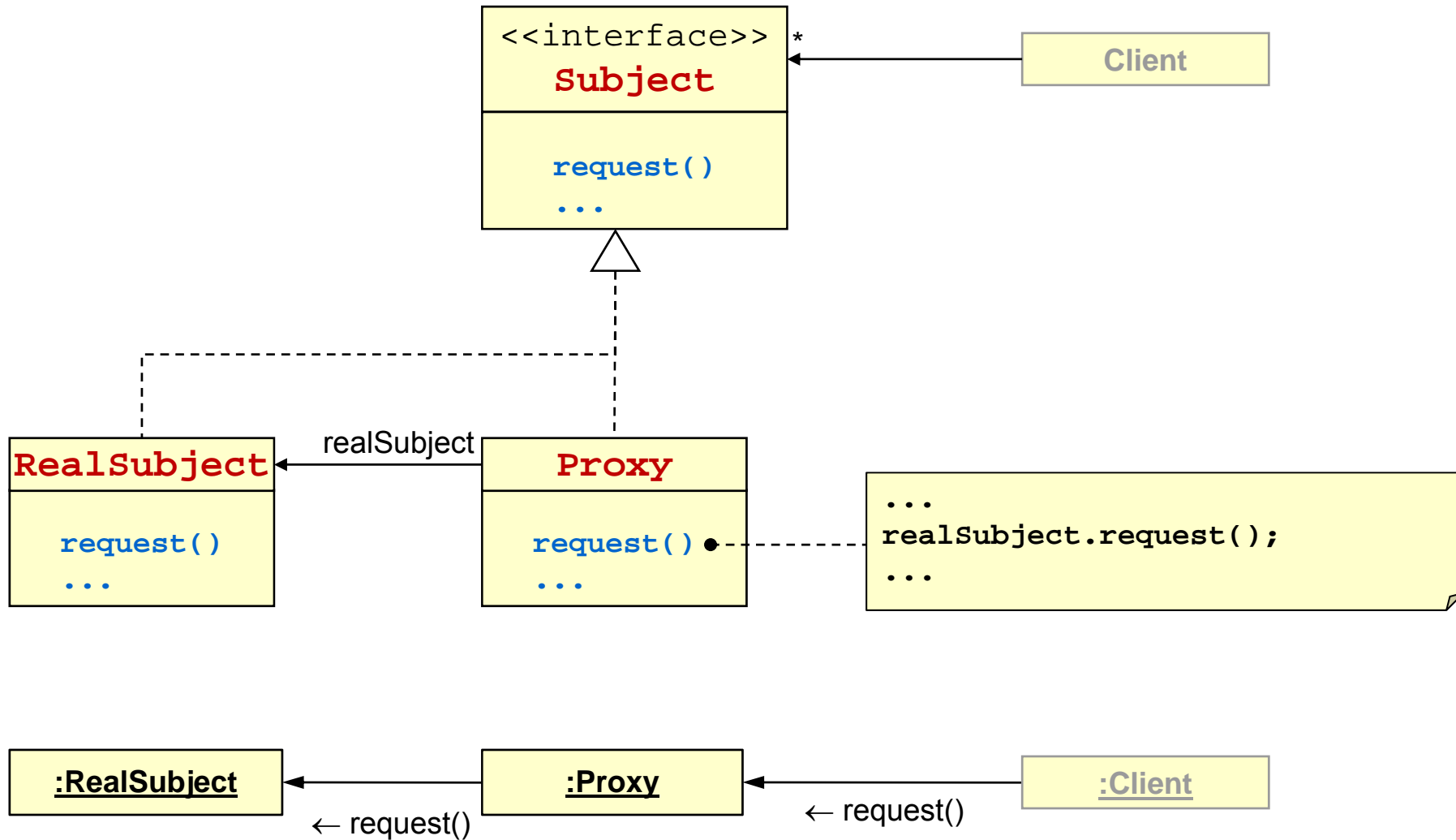
- Absicht
 - ◆ Stellvertreter für ein anderes Objekt
 - ◆ bietet Kontrolle über Objekt-Erzeugung und -Zugriff
- Motivation
 - ◆ kostspielige Objekt-Erzeugung verzögern (zB: große Bilder)
 - ◆ verzögerte Objekterzeugung soll Programmstruktur nicht global verändern
- Idee
 - ◆ Bild-Stellvertreter (Proxy) verwenden
 - ◆ Bild-Stellvertreter verhält sich aus Client-Sicht wie Bild
 - ◆ Bild-Stellvertreter erzeugt Bild bei Bedarf



Proxy Pattern: Beispiel



Proxy Pattern: Schema



Proxy Pattern: Verantwortlichkeiten

- Proxy
 - ◆ bietet gleiches Interface wie "Subject"
 - ◆ referenziert eine "RealSubject"-Instanz
 - ◆ kontrolliert alle Aktionen auf dem "RealSubject"
- Subject
 - ◆ definiert das gemeinsame Interface
- RealSubject
 - ◆ das Objekt das der Proxy vertritt
 - ◆ eigentliche Funktionalität
- Zusammenspiel
 - ◆ selektives Forwarding

Proxy Pattern: Anwendbarkeit

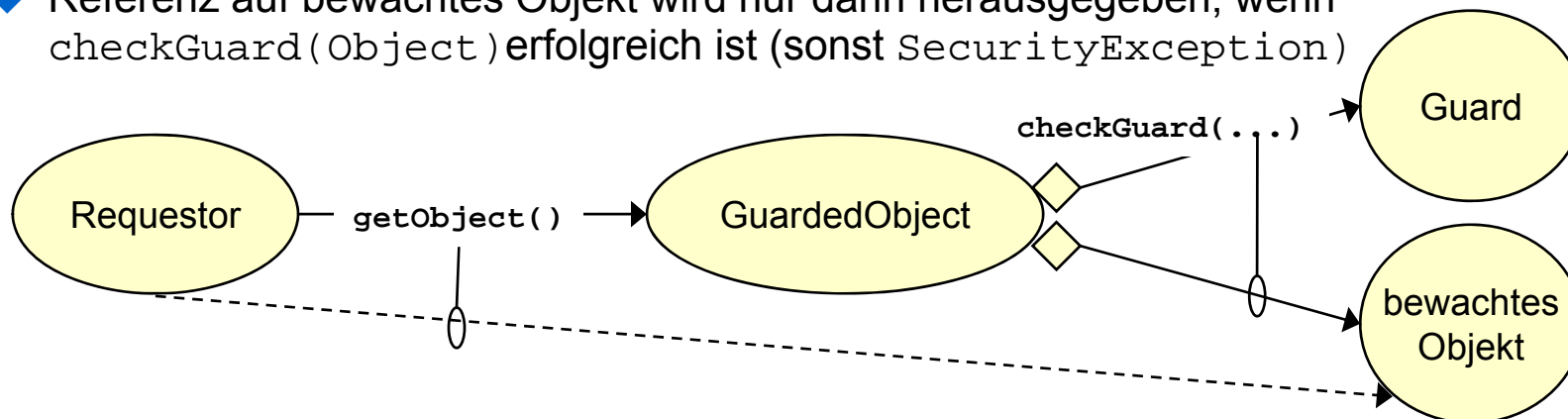
- Virtueller Proxy
 - ◆ verzögerte Erzeugung "teurer" Objekte bei Bedarf
 - ◆ Beispiel: Bilder in Dokument, persistente Objekte
- Remote Proxy
 - ◆ Zugriff auf entferntes Objekt
 - ◆ Objekt-Migration
 - ◆ Beispiele: CORBA, RMI, mobile Agenten
- Concurrency Proxy
 - ◆ nur eine direkte Referenz auf RealSubject
 - ◆ locking des RealSubjects vor Zugriff (threads)
- Copy-on-Write
 - ◆ kopieren erhöht nur internen "CopyCounter"
 - ◆ wirkliche Kopie bei Schreibzugriff und "CopyCounter">0
 - ➔ Verzögerung teurer Kopier-Operationen

Proxy Pattern: Anwendbarkeit

- Protection Proxy (Zugriffskontrolle)
 - ◆ Schmaleres Interface
 - ⇒ "Kritische" Operationen ausgeblendet
 - ⇒ andere via Forwarding implementiert
 - ◆ ganz anderes Interface
 - ⇒ Autorisierung prüfen
 - ⇒ direkten Zugriff gewähren oder Forwarding
 - ◆ Beispiel: "CHOICES" Betriebssystem, JDK

Protection-Proxy im JDK (ab 1.2): GuardedObject

- Problem
 - ◆ Sichere Weitergabe eines schützenswerten Objektes an unbekannten Empfänger
 - ◆ Objektspezifische Zugriffsrechte
 - ◆ Verzögerte Überprüfung der Zugriffsrechte
- Idee: GuardedObject
 - ◆ Enthält "bewachtes Objekt" und "Wächter" (Guard)
 - ◆ Guard-Interface enthält nur die Methode `checkGuard(Object)`
 - ◆ Referenz auf bewachtes Objekt wird nur dann herausgegeben, wenn `checkGuard(Object)` erfolgreich ist (sonst `SecurityException`)



- Verbleibendes Problem
 - ◆ Man muß sich darauf verlassen, daß der "Requestor" das "bewachte Objekt" selbst nur in ein GuardedObject verpackt weitergibt!

Proxy Pattern: Implementierung

- „Consultation“ = Weiterleiten von Anfragen
 - ◆ Allgemein: manuell erstellte Forwarding-Methoden
 - ◆ C++: Operator-Overloading
 - ⇒ Proxy redefiniert Dereferenzierungs-Operator: *anImage
 - ⇒ Proxy redefiniert Member-Access-Operator: anImage->extent()
 - ◆ Smalltalk: Reflektion
 - ⇒ Proxy redefiniert Methode "doesNotUnderstand: aMessage"
 - ◆ Lava: eigenes Sprachkonstrukt
 - ⇒ Proxy deklariert "consultee"-Variable

```
class Proxy {  
    private consultee ReallImage reallImage;  
    ...  
}
```
- Falls der Typ von "RealSubject", dem Proxy bekannt sein muß:
 - ◆ Je eine spezifische Proxy-Klasse für jeden RealSubject-Typ
- ... dem Proxy nicht bekannt sein muß:
 - ◆ Nur eine Proxy-Klasse für verschiedene RealSubject-Typen
 - ⇒ Beispiel: „Guarded Object“ (vorherige Folie)

Proxy Pattern: Implementierung

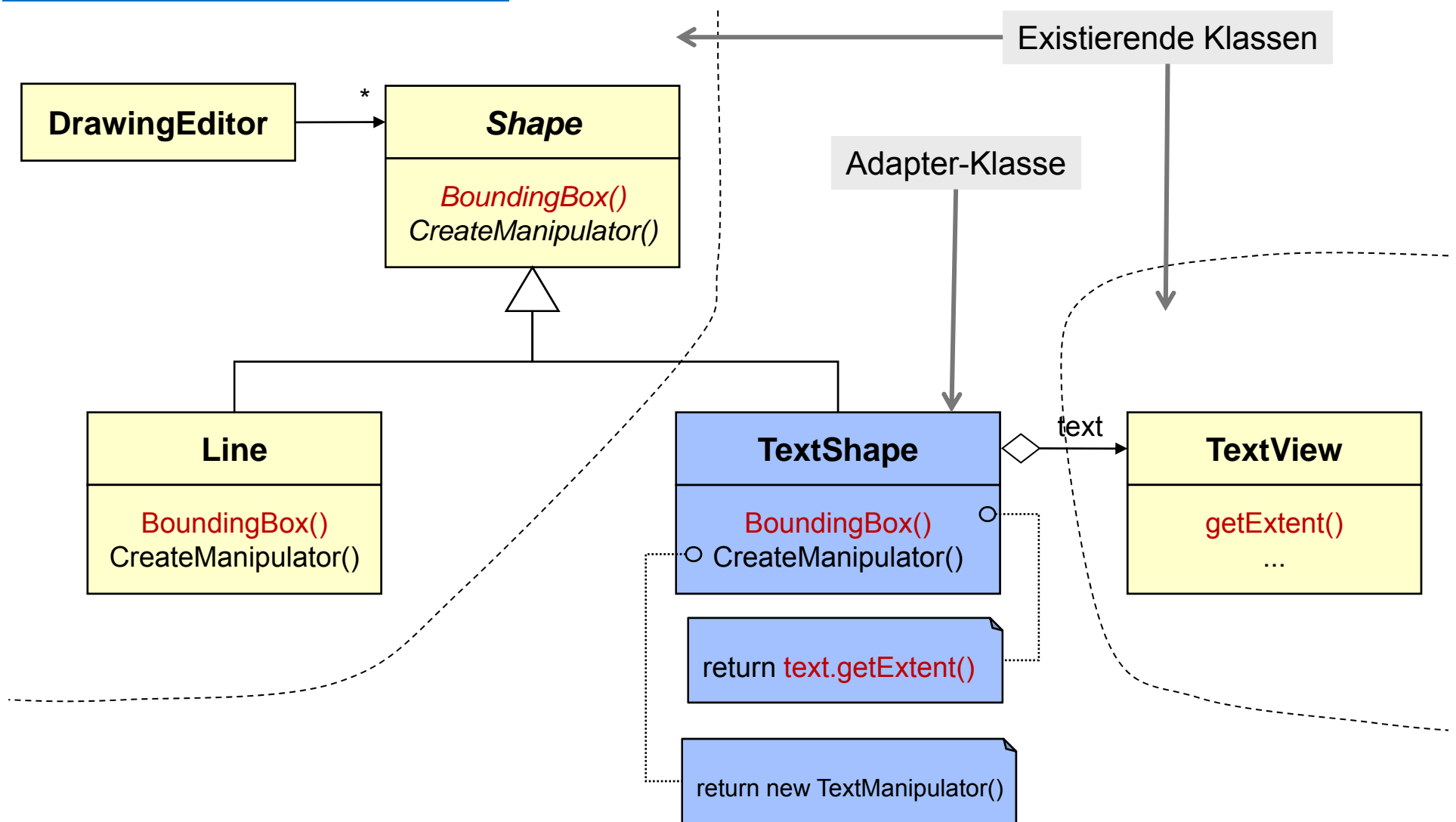
- Referenzierung des RealSubject vor Instantiierung
 - ◆ Orts- und Zeit-unabhängige "Ersatz-Referenzen"
 - ◆ Beispiele
 - ⇒ Dateinamen
 - ⇒ CORBA IOR (Interoperable Object Reference)

Das Adapter Pattern

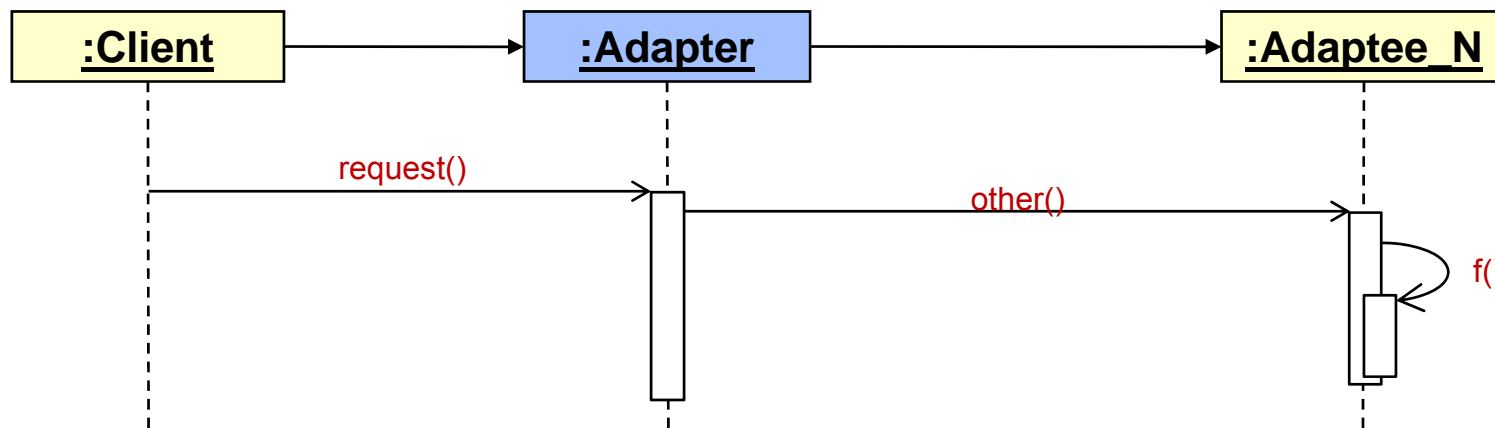
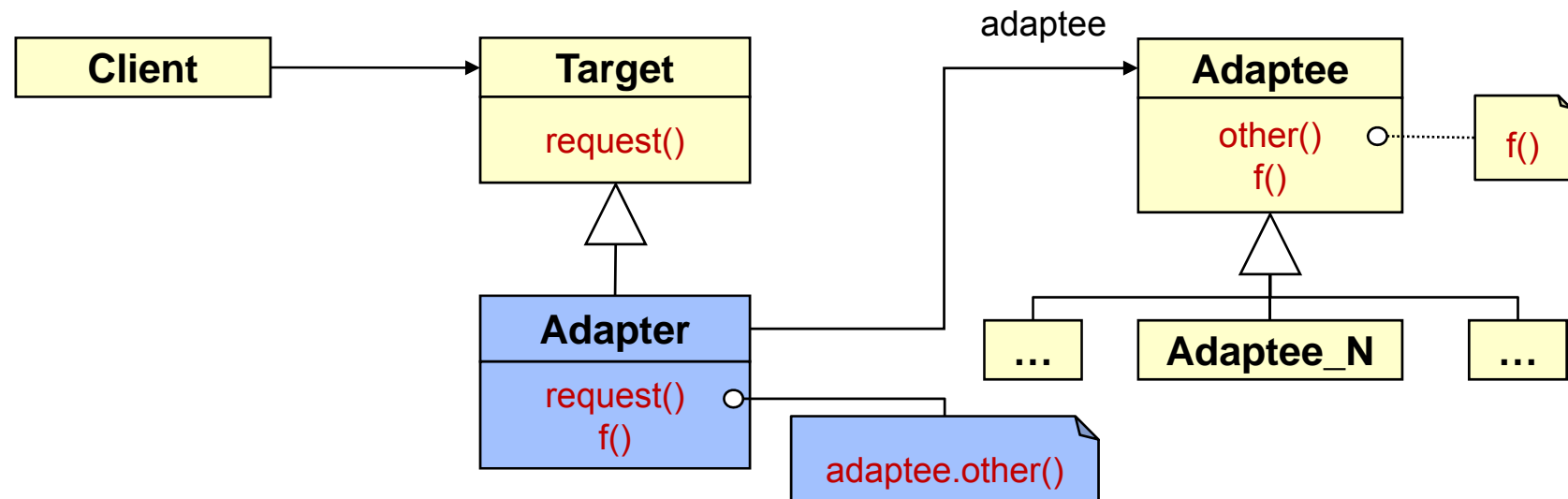
Adapter Pattern (auch: Wrapper)

- Absicht
 - ◆ Schnittstelle existierender Klasse an Bedarf existierender Clients anpassen
- Motivation
 - ◆ Graphik-Editor bearbeitet Shapes
 - ⇒ Linien, Rechtecke, ...
 - ⇒ durch "BoundingBox" beschrieben
 - ◆ Textelemente sollen auch möglich sein
 - ⇒ Klasse TextView vorhanden
 - ⇒ bietet keine "BoundingBox"-Operation
 - ◆ Integration ohne
 - ⇒ Änderung der gesamten Shape-Hierarchie und ihrer Clients
 - ⇒ Änderung der TextView-Klasse
- Idee
 - ◆ Adapter-Klasse stellt Shape-Interface zur Verfügung
 - ◆ ... implementiert Shape-Interface anhand der verfügbaren TextView-Methoden

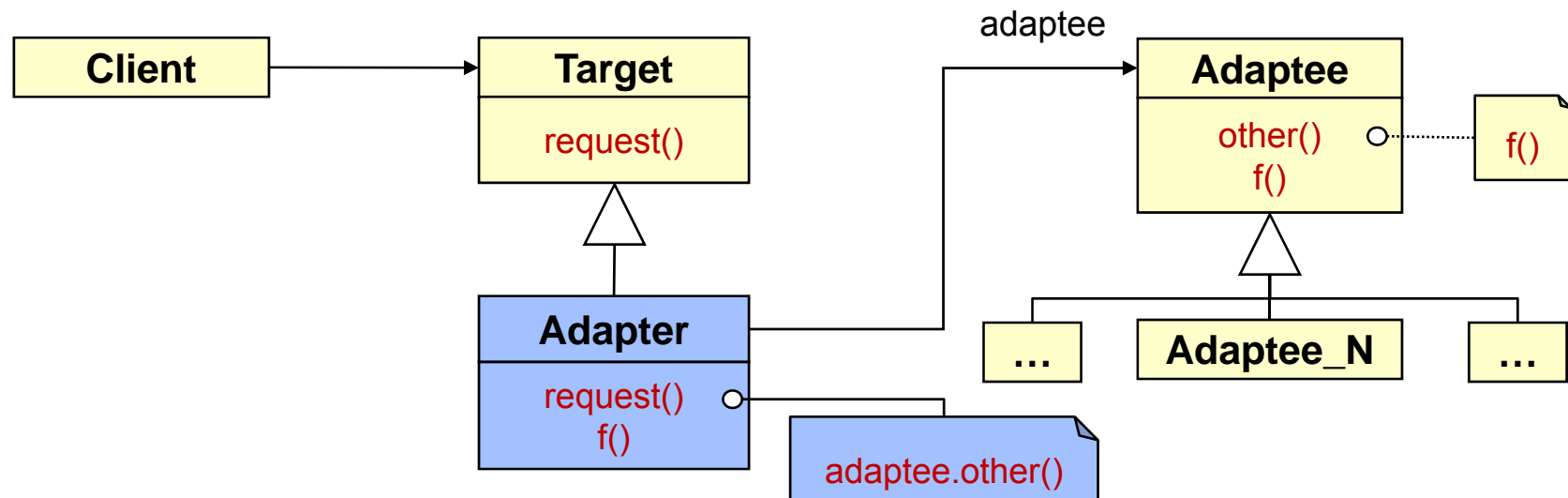
Adapter Pattern: Beispiel



Adapter Pattern (Objekt-Adapter): Schema

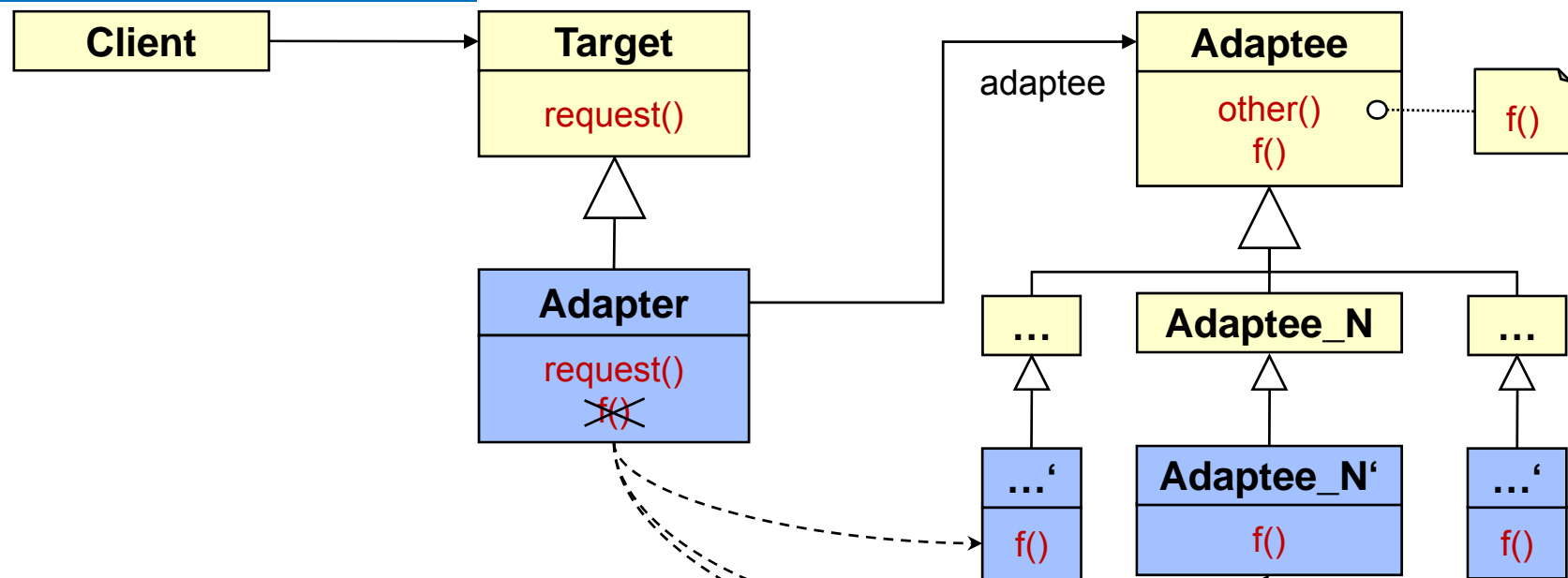


Adapter Pattern (Objekt-Adapter): Konsequenzen



- Adaptiert eine ganze Klassenhierarchie
 - ◆ Beliebige Adapter-Subtypen können mit beliebigen Adaptee-Subtypen kombiniert werden (→ siehe Objektdiagramm auf vorheriger Folie)
 - ◆ Kombination wird erst zur Laufzeit, auf Objektebene, festgelegt
- Overriding nicht möglich
 - ◆ Die f()-Definition aus Adapter wird nicht anstelle der f()-Definition aus Adaptee für den f()-Aufruf aus der Methode other() verwendet (→ siehe Objektdiagramm auf vorheriger Folie)

Adapter Pattern (Object Adapter): Variante



- Object Adapter mit Overriding

- ◆ zu Adaptee und jede Unterklasse des Adaptees eine weitere Unterklasse einfügen, in die das redefinierte Verhalten eingefügt wird (also z.B. f())

- ⇒ Warum neue Unterklassen? Weil die vorhandene Adaptee-Hierarchie nicht veränderbar ist!

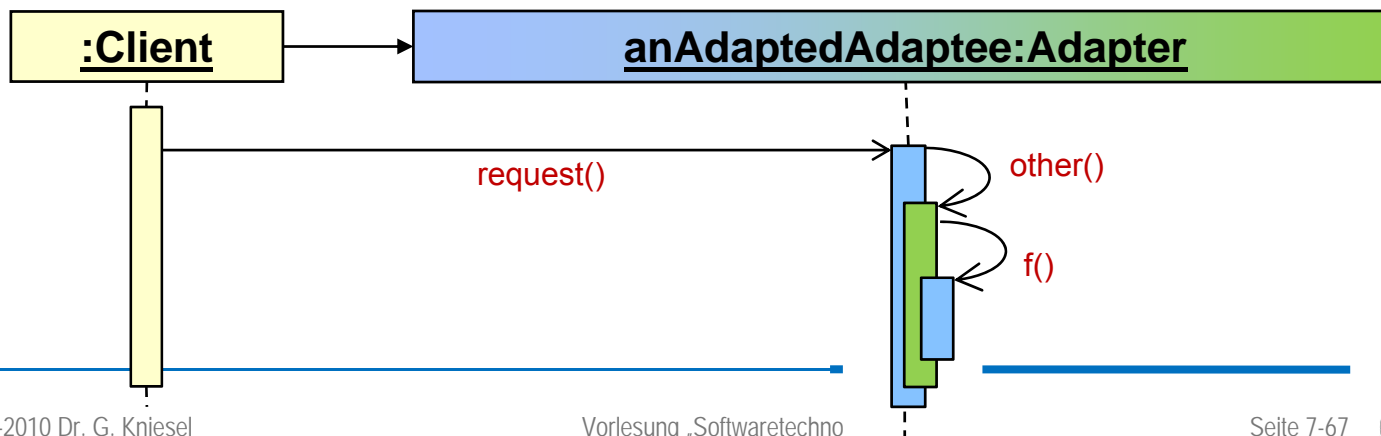
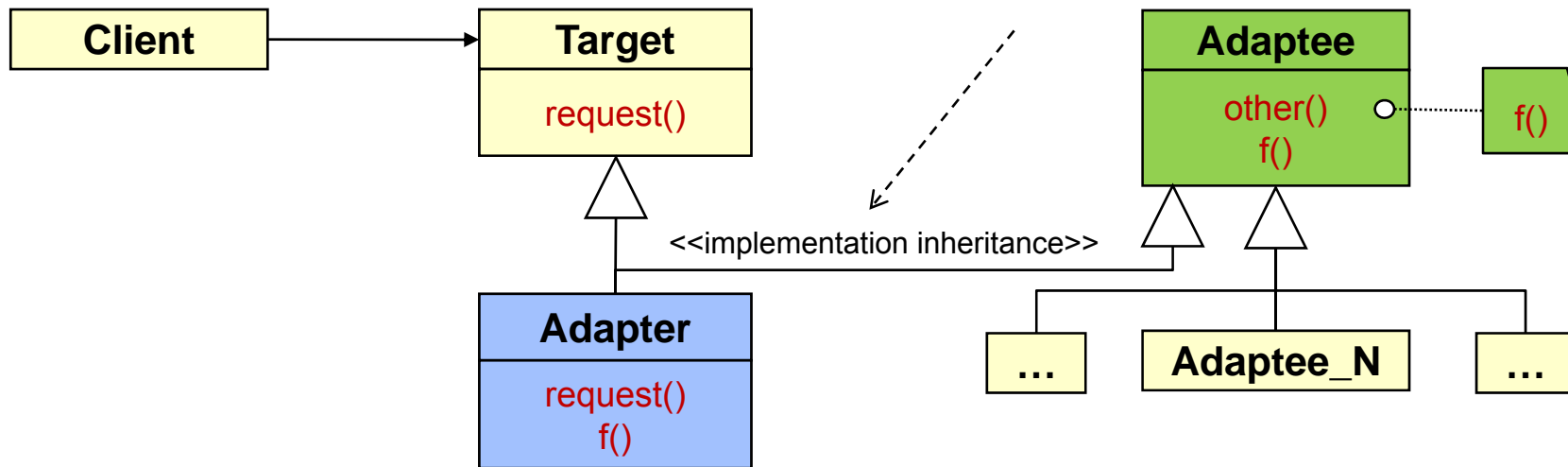
- ◆ Problem: Redundanz!

- ⇒ Was konzeptionell nur ein mal in „Adapter“ stehen sollte wird in jeder neuen Unterklasse von „Adaptee“ redundant eingefügt → Wartungsprobleme!

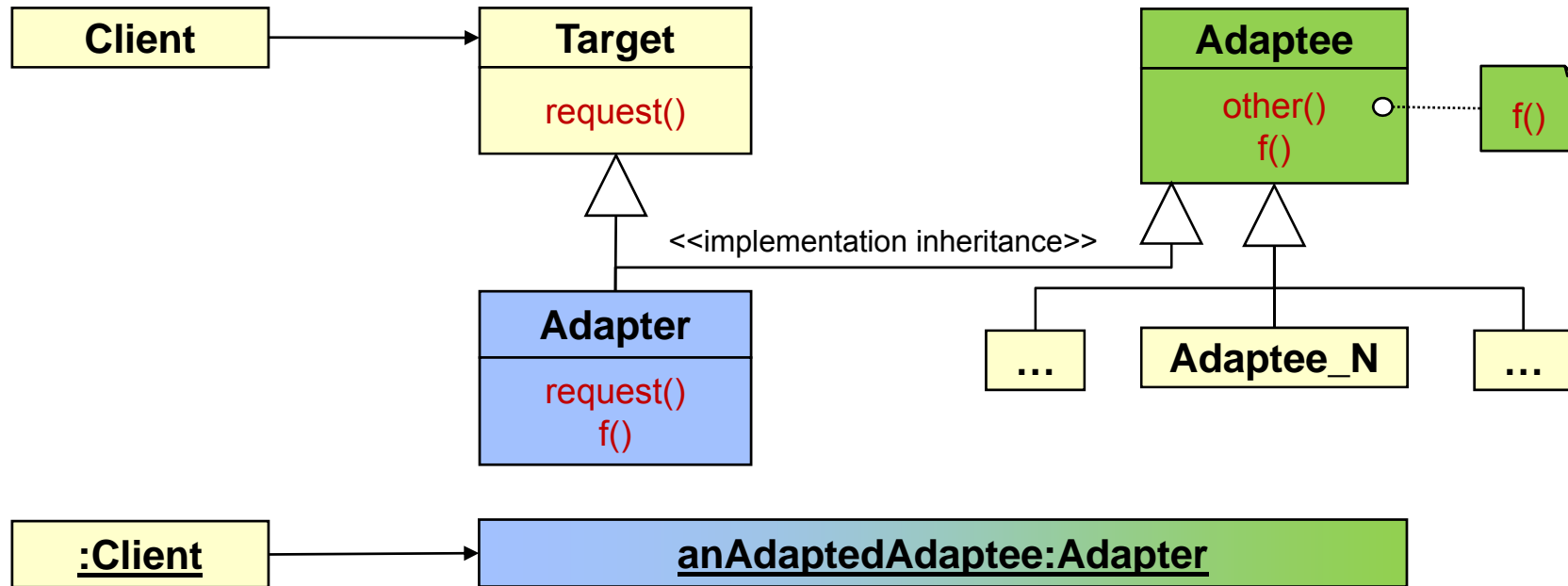
Class Adapter Idiom

„Vererbung ohne Subtyping“:
 Erbender erbt Methoden die nicht als
 Teil seines Interface sichtbar werden.

- "private inheritance" in C++
- "closed inheritance" in Eiffel



Class Adapter Idiom: Konsequenzen

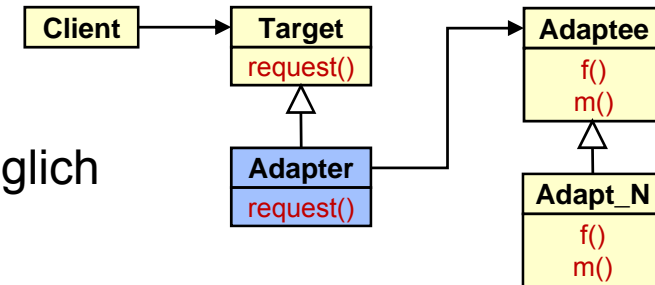


- Overriding des Adaptee-Verhaltens leicht möglich
 - ◆ Da Adapter eine Unterklasse von Adaptee ist, funktioniert das Overriding von f()
- Keine zusätzliche Indirektion
 - ◆ nur ein Objekt statt zwei
- Adaptiert nur eine Klasse
 - ◆ Die gelben Unterklassen von Adaptee werden nicht mit adaptiert

Adapter Pattern: Konsequenzen

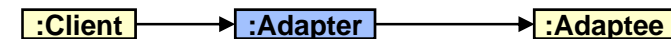
- Class Adapter

- ◆ adaptiert nur eine Klasse
- ◆ ... nicht ihre Unterklassen
- ◆ Overriding des Adaptee-Verhaltens leicht möglich
- ◆ keine zusätzliche Indirektion (nur ein Objekt)



- Object Adapter

- ◆ adaptiert eine ganze Klassenhierarchie
- ◆ Overriding schwieriger
 - ⇒ redefiniertes Verhalten in spezifische Unterklasse des Adaptees einfügen
 - ⇒ Adapter benutzt diese Unterklasse
 - ⇒ Kombinierbarkeit mit anderen Unterklassen geht verloren



- Object Adapter mit Delegation (roots.iai.uni-bonn.de/research/darwin)

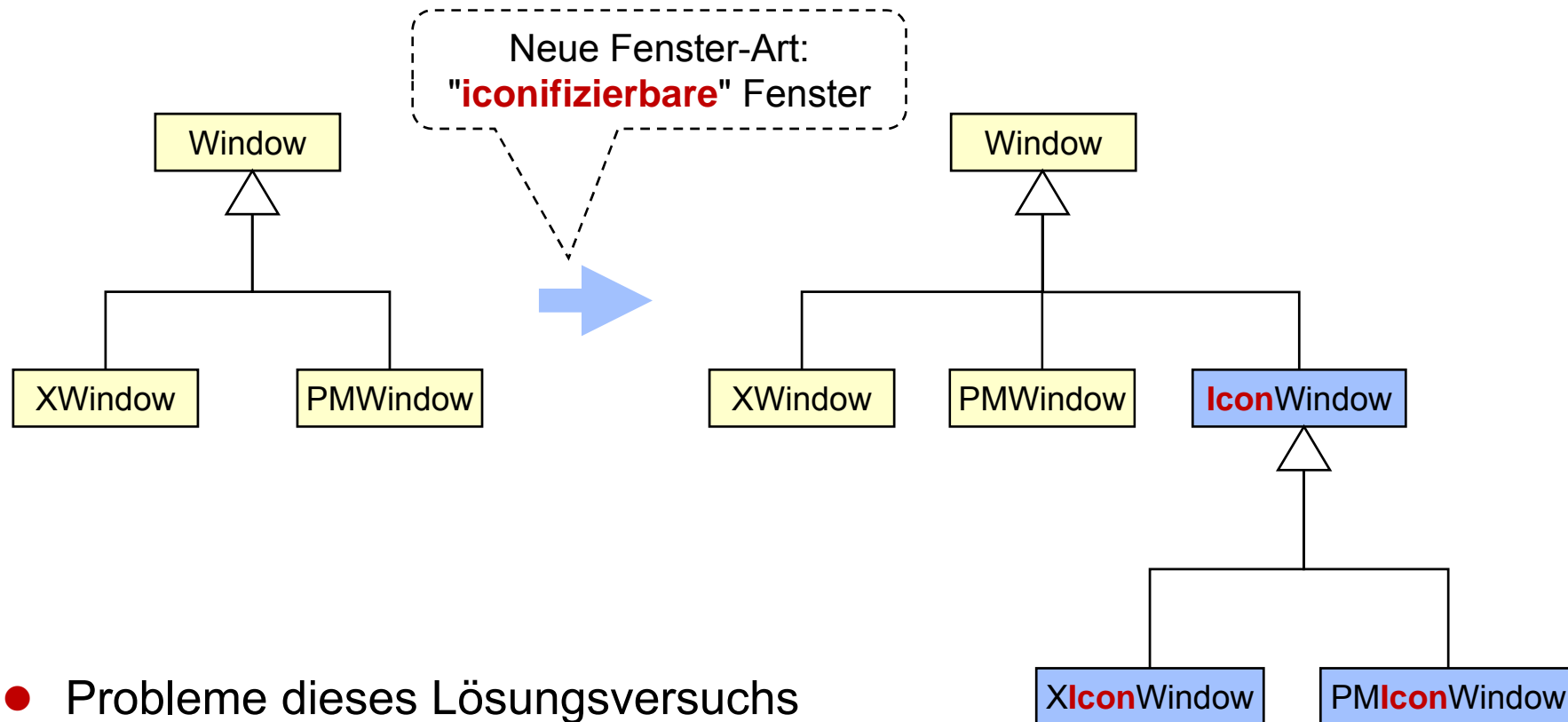
- ◆ adaptiert eine ganze Klassenhierarchie
- ◆ Overriding des Adaptee-Verhaltens leicht möglich

Das Bridge Pattern

Bridge Pattern (auch: Handle / Body)

- Absicht
 - ◆ Schnittstelle und Implementierung trennen
 - ◆ ... unabhängig variieren
- Motivation
 - ◆ portable "Window"-Abstraktion in GUI-Toolkit
 - ◆ mehrere Variations-Dimensionen
 - ⇒ Fenster-Arten:
 - normal / als Icon,
 - schließbar / nicht schließbar,
 - ...
 - ⇒ Implementierungen:
 - X-Windows,
 - IBM Presentation Manager,
 - MacOS,
 - Windows XYZ,
 - ...

Bridge Pattern: Warum nicht einfach Vererbung einsetzen?



- Probleme dieses Lösungsversuchs

- ◆ Eigene Klasse für jede Kombination Fenster-Art / Plattform

- ⇒ unwartbar

- ◆ Client wählt Kombination Fenster-Art / Plattform (bei der Objekterzeugung)

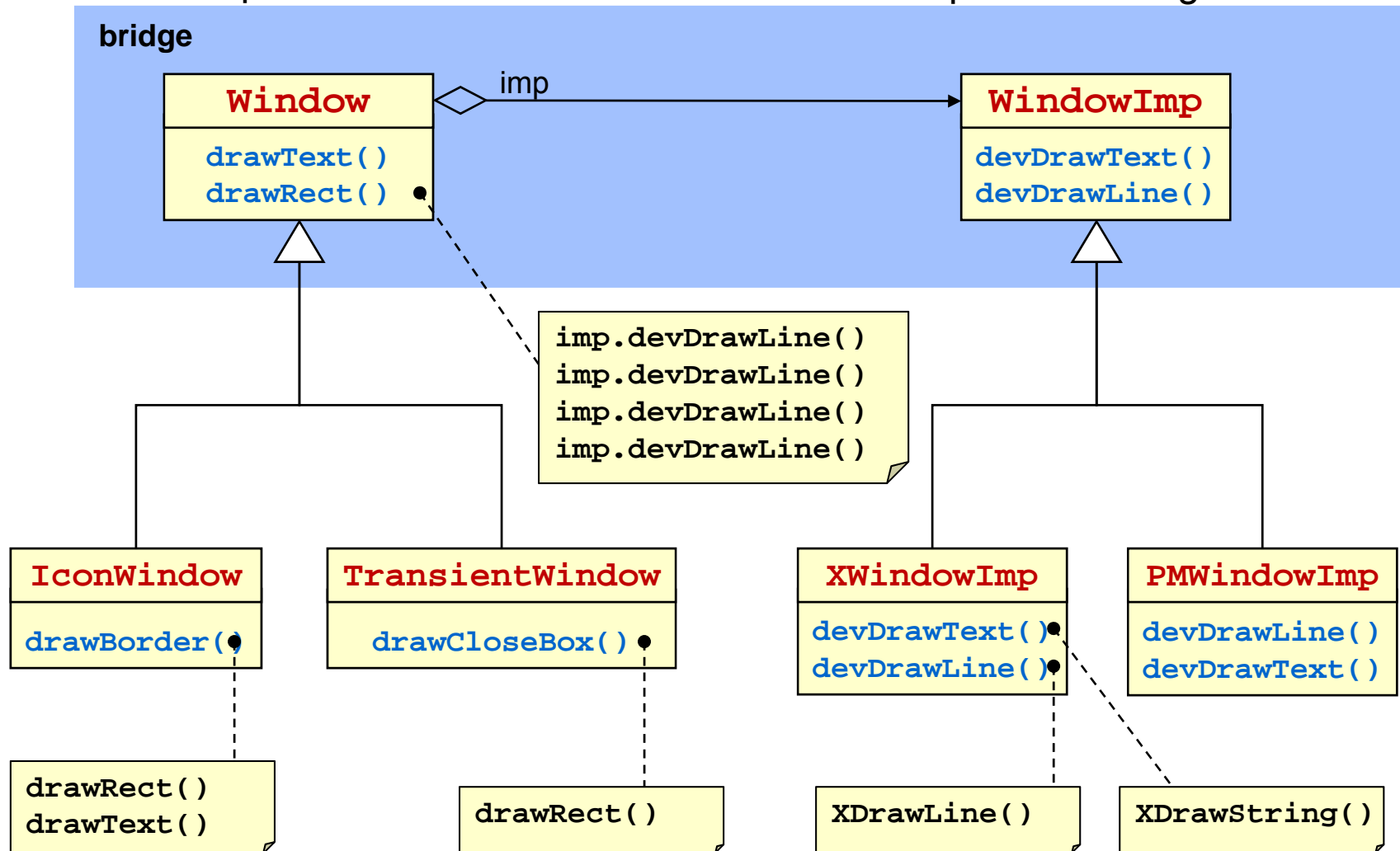
- ⇒ plattformabhängiger Client-Code

Bridge Pattern: Idee

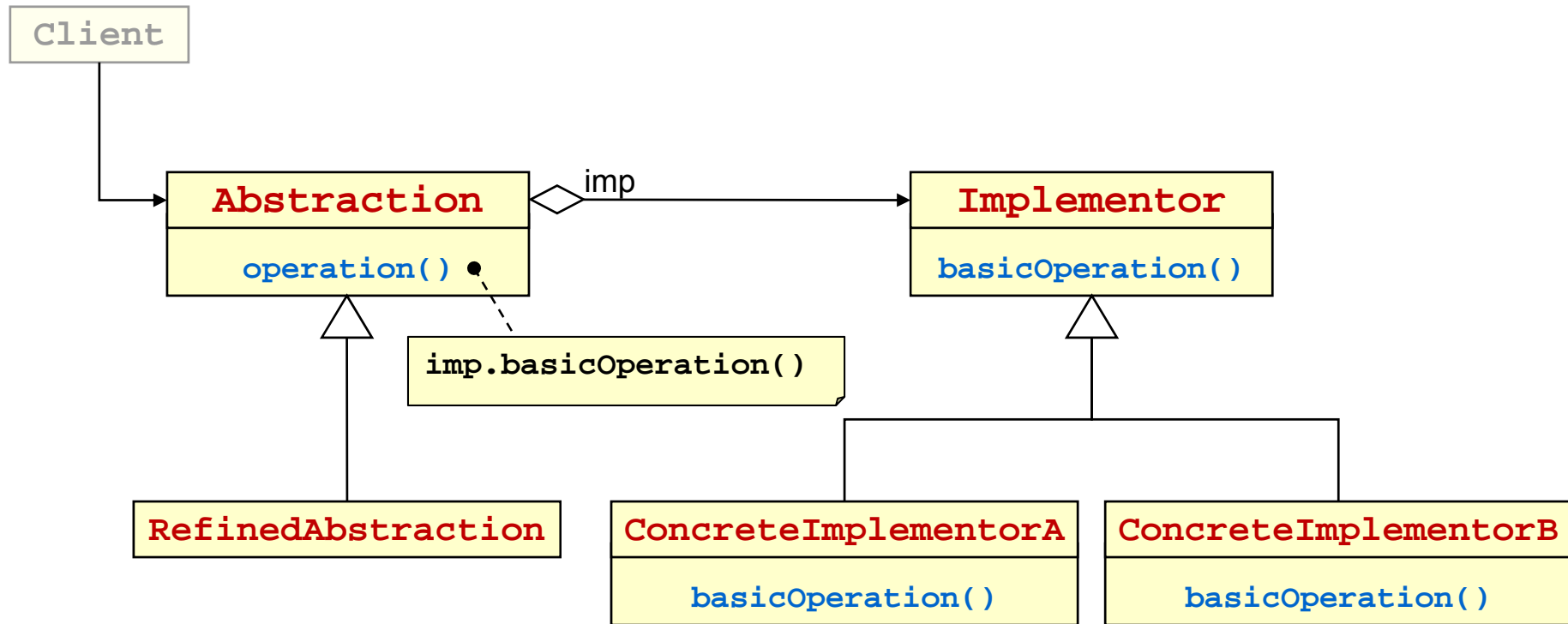
- Trennung von

◆ Konzeptueller Hierarchie

◆ Implementierungs-Hierarchie



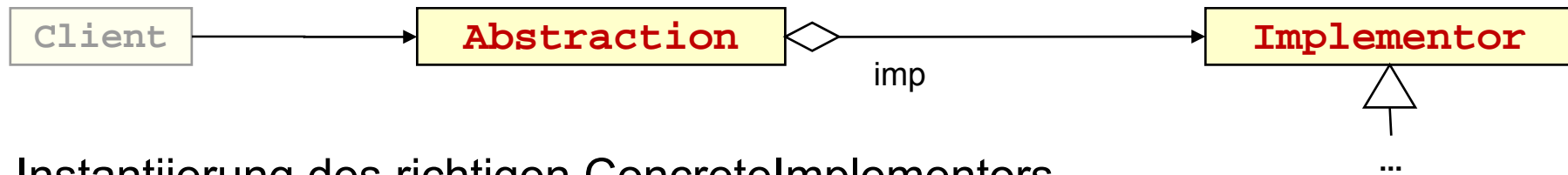
Bridge Pattern: Schema



Bridge Pattern: Anwendbarkeit

- Dynamische Änderbarkeit
 - ◆ Implementierung erst zur Laufzeit auswählen
- Unabhängige Variabilität
 - ◆ neue Unterklassen in beiden Hierarchien beliebig kombinierbar
- Implementierungs-Transparenz für Clients
 - ◆ Änderungen der Implementierung erfordern keine Änderung / Neuübersetzung der Clients
- Vermeidung von "Nested Generalisations"
 - ◆ keine Hierarchien der Art wie in der Motivations-Folie gezeigt
 - ◆ keine kombinatorische Explosion der Klassenanzahl
- Sharing
 - ◆ mehrere Clients nutzen gleiche Implementierung
 - ◆ z.B. Strings

Bridge Pattern: Implementierung



Instantiierung des richtigen ConcreteImplementors

- Falls Abstraction alle ConcreteImplementor-Klassen kennt:
 - ◆ Fallunterscheidung im Konstruktor der ConcreteAbstraction
 - ◆ Auswahl des ConcreteImplementor anhand von Parametern des Konstruktors
 - ◆ Alternativ: Default-Initialisierung und spätere situationsbedingte Änderung
- Falls Abstraction völlig unabhängig von ConcreteImplementor-Klassen sein soll:
 - ◆ Entscheidung anderem Objekt überlassen
 - ➔ Abstract Factory Pattern

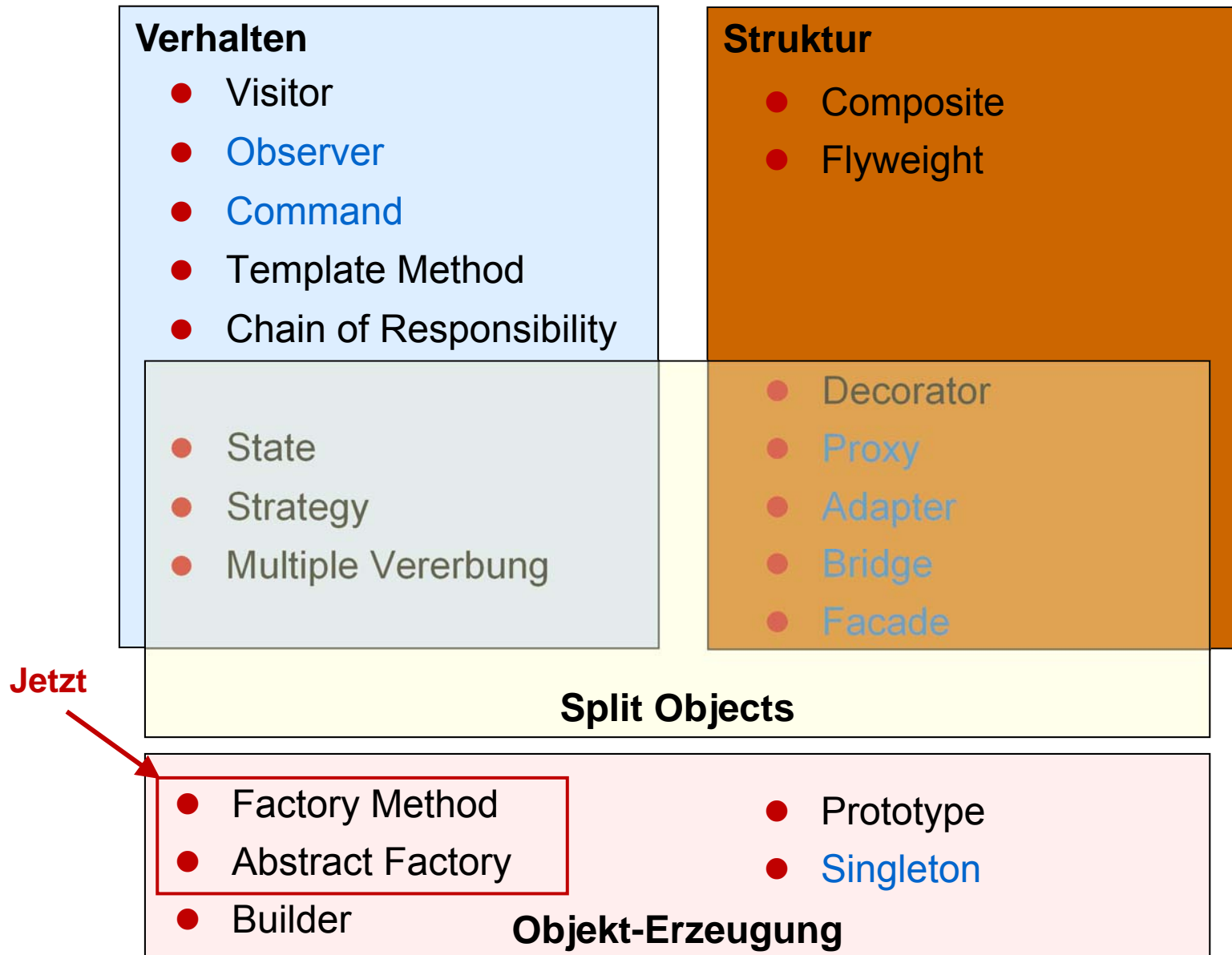
Hier geht's weiter am Do, 2.12.2010

Wichtige Entwurfsmuster, Teil 2

- Object Design Patterns -

Command
Factory Method
Abstract Factory
Composite
Visitor

"Gang of Four"-Patterns: Überblick und Klassifikation

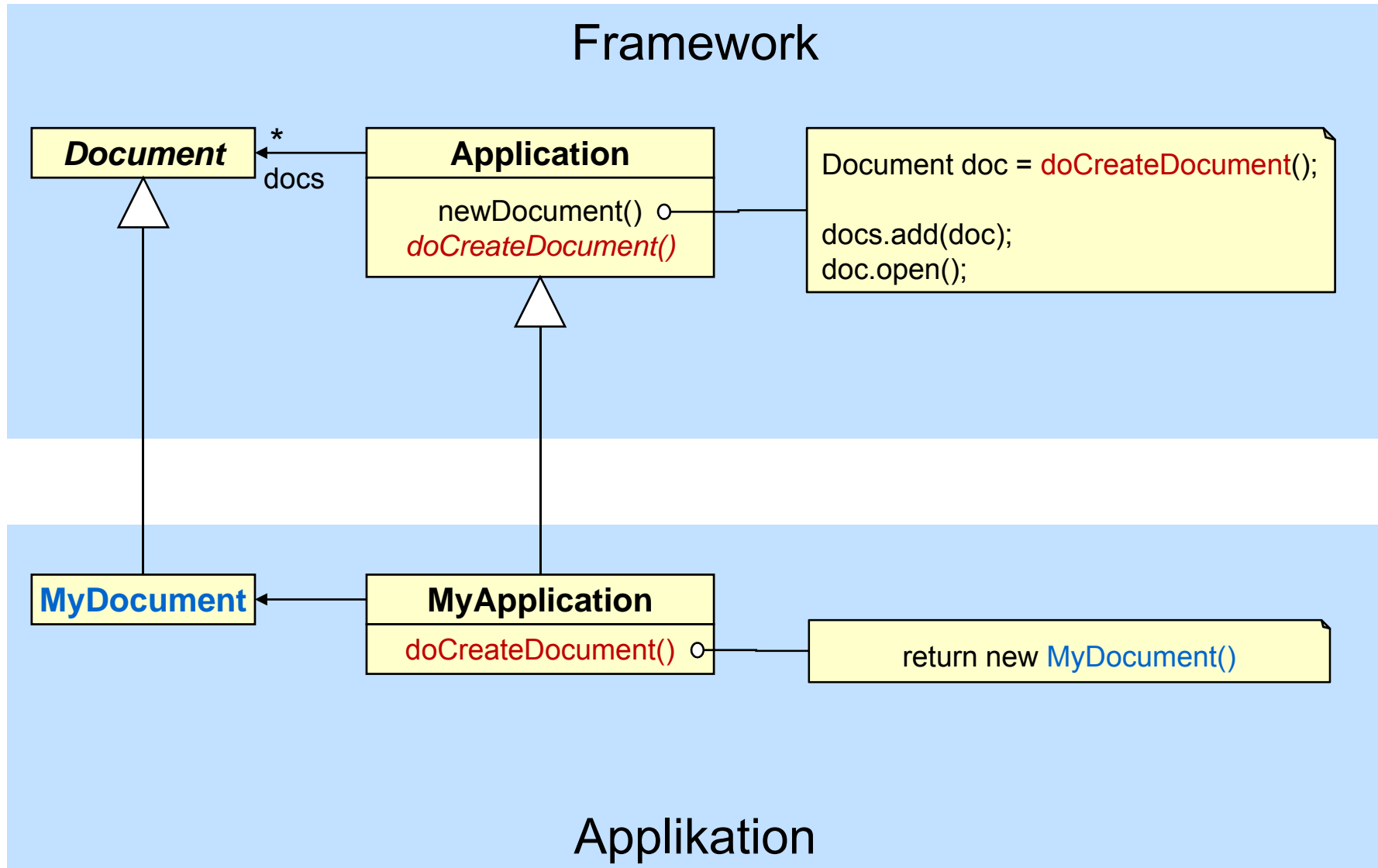


Das Factory Method Pattern

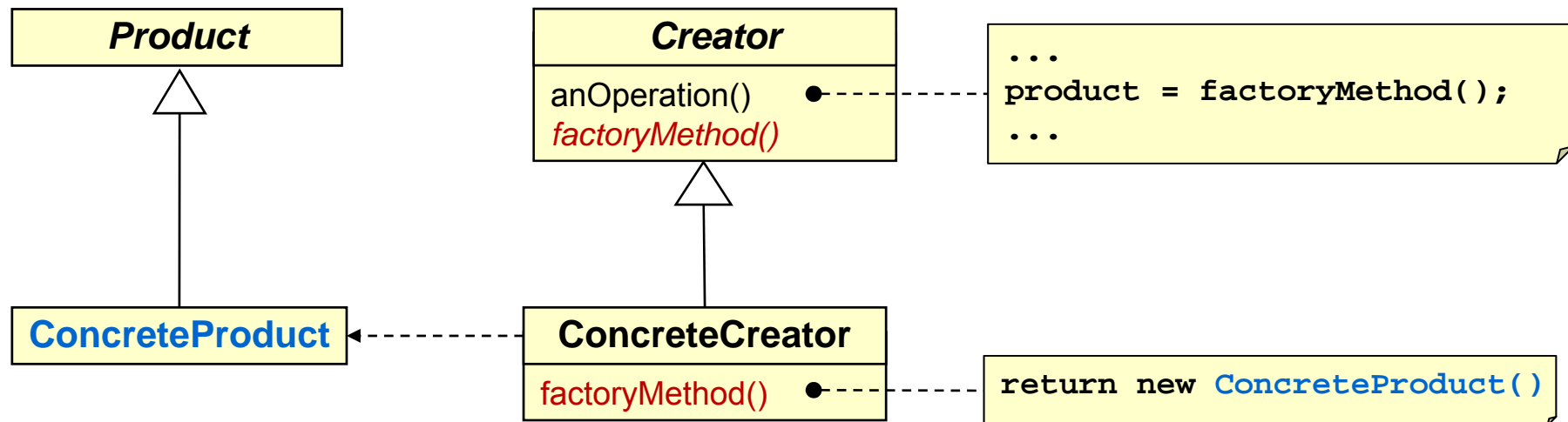
Factory Method

- Ziel
 - ◆ Entscheidung über konkreter Klasse neuer Objekte verzögern
- Motivation
 - ◆ Framework mit vordefinierten Klassen "Document" und "Application"
 - ◆ Template-Methode, für das Öffnen eines Dokuments:
 - ⇒ 1. Check ob Dokument-Art bekannt
 - ⇒ 2. Erzeugung einer Document-Instanz !?!
 - ◆ Erzeugung von Instanzen noch nicht bekannter Klassen!
- Idee
 - ◆ aktuelle Klasse entscheidet wann Objekte erzeugt werden
 - ⇒ Aufruf einer abstrakten Methode
 - ◆ Subklasse entscheidet was für Objekte erzeugt werden
 - ⇒ implementiert abstrakte Methode passend

Factory Method: Beispiel



Factory Method: Schema

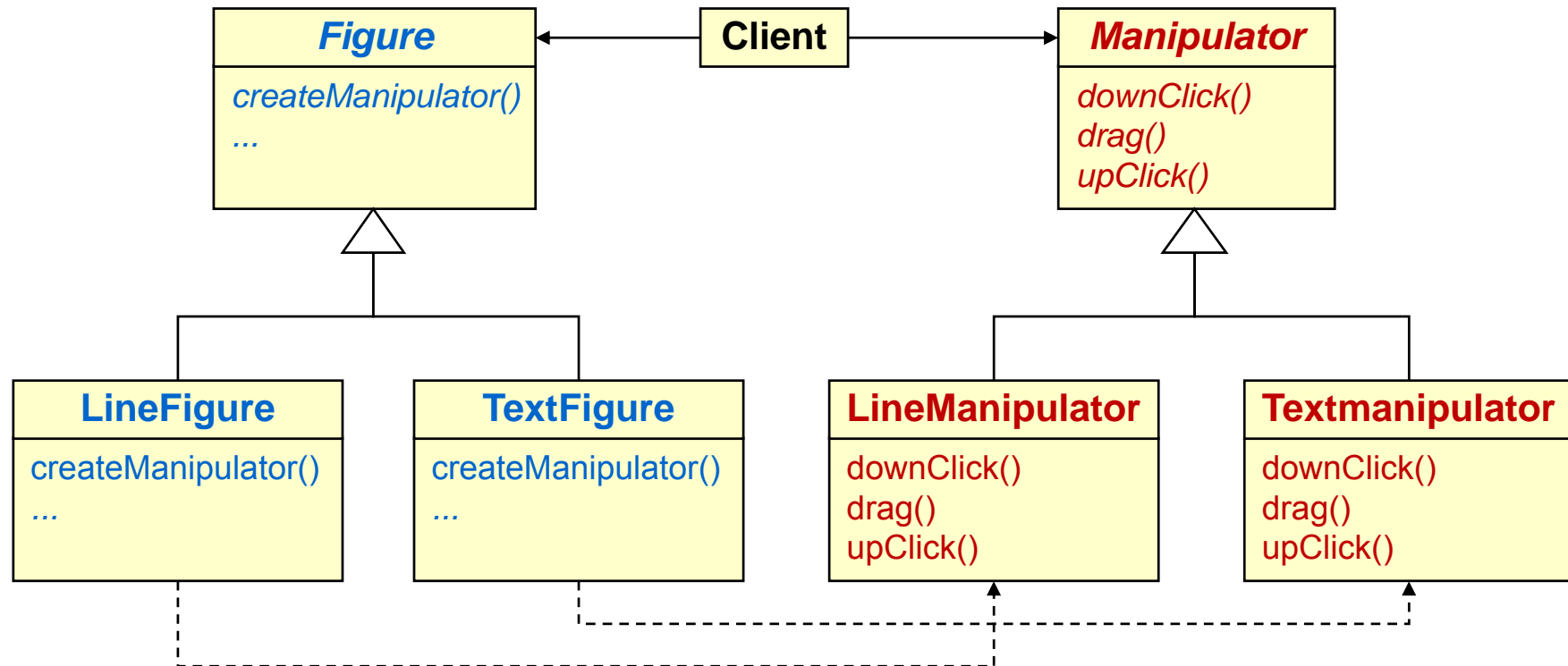


- Factory Method
 - ◆ kann abstrakt sein
 - ◆ kann Default-Implementierung haben
- Creator (Aufrufer der Factory Method)
 - ◆ kann Klasse sein, die die Factory Method definiert
 - ◆ kann Client-Klasse sein
 - ⇒ Beispiel: parallele Vererbungs-Hierarchien

Factory Method: Anwendbarkeit

- Klasse neuer Objekte unbekannt
- Verzögerte Entscheidung über Instantiierung
 - ◆ erst in Subklasse
 - ◆ erst beim Client
- Mehrere Hilfsklassen
 - ◆ Wissen über konkrete Hilfsklassen an einer Stelle lokalisieren
 - ◆ Beispiel: Parallele Hierarchien (nächste Folie)

Factory Method: Anwendbarkeit



- Verbindung paralleler Klassenhierarchien
 - ◆ Factory Method lokalisiert das Wissen über zusammengehörige Klassen
 - ◆ Restlicher Code der Figure-Hierarchie und Client-Code ist völlig unabhängig von *spezifischen* Manipulators (nur vom Manipulator-Interface)

Factory Method: Implementierung

- Fester "Produkt-Typ"
 - ◆ jede Unterklasse erzeugt festgelegte Produkt-Art
- Codierter "Produkt-Typ"
 - ◆ Parameter oder Objekt-Variable kodiert "Produkt-Typ"
 - ◆ Fallunterscheidung anhand Code
 - mehrere Produkte
 - mehr Flexibilität
- Klassen-Objekt als "Produkt-Typ"
 - ◆ Parameter oder Objekt-Variable ist "Produkt-Typ"
 - ◆ direkte Instantiierung
 - mehr Flexibilität
 - bessere Wartbarkeit
 - kein statischer Typ-Check

```
class Creator {  
    Product create() { MyProduct(); }  
}
```

```
class Creator {  
    Product create(int id) {  
        if (id==1) return MyProduct1();  
        if (id==2) return MyProduct2();  
        ...  
    }  
}
```

Idiom reflektiver Sprachen

- Smalltalk
- Java

```
class Creator {  
    Object create(Class prodType) {  
        return prodType.getInstance();  
    }  
}
```

Reflektiver Aufruf des parameterlosen Default-Konstruktors

Factory Method: Konsequenzen

- Code wird abstrakter / wiederverwendbarer
 - ◆ keine festen Abhängigkeiten von spezifischen Klassen
- Verbindung paralleler Klassenhierarchien
 - ◆ Factory Method lokalisiert das Wissen über Zusammengehörigkeiten
- evtl. künstliche Subklassen
 - ◆ nur zwecks Objekterzeugung

Factory Method: Anwendungen

- überall in Toolkits, Frameworks, Class Libraries
 - ◆ Unidraw: Beispiel "Figure und Manipulator"
 - ◆ MacApp und ET++: Beispiel "Document und Application"
- Smalltalk's Model-View-Controller Framework
 - ◆ FactoryMethoden-Template: `defaultController`
 - ◆ Hook-Methode: `defaultControllerClass`
- Orbix
 - ◆ Erzeugung des richtigen Proxy
 - ⇒ leichte Ersetzung von Standard-Proxy durch Caching Proxy

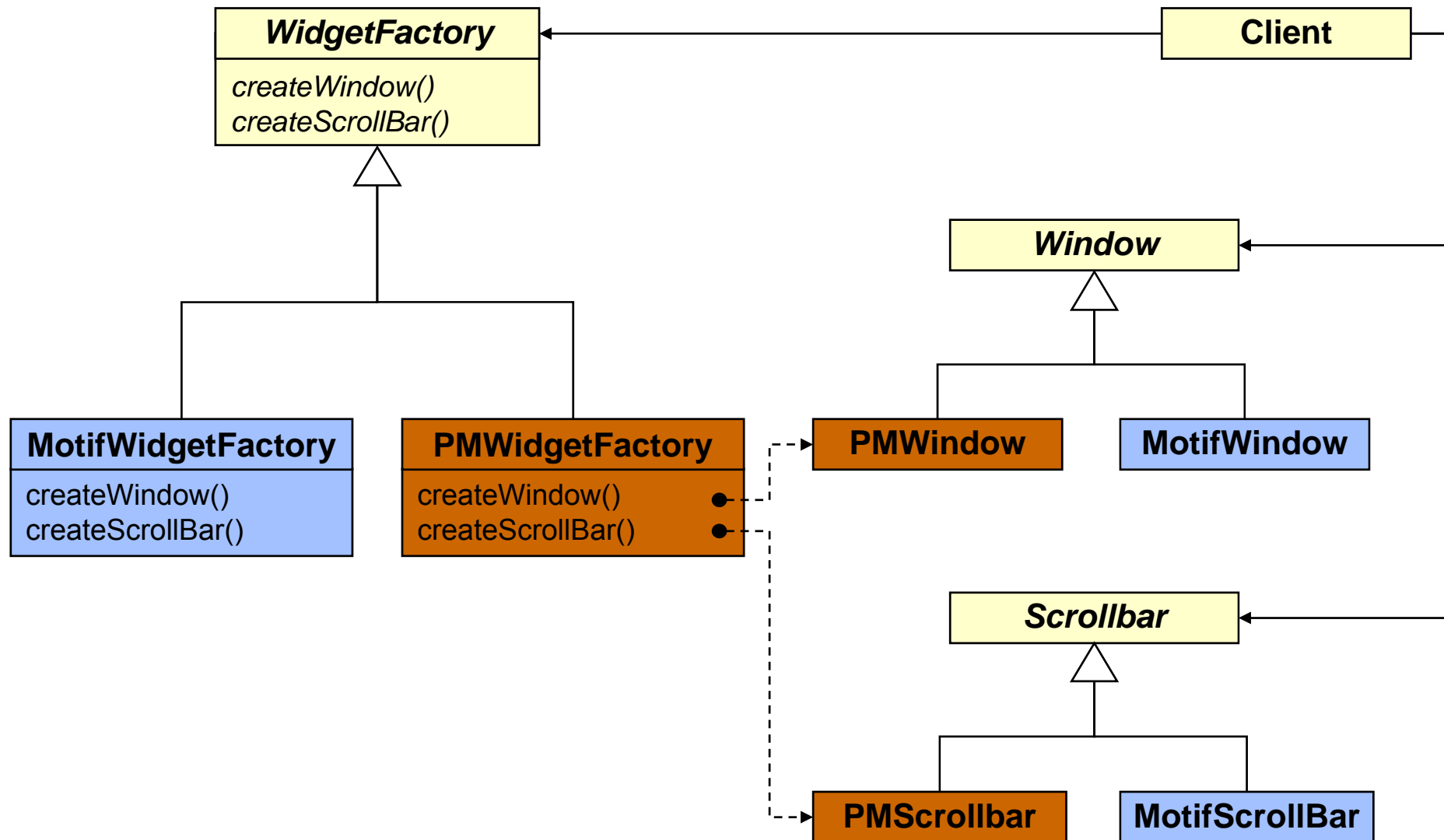
Das Abstract Factory Pattern

Abstract Factory

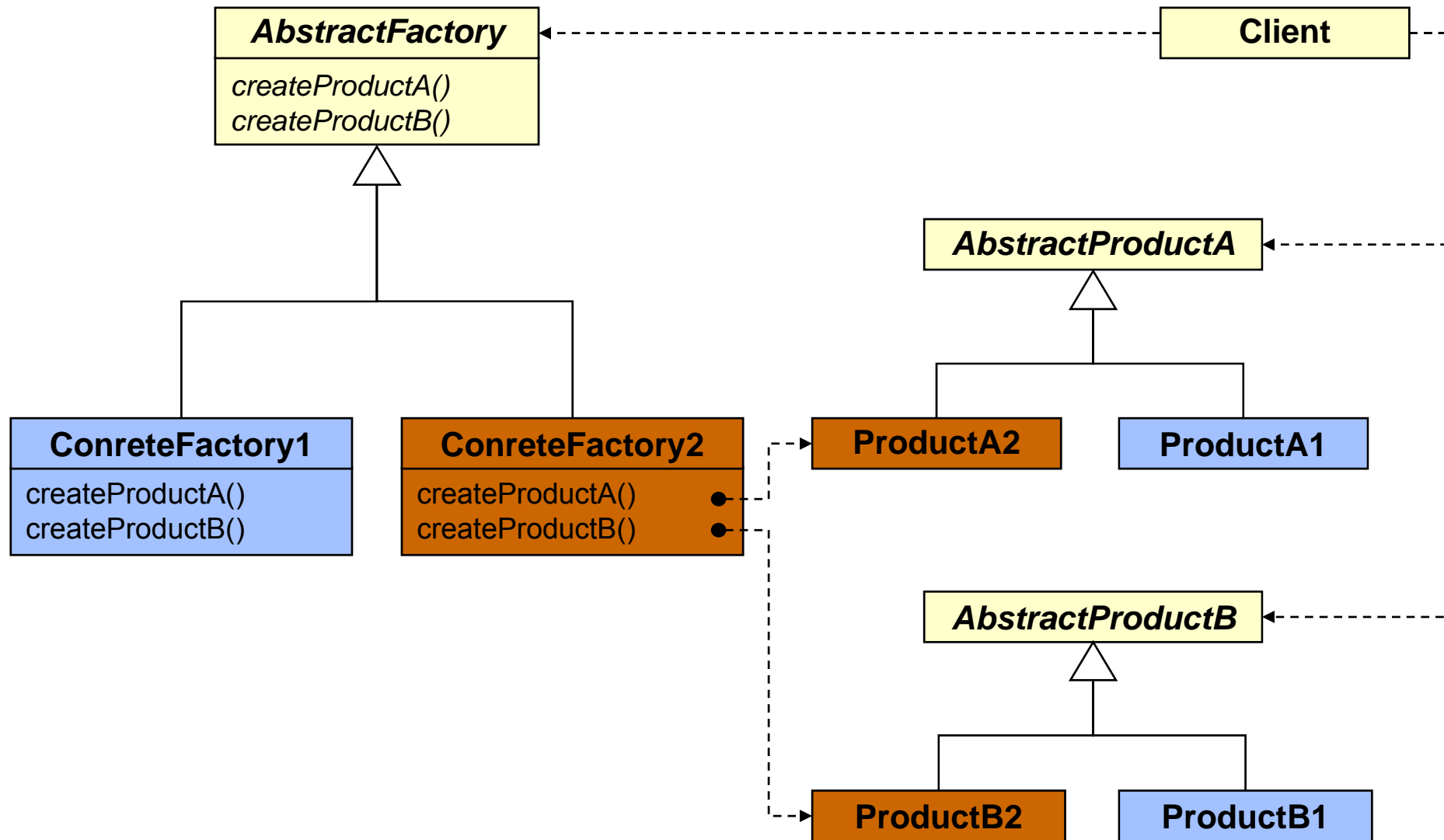
- Ziel
 - ◆ zusammengehörige Objekte verwandter Typen erzeugen
 - ◆ ... ohne deren Klassenzugehörigkeit fest zu codieren

- Motivation
 - ◆ GUI-Toolkit für mehrere Plattformen
 - ◆ Anwendungsklassen nicht von plattformspezifischen Widgets abhängig machen
 - ◆ Trotzdem soll die Anwendung
 - ⇒ alle Widgets konsistent zu einem "look and feel" erzeugen können
 - ⇒ "look and feel" umschalten können

Abstract Factory: Beispiel



Abstract Factory: Schema



Abstract Factory: Implementierung

- ConcreteFactories sind Singletons
- Produkt-Erzeugung via Factory-Methods
- fester Produkt-Typ (eine Methode für jedes Produkt)
 - ◆ Nachteil
 - ⇒ neues Produkt erfordert Änderung der gesamten Factory-Hierarchie
- Codierter "Produkt-Typ" (eine parametrisierte Methode für alle Produkte)
 - ◆ Vorteil
 - ⇒ leichte Erweiterbarkeit um neues Produkt
 - ◆ Nachteile:
 - ⇒ alle Produkte müssen gemeinsamen Obertyp haben
 - ⇒ Clients können nur die Operationen des gemeinsamen Obertyps nutzen
 - ⇒ Verzicht auf statische Typsicherheit
- Klassen-Objekt als "Produkt-Typ" (eine parametrisierte Methode)
 - ◆ Vorteil
 - ⇒ neues Produkt erfordert keinerlei Änderungen der Factory
 - ◆ Nachteil
 - ⇒ Verzicht auf jegliche statische Typinformation / Typsicherheit

Abstract Factory: Implementierung

- Produktfamilie = Subklasse
 - ◆ Vorteil
 - ⇒ Konsistenz der Produkte
 - ◆ Nachteil
 - ⇒ neue Produktfamilie erfordert neue Subklasse, auch bei geringen Unterschieden

- Produktfamilie = von Clients konfigurierte assoziative Datenstruktur
 - ◆ Varianten
 - ⇒ Prototypen und Cloning
 - ⇒ Klassen und Instantiierung
 - ◆ Vorteil
 - ⇒ keine Subklassenbildung erforderlich
 - ◆ Nachteil
 - ⇒ Verantwortlichkeit an Clients abgegeben
 - ⇒ konsistente Produktfamilien können nicht mehr garantiert werden

Abstract Factory: Konsequenzen

- Abschirmung von Implementierungs-Klassen
 - ◆ Namen von Implementierungsklassen erscheinen nicht im Code von Clients
 - ◆ Clients benutzen nur abstraktes Interface
- Leichte Austauschbarkeit von Produktfamilien
 - ◆ Name einer ConcreteFactory erscheint nur ein mal im Code
 - ⇒ bei ihrer Instantiierung
 - ◆ Leicht austauschbar gegen andere ConcreteFactory
 - ◆ Beispiel: Dynamisches Ändern des look-and-feel
 - ⇒ andere ConcreteFactory instantiieren
 - ⇒ alle GUI-Objekte neu erzeugen
- Konsistente Benutzung von Produktfamilien
 - ◆ Keine Motif-Scrollbar in Macintosh-Fenster
- Schwierige Erweiterung um neue Produktarten
 - ◆ Schnittstelle der AbstractFactory legt Produktarten fest
 - ◆ Neue Produktart = Änderung der gesamten Factory-Hierarchie

Abstract Factory: Anwendbarkeit

- System soll unabhängig sein von
 - ◆ Objekt-Erzeugung
 - ◆ Objekt-Komposition
 - ◆ Objekt-Darstellung
- System soll konfigurierbar sein
 - ◆ Auswahl aus verschiedenen Produkt-Familien
- konsistente Produktfamilien
 - ◆ nur Objekte der gleichen Familie "passen zusammen"
- Library mit Produktfamilie anbieten
 - ◆ Clients sollen nur Schnittstelle kennen
 - ◆ ... nicht die genauen Teilprodukt-Arten / -Implementierungen

"Gang of Four"-Patterns: Überblick und Klassifikation

Jetzt

Verhalten

- Visitor
- Observer
- Command
- Template Method
- Chain of Responsibility

- State
- Strategy
- Multiple Vererbung

Struktur

- Composite
- Flyweight

- Decorator
- Proxy
- Adapter
- Bridge
- Facade

Split Objects

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Objekt-Erzeugung

Das Command Pattern

Das Command Pattern: Motivation

- Kontext

- ◆ GUI-Toolkits mit Buttons, Menüs, etc.

- Forces

- ◆ Vielfalt trotz Einheitlichkeit

- ⇒ Einheitlicher Code in allen GUI-Elementen
- ⇒ .. trotzdem verschiedene Effekte

- ◆ Wiederverwendung

- ⇒ Über verschiedene GUI-Elemente ansprechbare Operationen nur ein mal programmieren

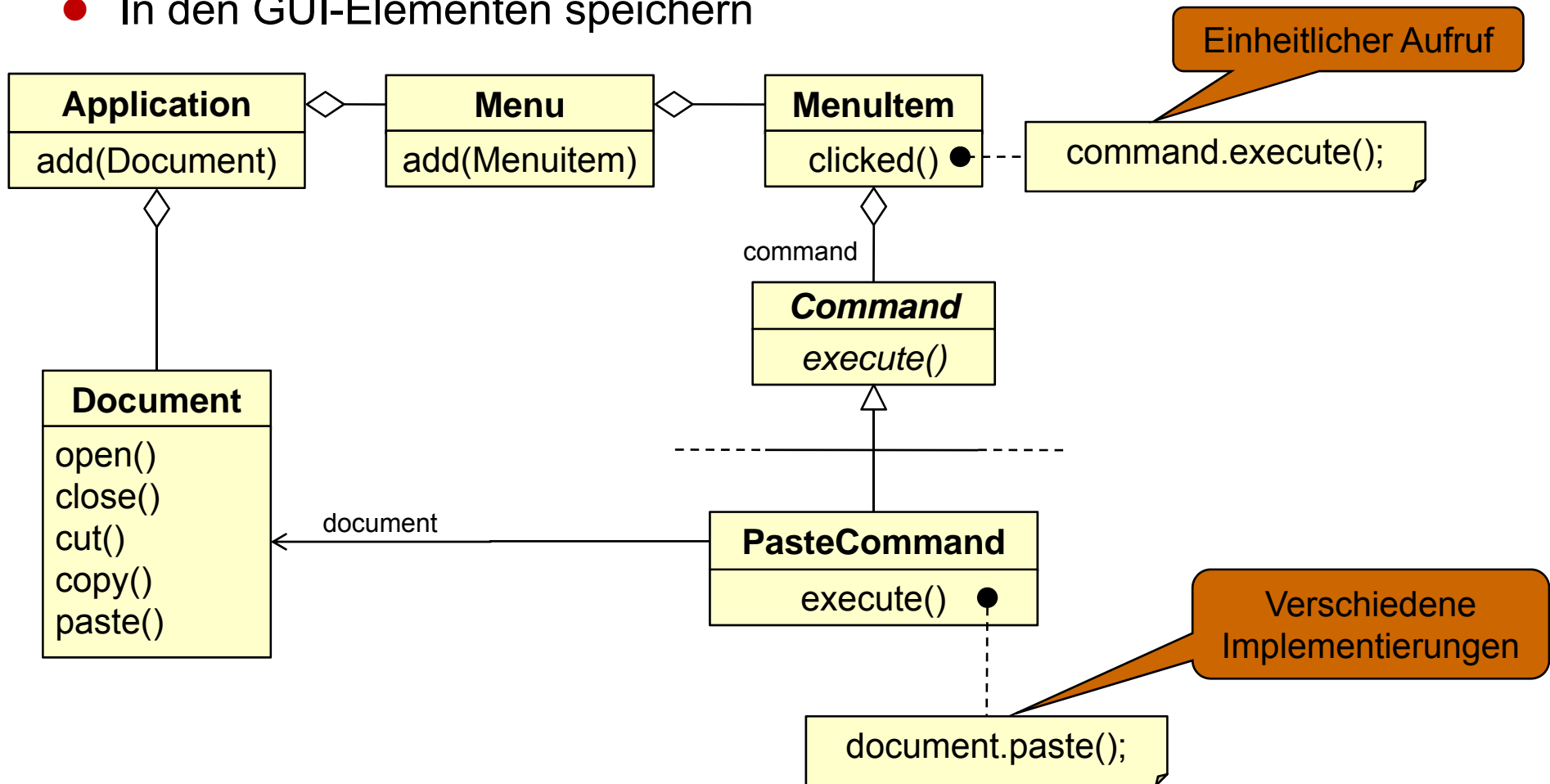
- ◆ Dynamik

- ⇒ dynamische Änderung der Aktionen von Menü-Einträgen
- ⇒ kontext-sensitive Aktionen

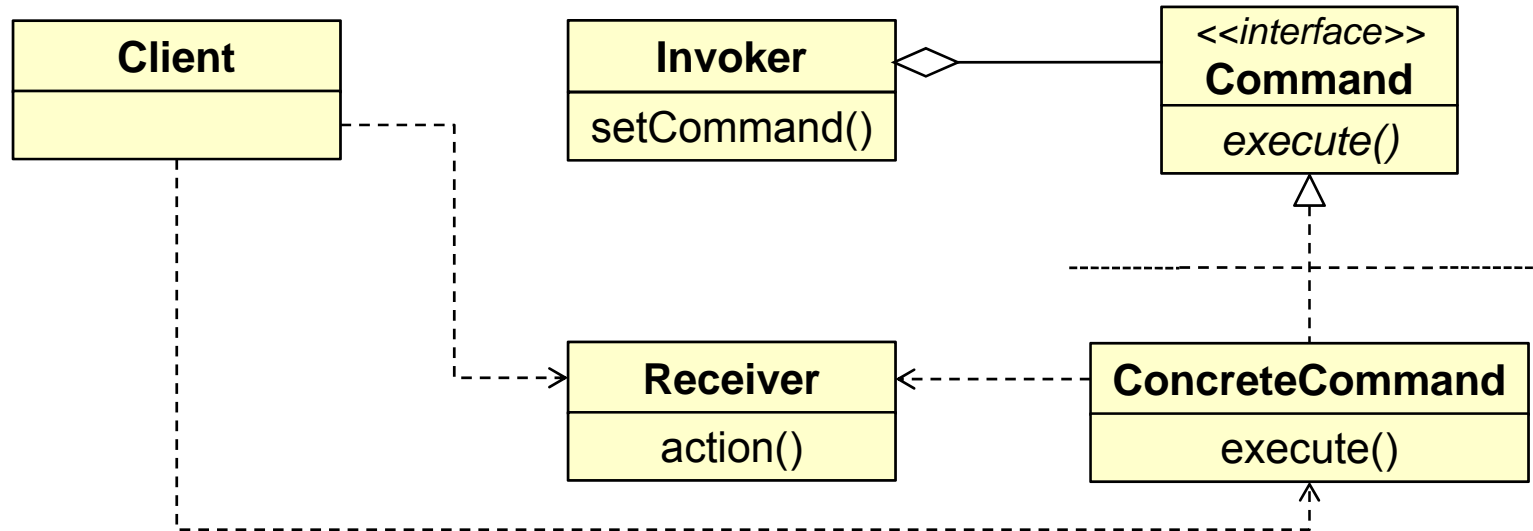


Das Command Pattern: Idee

- Operationen als Objekte mit Methode execute() darstellen
- In den GUI-Elementen speichern



Command Pattern: Schema und Rollen



- Command Interface
 - ◆ Schnittstelle, die festlegt, was Commands tun können
 - ◆ Mindestens Execute, optional auch Undo, Redo
- Concrete Command
 - ◆ Implementiert Command Interface
 - ◆ „Controllers“ sind oft „ConcreteCommands“
- Execute-Methode
 - ◆ Ausführen von Kommandos
- Undo-Methode
 - ◆ Rückgängig machen von Kommandos
- Redo-Methode
 - ◆ Rückgängig gemachtes Kommando wieder ausführen

Das Command Pattern: Konsequenzen

- Entkopplung
 - ◆ von Aufruf einer Operation und Spezifikation einer Operation.
- Kommandos als Objekte
 - ◆ Command-Objekte können wie andere Objekte auch manipuliert und erweitert werden.
- Komposition
 - ◆ Folgen von Command-Objekte können zu weiteren Command-Objekten zusammengefasst werden.
- Erweiterbarkeit
 - ◆ Neue Command-Objekte können leicht hinzugefügt werden, da keine Klassen geändert werden müssen.

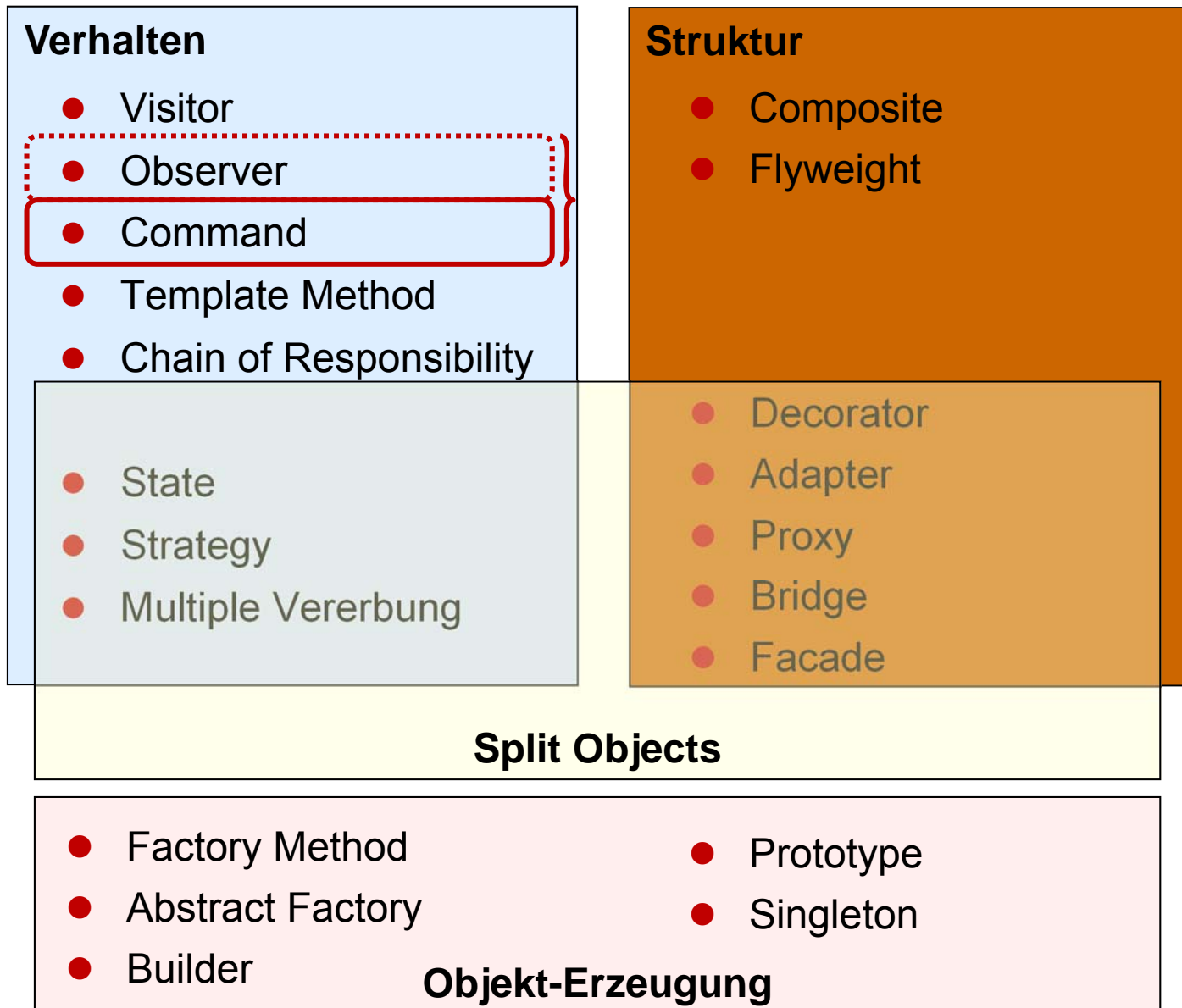
Das Command Pattern: Anwendbarkeit

- Operationen als Parameter
- Variable Aktionen
 - ◆ referenziertes Command-Objekt austauschen
- Zeitliche Trennung
 - ◆ Befehl zur Ausführung, Protokollierung, Ausführung
- Speicherung und Protokollierung von Operationen
 - ◆ History-Liste
 - ◆ Serialisierung
- "Undo" von Operationen
 - ◆ Command-Objekte enthalten neben `execute()` auch `unexecute()`
 - ◆ werden in einer History-Liste gespeichert
- Recover-Funktion nach Systemabstürzen
 - ◆ History-Liste wird gespeichert
 - ◆ Zustand eines Systems ab letzter Speicherung wiederstellbar
- Unterstützung von Transaktionen
 - ◆ Transaktionen können sowohl einfache ("primitive"), als auch komplexe Command-Objekte sein.

Implementierung des Command Patterns

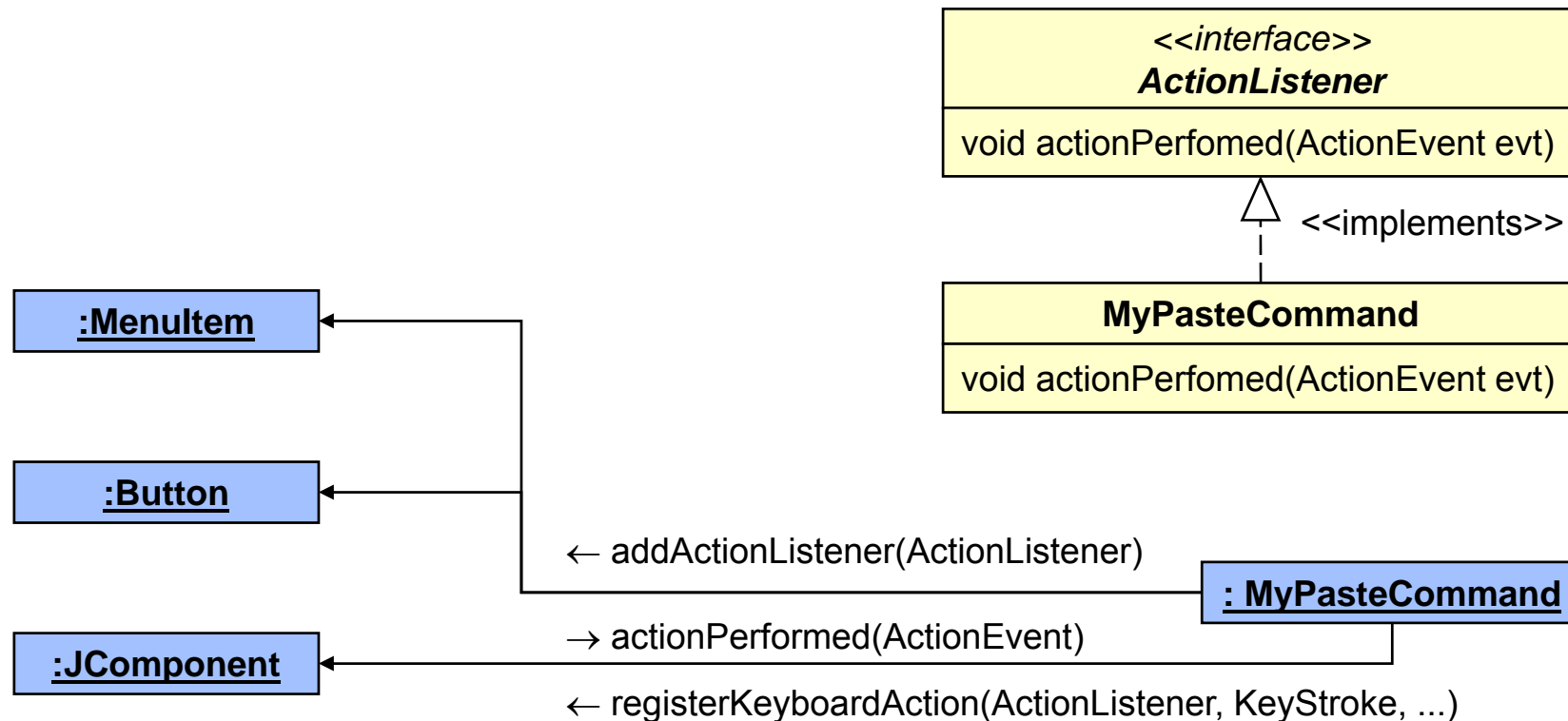
- Unterschiedliche Grade von Intelligenz
 - ◆ Command-Objekte können "nur" Verbindung zwischen Sender und Empfänger sein, oder aber alles selbstständig erledigen.
- Unterstützung von "undo"
 - ◆ Semantik
 - ⇒ Undo (unexecute()) und redo (execute()) müssen den exakt gegenteiligen Effekt haben!
 - ◆ Problem: In den wenigsten Fällen gibt es exact inverse Operationen
 - ⇒ Die Mathematik ist da eine Ausnahme...
 - ◆ Daher Zusatzinfos erforderlich
 - ⇒ Damit ein Command-Objekt "undo" unterstützen kann, müssen evtl. zusätzliche Informationen gespeichert werden.
 - ⇒ Typischerweise: Kopien des alten Zustands der Objekte die wiederhergestellt werden sollen.
 - ◆ History-Liste
 - ⇒ Für mehrere Levels von undo/redo
 - ◆ Klonen vor Speicherung in der History-Liste
 - ⇒ Es reicht nicht eine Referenz auf die Objekte zu setzen!
 - ⇒ Man muss das Objekt "tief" klonen, um sicherzustellen dass sein Zustand nicht verändert wird.

Patterns-Überblick



Verbindung von Command und Observer im JDK

Verbindung von Observer und Command in Java



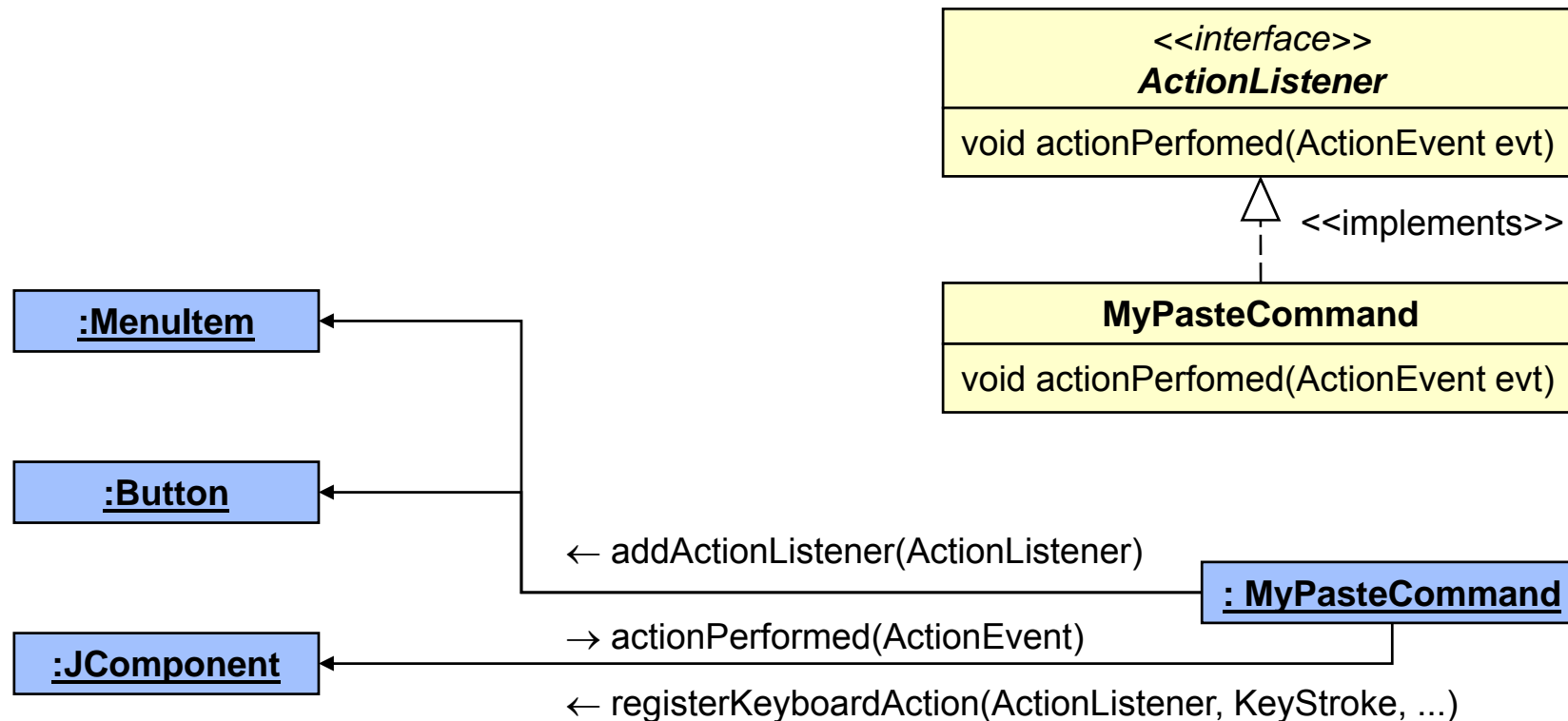
● Observer

- ◆ Buttons, Menue-Einträge und Tasten generieren "ActionEvents"
- ◆ Interface "ActionListener" ist vordefiniert
- ➔ "ActionListener" implementieren
- ➔ ... und Instanzen davon bei Buttons, Menultems, etc registrieren

● Command & Observer

- ◆ gleiche Methode (z.B. `actionPerformed`) spielt die Rolle der `run()` Methode eines Commands und die Rolle der `update()` Methode eines Observers
- ◆ implementierende Klasse (z.B. `MyPasteCommand`) ist gleichzeitig ein Command und ein Observer

Verbindung von Observer und Command in Java verbindet auch Push und Pull



- Push Observer

- ◆ Subject „pushed“ eine „Event“-Instanz
- ◆ Der „Event“-Typ kann selbst definiert werden
- ◆ In dieser Instanz können somit weitere Informationen mit „gepushed“ werden (als Werte von Instanzvariablen)

- Pull Observer

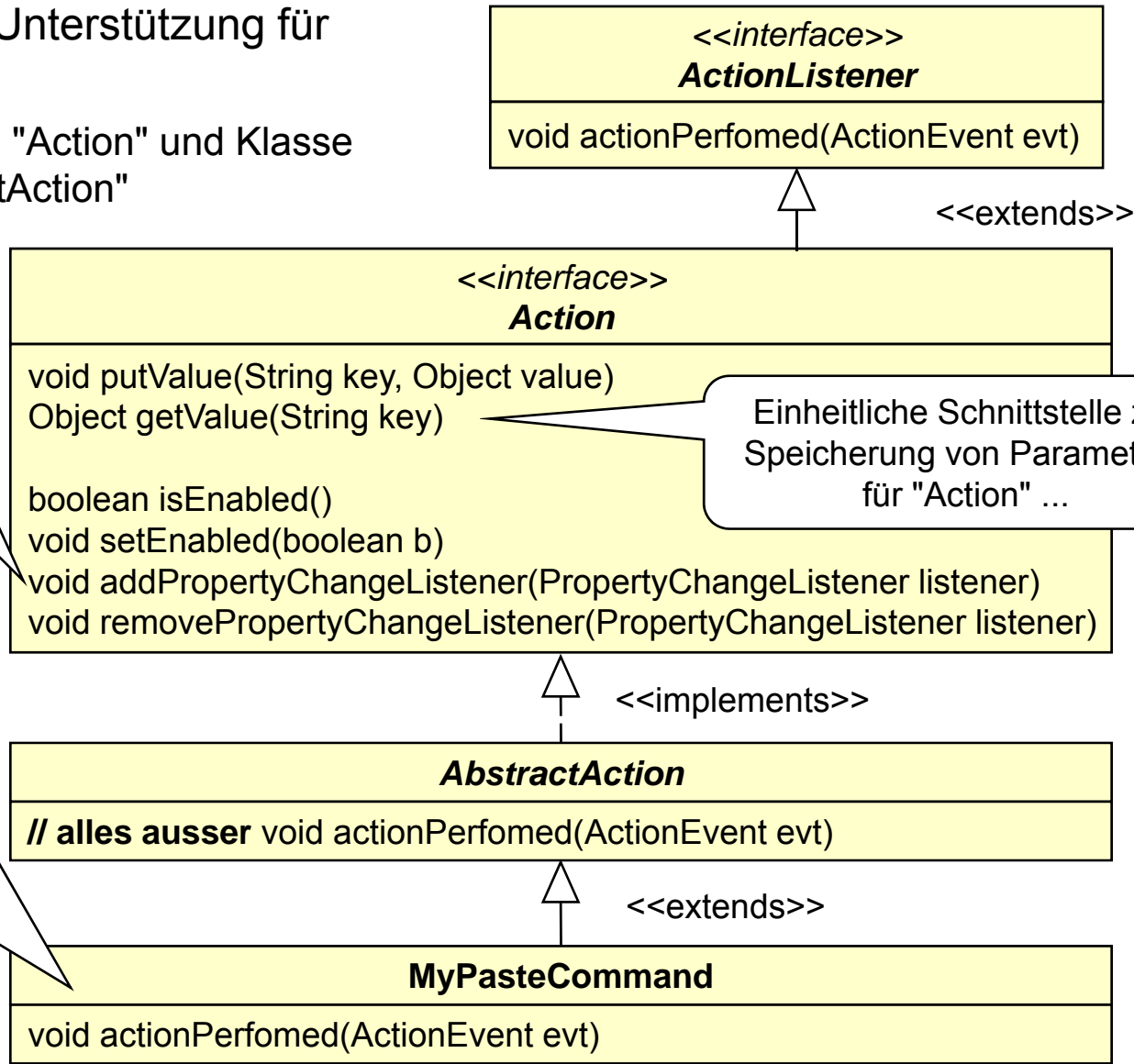
- ◆ Die als Parameter übergebene „Event“-Instanz enthält immer eine Referenz auf die Quelle des Events
- ◆ Somit ist es möglich von dort weitere Informationen zu anzufragen („pull“)

Verbindung von Observer und Command in Java (2)

- Zusätzliche Unterstützung für "Command"
 - ◆ Interface "Action" und Klasse "AbstractAction"

Aktivierung / Deaktivierung von Menüitems denen diese "Action" zugeordnet ist

... Parameter können allerdings auch direkt als Instanz-Variablen realisiert werden.



Einheitliche Schnittstelle zur Speicherung von Parametern für "Action" ...

im JDK enthalten

Beispiel: Änderung der Hintergrundfarbe (1)

```
class ColorAction extends AbstractAction
{   public ColorAction(..., Color c, Component comp)
    {   ...
        color = c;
        target = comp;
    }

    public void actionPerformed(ActionEvent evt)
    {   target.setBackground(color);
        target.repaint();
    }

    private Component target;
    private Color color;
}
```

```
class ActionButton extends JButton
{   public ActionButton(Action a)
    {   ...
        addActionListener(a);
    }
}
```

- ColorAction
 - ◆ Ändern der Hintergrundfarbe einer GUI-Komponente
- ActionButton
 - ◆ Buttons die sofort bei Erzeugung mit einer Action verknüpft werden

Beispiel: Änderung der Hintergrundfarbe (2)

- Nutzung der ColorAction
 - ◆ Erzeugung einer Instanz
 - ◆ Registrierung

```
class SeparateGUIFrame extends JFrame
{ public SeparateGUIFrame()
  { ...
    JPanel panel = new JPanel();

    Action blueAction = new ColorAction("Blue", ..., ..., panel);

    panel.registerKeyboardAction(blueAction, ...);

    panel.add(new JButton(blueAction));

    JMenu menu = new JMenu("Color");
    menu.add(blueAction);
    ...
  }
}
```

Beispiel und Erläuterung siehe: "Core Java 2", Kapitel 8,
Abschnitt "Separating GUI and Application Code".

Fazit

- ✓ Elegante Verbindung von Observer und Command
 - ◆ Commands sind `ActionListener` von Buttons, Menus, etc.
 - ⇒ Einheitlicher Aufruf via `actionPerformed(ActionEvent evt)`
 - ◆ Buttons und Menus sind `PropertyChangeListener` von Commands
 - ⇒ Aktivierung / Deaktivierung

- ✓ Wiederverwendung
 - ◆ gleiche `Action` für Menu, Button, Key

- Dynamik
 - ◆ Wie ändert man die aktuelle Aktion?
 - ◆ ... konsistent für alle betroffenen Menüitems, Buttons, etc.???

- ➔ Strategy Pattern!

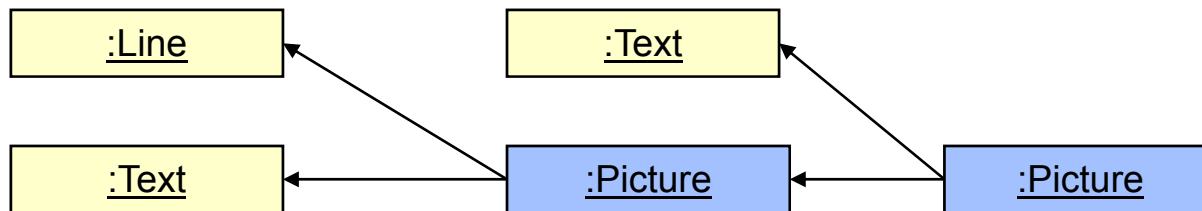
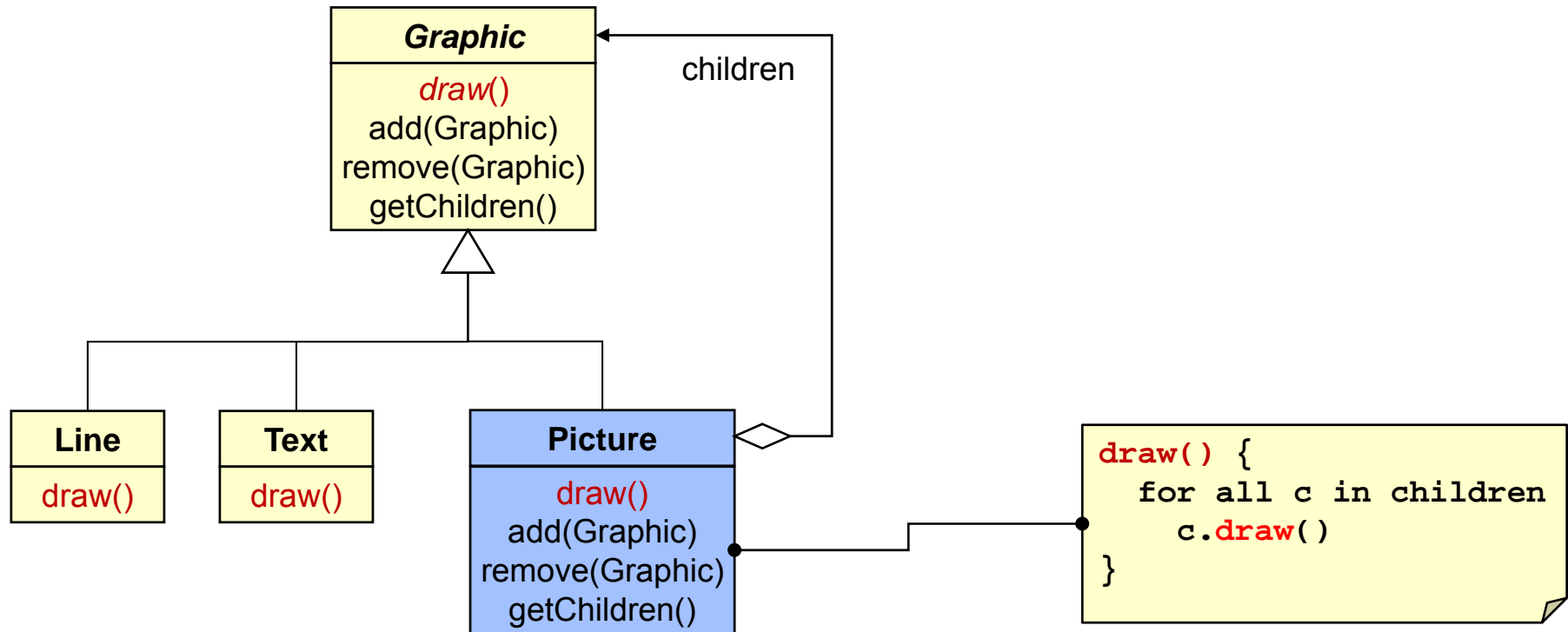
Das Composite Pattern

Composite Pattern

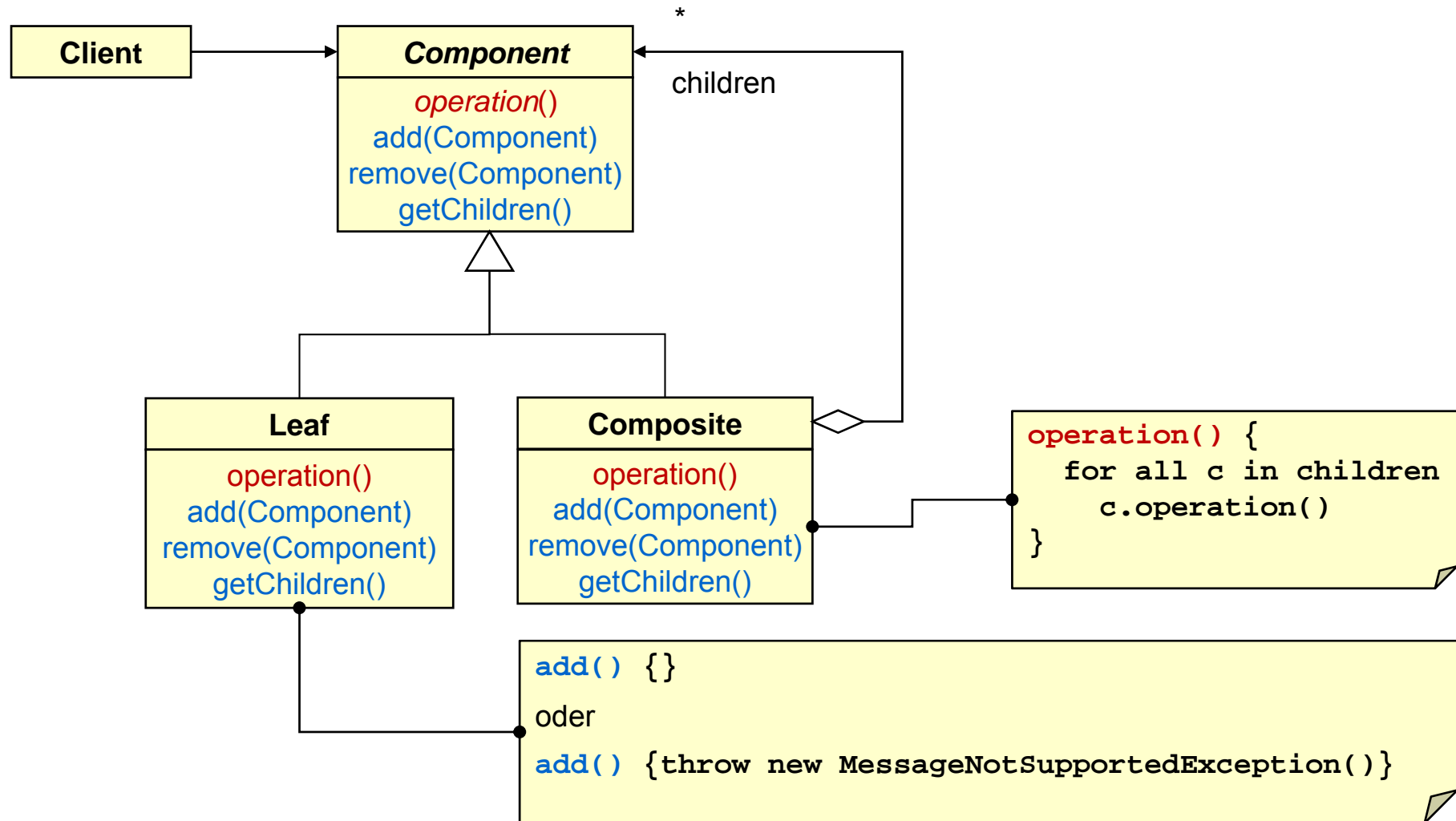
- Ziel
 - ◆ rekursive Aggregations-Strukturen darstellen ("ist Teil von")
 - ◆ Aggregat und Teile einheitlich behandeln

- Motivation
 - ◆ Gruppierung von Graphiken

Composite Pattern: Beispiel



Composite Pattern: Struktur

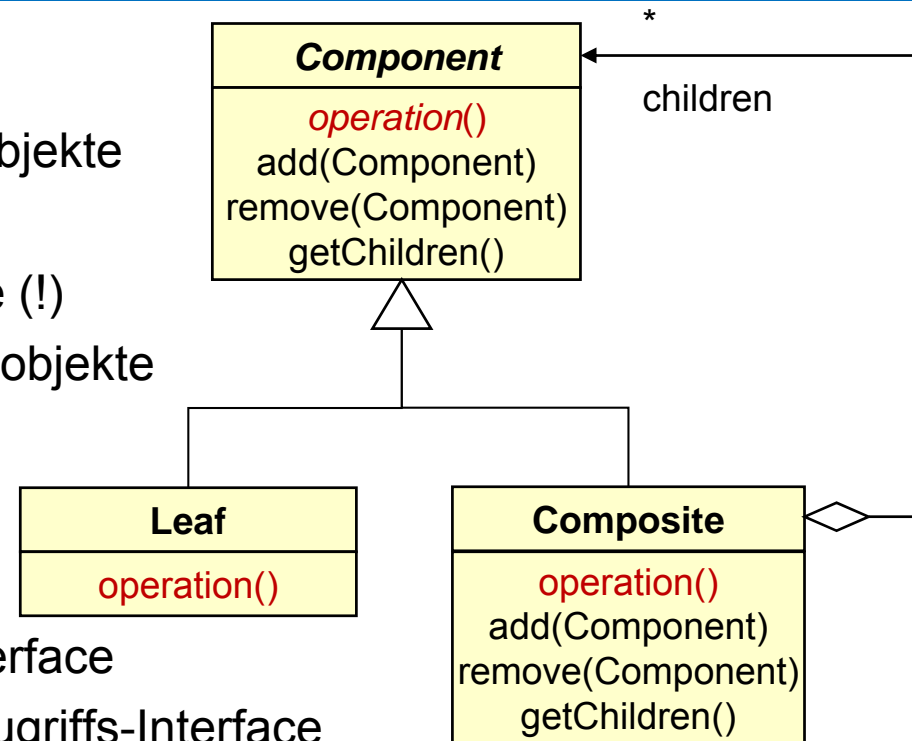


Composite Pattern: Verantwortlichkeiten

- Component (Graphic)
 - ◆ gemeinsames Interface aller Teilobjekte
 - ◆ default-Verhalten
 - ◆ Interface für Zugriff auf Teilobjekte (!)
 - ◆ evtl. Interface für Zugriff auf Elternobjekte

- Leaf (Rectangle, Line, Text)
 - ◆ "primitive" Teil-Objekte
 - ◆ implementieren gemeinsames Interface
 - ◆ leere Implementierungen für Teilzugriffs-Interface

- Composite (Picture)
 - ◆ speichert Teilobjekte
 - ◆ implementiert gemeinsames Interface durch forwarding
 - ◆ implementiert Teilzugriffs-Interface



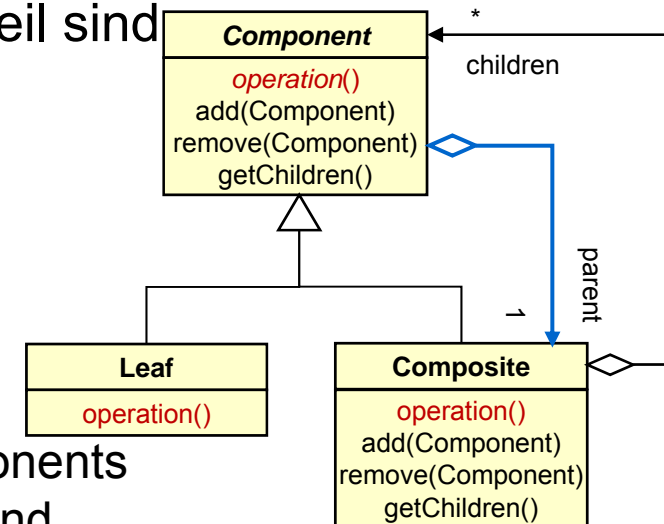
Composite Pattern: Implementierung

- Wenn Composites wissen sollen wovon sie Teil sind

- ◆ Explizite Referenzen auf Composite in Component Klasse

- Wenn mehrere Composites Components gemeinsam nutzen sollen

- ◆ Schließt explizite Referenz der Components auf Composite aus oder erfordert, dass Components wissen, dass sie Teile mehrerer Composites sind



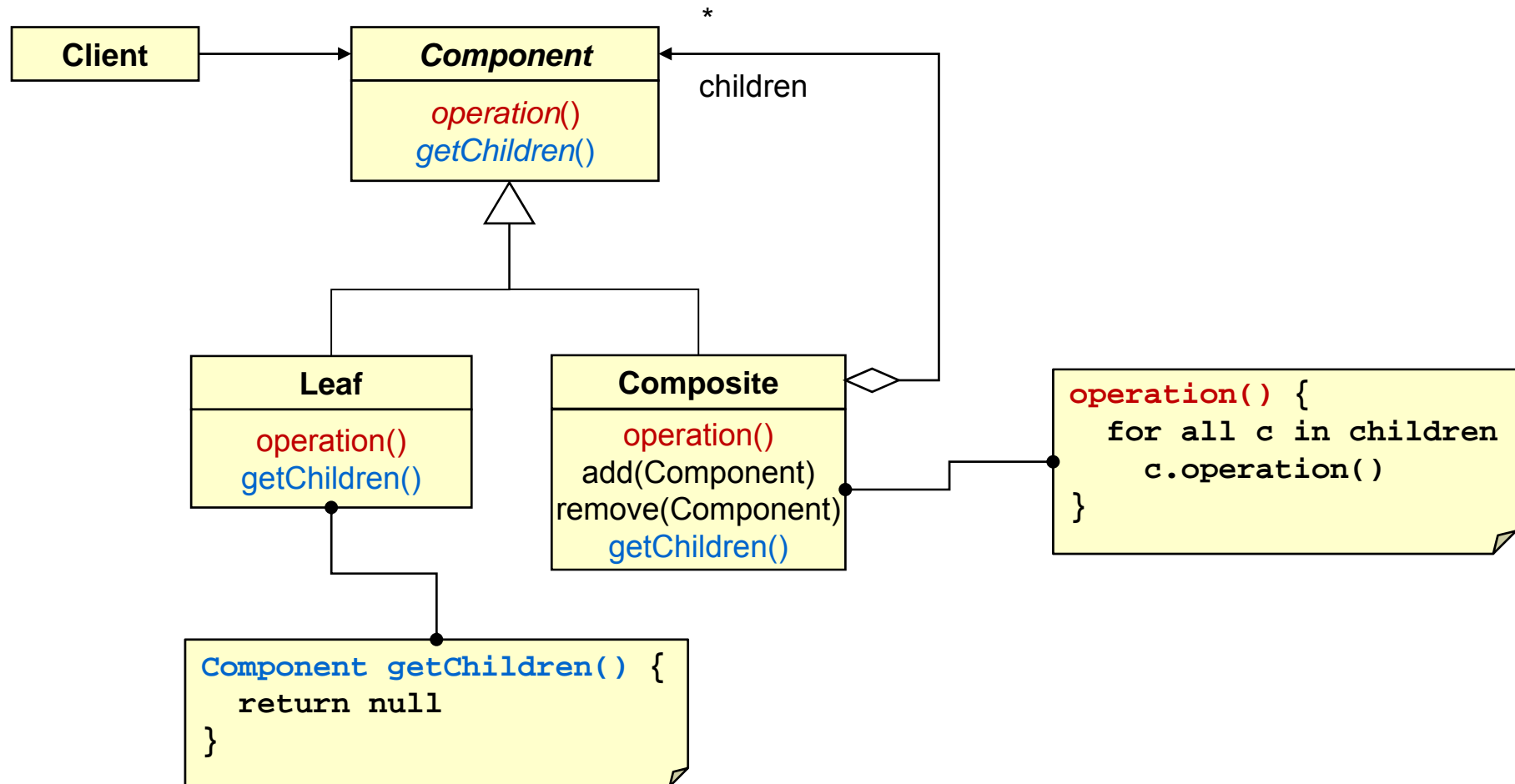
- Component Interface

- ◆ "Sauberes Design": Verwaltungs-Operationen (add, remove) in Composite, da sie für Leafs nicht anwendbar sind.

- ◆ Wunsch nach einheitlicher Behandlung aller Graphic-Objekte durch Clients
→ Verwaltungs-Operationen in Component mit default-Implementierung die nichts tut

- Leaf-Klassen sind damit zufrieden, Composites müssen die Operationen passend implementieren.

Composite Pattern: Alternative Struktur (add / remove nicht in „Component“)



Composite Pattern: Konsequenzen

- Einheitliche Behandlung
 - ◆ Teile
 - ◆ Ganzes
- Einfache Clients
 - ◆ Dynamisches Binden statt Fallunterscheidungen
- Leichte Erweiterbarkeit
 - ◆ neue Leaf-Klassen
 - ◆ neue Composite-Klassen
 - ◆ ohne Client-Änderung
- Evtl. zu allgemein
 - ◆ Einschränkung der Komponenten-Typen schwer
 - ◆ „run-time type checks“ (instanceof)

Hier geht's weiter am Di, 7.12.2010

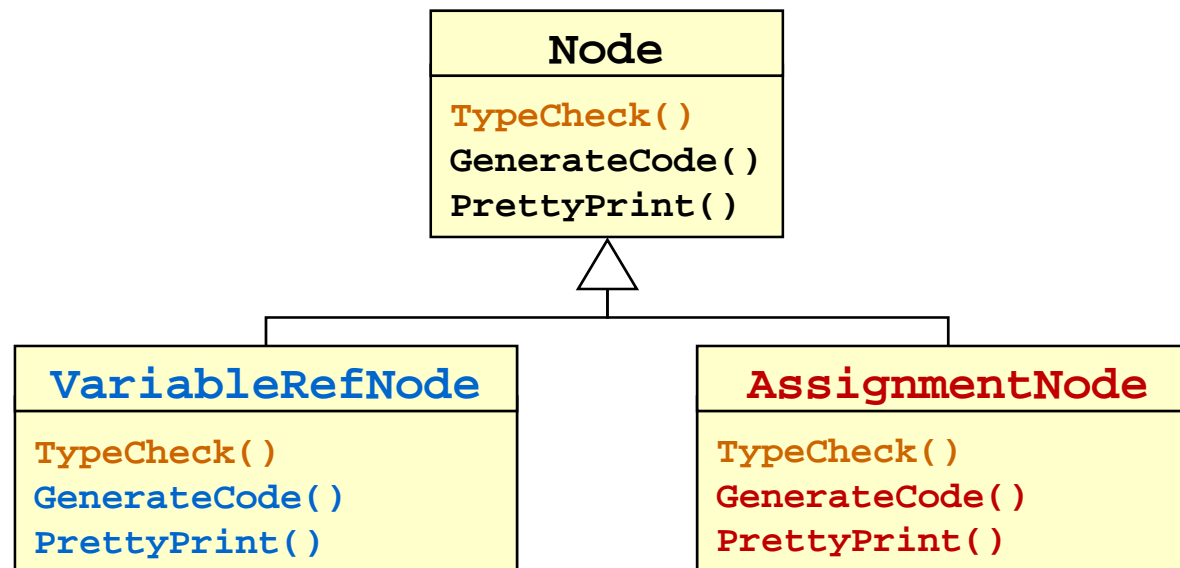
Das Visitor Pattern

Das Visitor Pattern

- Absicht
 - ◆ Repräsentation von Operationen auf Elementen einer Objektstruktur
 - ◆ Neue Operationen definieren, ohne die Klassen dieser Objekte zu ändern

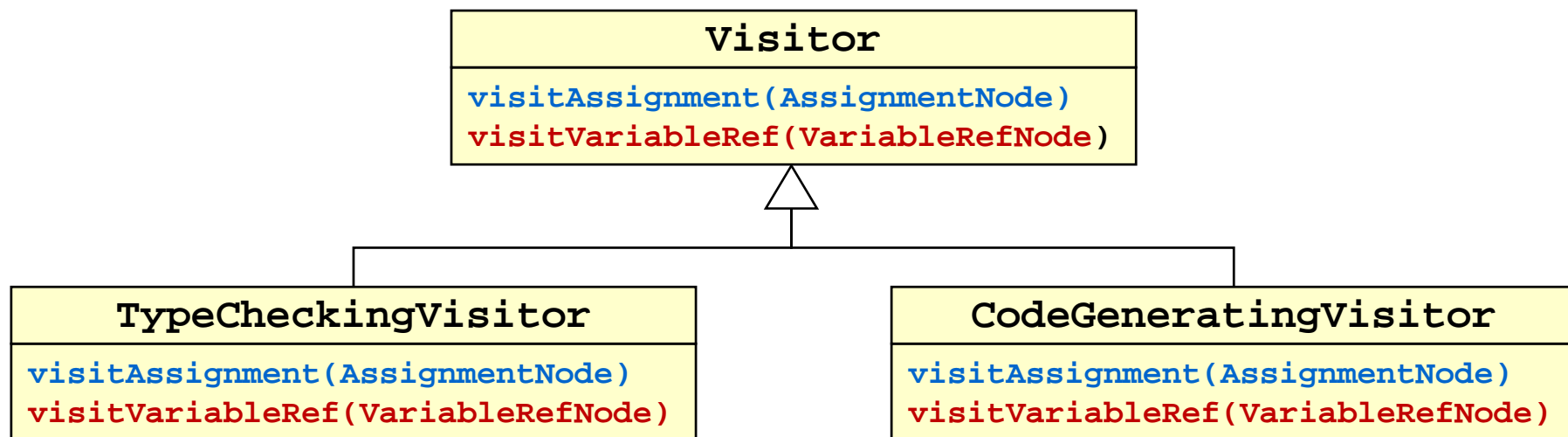
Visitor Pattern: Motivation

- Beispiel: Compiler
 - ◆ enthält Klassenhierarchie für Ausdrücke einer Programmiersprache
- Problem
 - ◆ Code einer Operation (z.B. **Typüberprüfung**) ist über mehrere Klassen einer Datenstruktur verteilt
 - ◆ Hinzufügen oder ändern einer Operation erfordert Änderung der gesamten Hierarchie



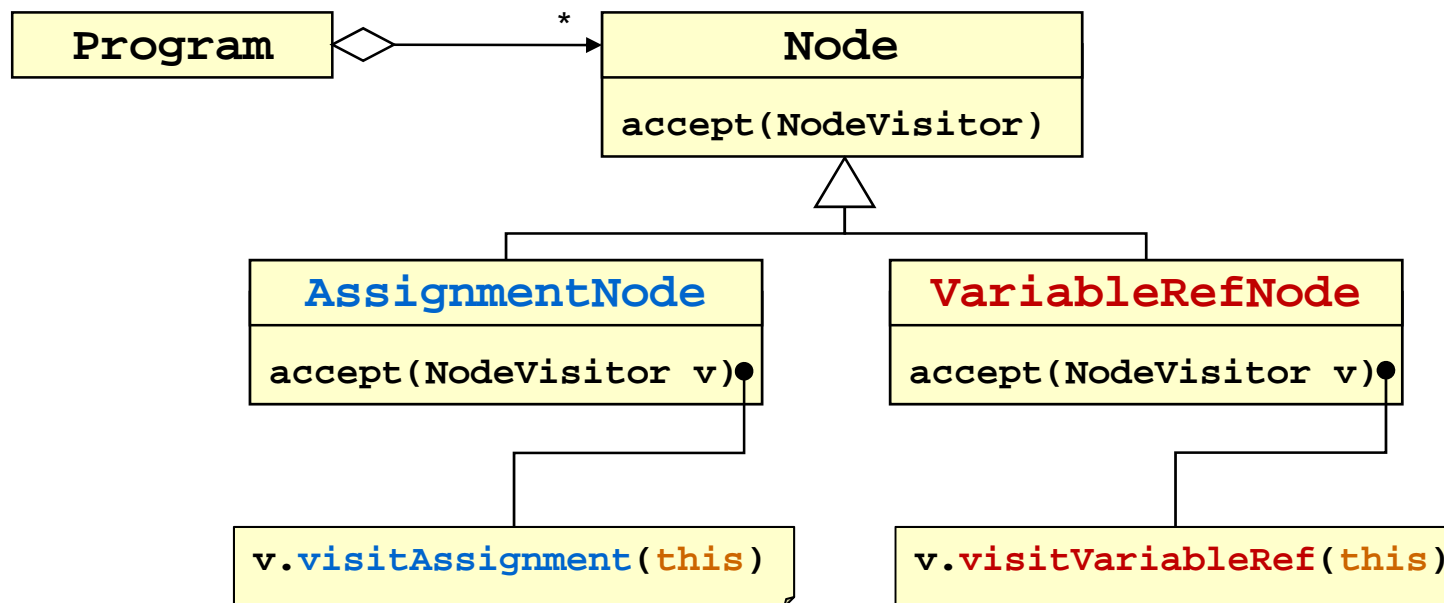
Visitor Pattern: Idee (1)

- Visitor
 - ◆ Klasse die alle visit...-Methoden zusammenfasst, die gemeinsam eine Operation auf einer Objektstruktur realisieren
- Visit...-Methode
 - ◆ Ausprägung der Operation auf einem bestimmtem Objekttyp
 - ◆ Hat Parameter vom entsprechenden Objekttyp
 - ⇒ Kann somit alle Funktionalitäten des Typs nutzen um das zu besuchende Objekt zu bearbeiten

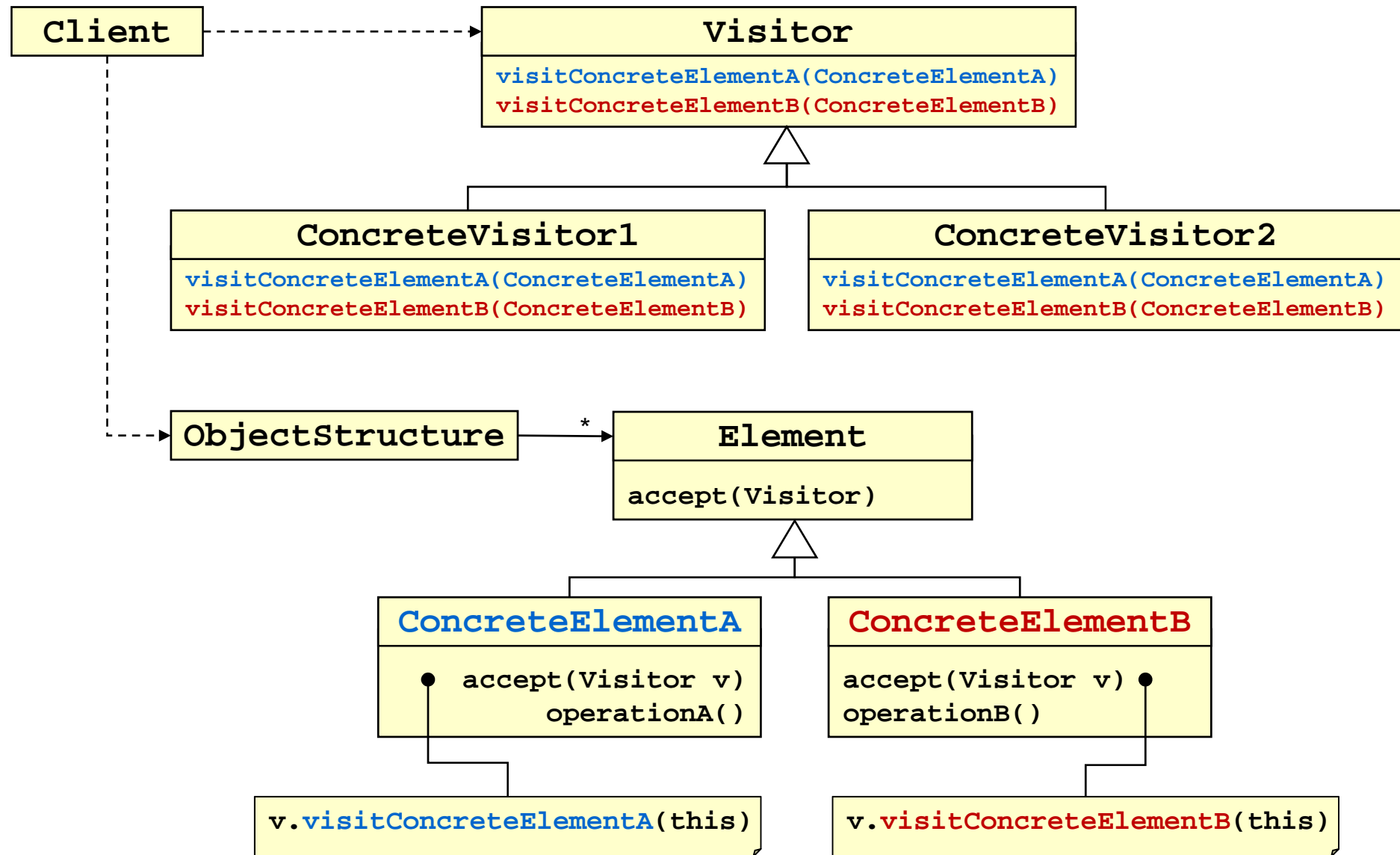


Visitor Pattern: Idee (2)

- Accept-Methoden in allen Klassen der betroffenen Klassenhierarchie
 - ◆ Haben **Visitor** als Parameter
 - ⇒ „**Diese Operation** soll auf mir ausgeführt werden!“
 - ◆ Rufen die jeweils zu ihnen **passende visit...-Methode** auf
 - ⇒ „**Diese Variante der Operation** muss auf **mir** ausgeführt werden!“
 - ⇒ Übergeben „**this**“ als Parameter



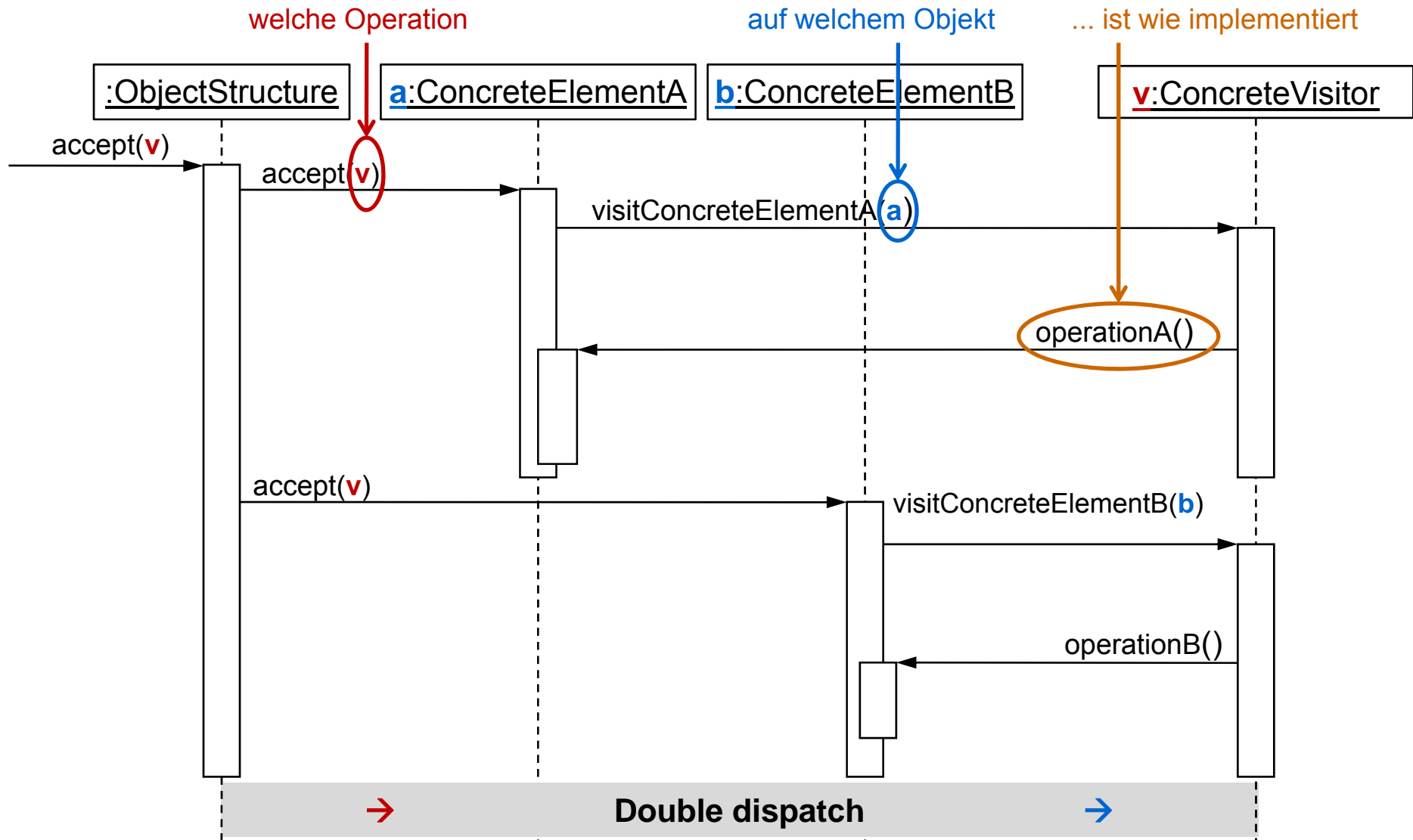
Visitor Pattern: Schema (Statisch)



Visitor Pattern: Verantwortlichkeiten / Implementation

- Objektstruktur-Hierarchie
 - ◆ Einheitliche accept-Methode
- Visitor-Hierarchie
 - ◆ Je eine visit-Methode für jede Klasse der Objektstruktur mit Parameter vom Typ der jeweilige Klasse
- Traversierung der Objektstruktur kann definiert sein in
 - ◆ der besuchten Objektstruktur (Methode accept(...)) oder
 - ◆ dem Visitor-Objekt (Method visit...(...))

Visitor Pattern: Zusammenarbeit



Das Visitor Pattern: Konsequenzen

- Hinzufügen neuer Operationen ist einfach.
 - ◆ Neue Visitor-Klasse
- Hinzufügen neuer Objektklassen ist sehr aufwendig.
 - ◆ Neue Objekt-Klasse
 - ◆ Neue visit... - Methode in allen Visitors
- Sammeln von Zuständen
 - ◆ Visitor-Objekte können Zustände der besuchten Objekte aufsammeln und (evtl. später) auswerten.
 - ◆ Eine Übergabe von zusätzlichen Parametern ist dafür nicht nötig.
- Verletzung von Kapselung
 - ◆ Die Schnittstellen der Klassen der Objektstruktur müssen ausreichend Funktionalität bieten, damit Visitor-Objekte ihre Aufgaben erledigen können.
 - ◆ Oft muss mehr preisgegeben werden als (ohne Visitor) nötig wäre.

Das Visitor Pattern: Anwendbarkeit

- Funktionale Dekomposition
 - ◆ Zusammengehörige Operationen sollen zusammengefasst werden
 - ◆ ... statt in einer Klassenhierarchie verteilt zu sein
- Stabile Objekt-Hierarchie
 - ◆ selten neue Klassen
 - ◆ aber oft neue Operationen
- Heterogene Hierarchie
 - ◆ viele Klassen mit unterschiedlichen Schnittstellen
 - ◆ Operationen die von den Klassen abhängen

- Anwendungsbeispiel
 - ◆ Java-Compiler des JDK 1.3

Das Visitor Pattern: Implementierung

- Abstrakter Visitor
 - ◆ Jede Objektstruktur besitzt **eine** (abstrakte) Visitor-Klasse.
 - ◆ **Für jeden Typ T** in der Objektstruktur, enthält die Visitor-Klasse **je eine** Methode mit einem Parameter vom Typ T → `visitT(T)`
- Konkrete Visitors
 - ◆ Jede Unterklasse der Visitor-Klasse redefinieren die visit-Methoden, um ihre jeweilige Funktionalität zu realisieren.
 - ◆ Jede konkrete `visitT(T)` Methode benutzt dabei die spezifischen Operationen des besuchten Objekttyps T
- Traversierung der Objektstruktur
 - ◆ kann in der Objektstruktur (`accept(...)` Methoden) definiert sein
 - ◆ ... oder im Visitor-Objekt (`visit...(...)` Methoden).

Gemeinsamkeiten und Unterschiede der Pattern

Abgrenzung: Facade, Singleton, Abstract Factory

- Facade
 - ◆ Versteckt Subsystem-Details (Typen und Objekte)
- Singleton
 - ◆ Facaden sind meist Singletons (man braucht nur eine einzige)
- Abstract Factory
 - ◆ Zur Erzeugung konsistenter Sätze von Subsystem-Objekten
 - ◆ Als Alternative zu Facade → "Verstecken" plattform-spezifischer Klassen

Abgrenzung: Proxy, Decorator, Adapter

- Proxy
 - ◆ gleiches Interface
 - ◆ kontrolliert Zugriff
 - ◆ "Implementierungs-Detail"
- Decorator
 - ◆ erweitertes Interface
 - ◆ zusätzliche Funktionen
 - ◆ „konzeptionelle Eigenschaft“
- Adapter
 - ◆ verschiedenes Interface
 - ◆ ähnlich Protection-Proxy

Abgrenzung: Bridge, Adapter, Abstract Factory

- Bridge
 - ◆ antizipierte Entkopplung von Schnittstelle und Implementierung
- Adapter
 - ◆ nachträgliche Anpassung der Schnittstelle einer Implementierung
- Abstract Factory
 - ◆ nutzbar zur Erzeugung und Konfiguration einer Bridge

Abgrenzung: Factory / Template Method, Abstract Factory, Prototype

- Factory Method
 - ◆ Verzögerte Entscheidung über die Klasse eines zu erzeugenden Objektes
- Template Method
 - ◆ ruft oft Factory Method auf
- Abstract Factory
 - ◆ gleiche Motivation
 - ◆ gruppiert viele Factory Methods die „zusammengehörige“ Objekte erzeugen
 - ◆ erzeugt feste Menge von Objekten von festen Objekttypen
- Prototype
 - ◆ erfordert keine Unterklassenbildung
 - ◆ ... dafür aber explizite initialize()-Methode

„Patterns Create Architectures“

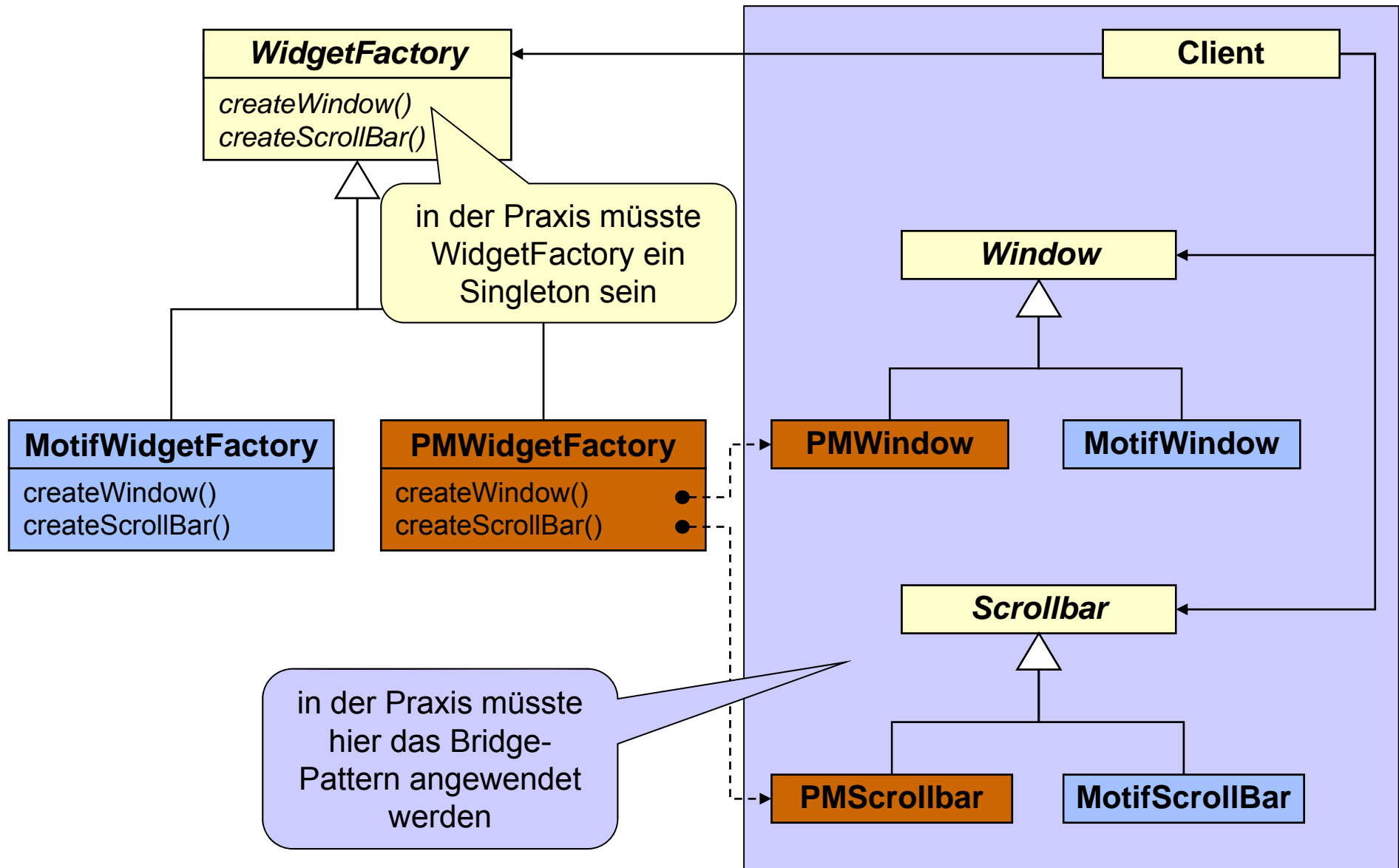
Ein Beispiel zum Zusammenspiel von Patterns

Bridge & Abstract Factory & Singleton

„Patterns Create Architectures“

- Idee
 - ◆ Wenn man Patterns wohlüberlegt zusammen verwendet, entsteht ein Grossteil einer Software-Architektur aus dem Zusammenspiel der Patterns.
- Beispiel
 - ◆ Zusammenspiel von Bridge, Factory und Singleton

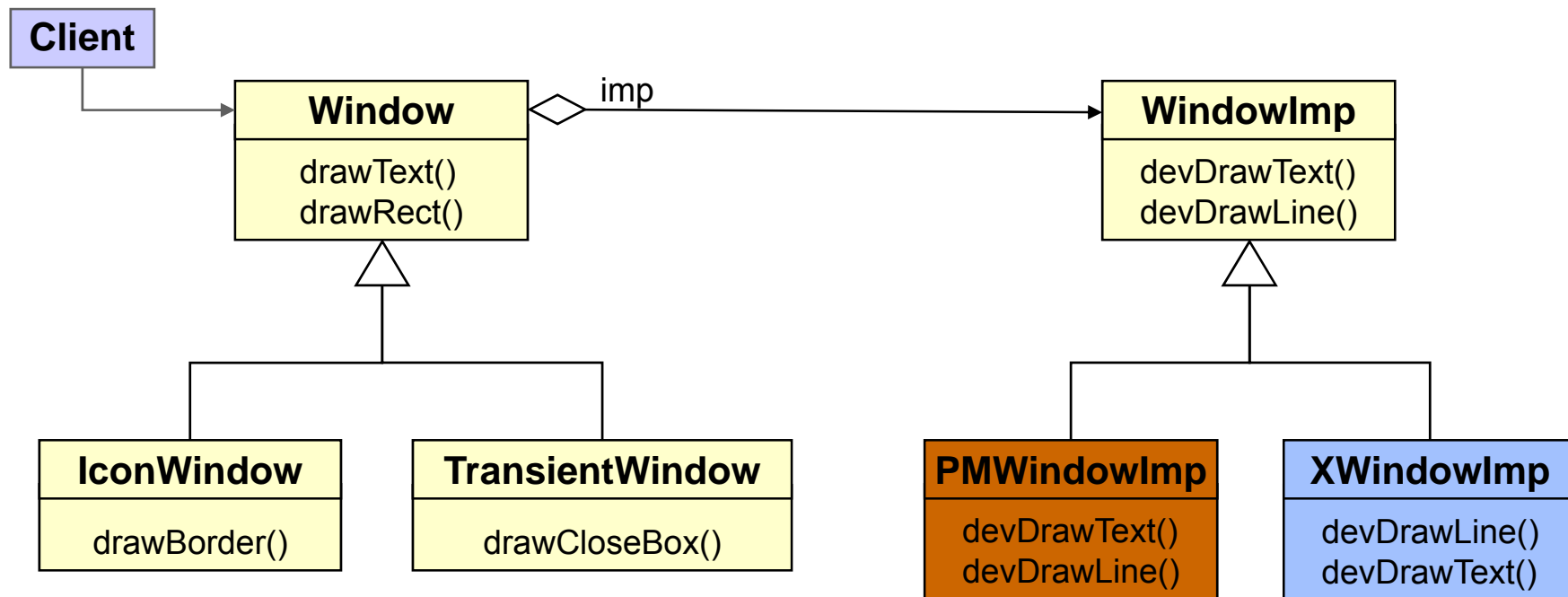
Erinnerung ans Abstract Factory Pattern



Erinnerung ans Bridge Pattern

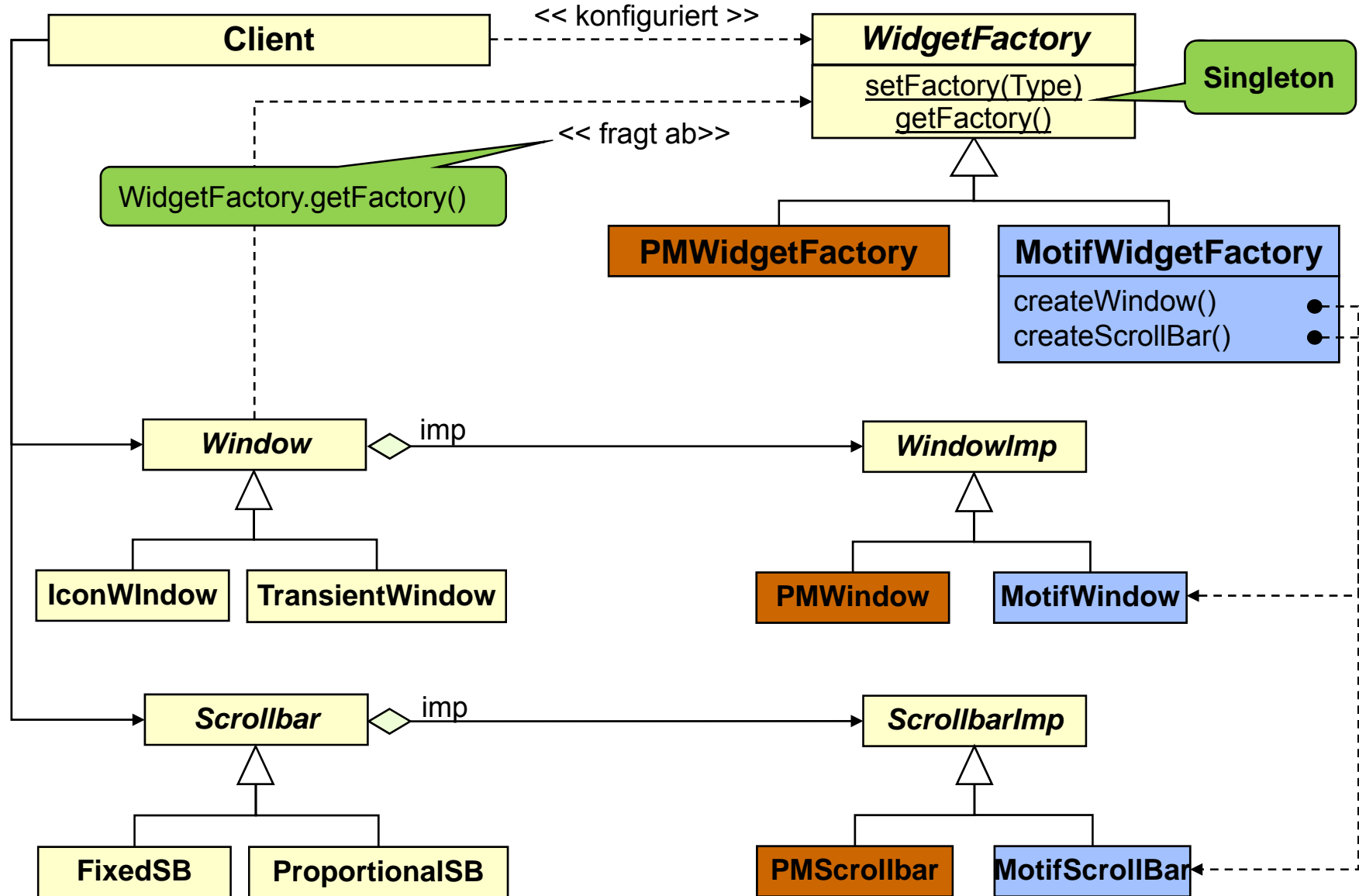
- Trennung von
 - ◆ Konzeptueller Hierarchie

- ◆ Implementierungshierarchie



Zusammenspiel: Bridge, Factory, Singleton

`WidgetFactory.setFactory(WidgetFactory.MOTIF)`



Rückblick: Was nützen Patterns?

Rückblick: Was nützen Patterns?

Nutzen von Design Patterns (1)

- Objekte / Klassen identifizieren
 - ◆ Abstraktionen, die kein Gegenstück in der realen Welt / dem Analyse-Modell haben
 - ◆ Beispiel:
 - ⇒ Composite Pattern
 - ⇒ Abstraktionen von Prozessen
 - ⇒ Strategy
 - ⇒ State
 - ◆ **"Strict modelling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible."**

Nutzen von Design Patterns (2)

- Granularität der Objekte festlegen
 - ◆ Facade
 - ⇒ ein "Riesen-Objekt"
 - ◆ Flyweight
 - ⇒ viele kleine, gemeinsam verwendbare Teilobjekte
 - ◆ Builder
 - ⇒ Komposition von Gesamt-Objekt aus Teilobjekten
 - ◆ Visitor
 - ⇒ Gruppe von konsistenten Aktionen
 - ◆ Command
 - ⇒ einzelne Aktion

Nutzen von Design Patterns (3)

- Schnittstellen identifizieren
 - ◆ was gehört dazu
 - ◆ was gehört nicht dazu (Bsp. Memento)
- Beziehungen zwischen Schnittstellen festlegen
 - ◆ Subtyping
 - ⇒ Proxy
 - ⇒ Decorator
 - ◆ Je eine Methode pro Klasse aus einer anderen Objekthierarchie
 - ⇒ Abstract Factory
 - ⇒ Visitor
 - ⇒ Builder

Nutzen von Design Patterns (4)

- Wahl der Implementierung
 - ◆ Interface, Abstrakte Klasse oder konkrete Klasse
 - ⇒ Grundthema fast aller Patterns
 - ◆ Abstrakte Methode oder Hook Methode
 - ⇒ von template method aufgerufene Methoden
 - ◆ Overriding oder fixe Implementierung
 - ⇒ Factory method
 - ⇒ Template method
 - ◆ Vererbung oder Subtyp-Beziehung
 - ⇒ Adapter, Decorator, State, Strategy, Command, Chain of Responsibility → Gemeinsamer Obertyp, nicht gemeinsame Implementierung
 - ◆ Vererbung oder Aggregation
 - ⇒ Vererbung ist statisch, Aggregation dynamisch
 - ⇒ Es wird oft mehr vererbt als gewünscht
 - ⇒ Beispiel: alle "split object patterns"

Nutzen von Design Patterns (5):

- Patterns verkörpern Grundprinzipien guten Designs
- Implementierungen austauschbar machen
 - ◆ Typdeklarationen mit Interfaces statt mit Klassen
 - ◆ Design an Interfaces orientieren, nicht an Klassen
 - ◆ Client-Code wiederverwendbar für neue Implementierungen des gleichen Interface
- Objekt-Erzeugung änderbar gestalten
 - ◆ "Creational Patterns" statt "new MyClass()"
 - ◆ Client-Code wiederverwendbar für neue Implementierungen des gleichen Interface
- Funktionalität änderbar gestalten
 - ◆ Aggregation und Forwarding statt Vererbung
 - ◆ späte Konfigurierbarkeit, Dynamik
 - ◆ weniger implizite Abhängigkeiten (kein "fragile base class problem")

Nichtfunktionale Anforderungen geben Hinweise zur Nutzung von Entwurfsmustern (Design Patterns)

- Identifikation von **Entwurfsmustern** anhand von Schlüsselwörtern in der Beschreibung nichtfunktionaler Anforderungen
 - ◆ Analog zu Abbot's Technik bei der Objektidentifikation
- **Facade** Pattern
 - ◆ „muss mit einer Menge existierender Objekte zusammenarbeiten“,
 - ◆ „stellt Dienst bereit“
- **Adapter** Pattern
 - ◆ „muss mit einem existierenden Objekt zusammenarbeiten“
- **Bridge** Pattern
 - ◆ „muss sich um die Schnittstelle zu unterschiedlichen Systemen kümmern von denen einige erst noch entwickelt werden.“
 - ◆ „ein erster Prototyp muss vorgeführt werden“

Nichtfunktionale Anforderungen geben Hinweise zur Nutzung von Entwurfsmustern (Fortsetzung)

- **Proxy** Pattern
 - ◆ “muss ortstransparent sein”
- **Observer** Pattern
 - ◆ “muss erweiterbar sein”, “muss skalierbar sein”
- **Strategy** Pattern
 - ◆ “muss Funktionalität X in unterschiedlichen, dynamisch auswählbaren Ausprägungen bereitstellen können”
- **Composite** Pattern
 - ◆ “komplexe Struktur”
 - ◆ “muss variable Tiefe und Breite haben”
- **Abstract Factory** Pattern
 - ◆ “Herstellerunabhängig”,
 - ◆ “Geräteunabhängig”,
 - ◆ “muss eine Produktfamilie unterstützen”

Überblick

- Einführung
 - ◆ Grundidee, Literatur, MVC-Framework als Beispiel
- Beispiele wichtiger Patterns
 - ◆ Observer, Strategy, Command
 - ◆ Facade, Singleton, Proxy, Adapter, Bridge
 - ◆ Factory Method, Abstract Factory
 - ◆ Composite, Visitor
- Patterns Create Architectures
 - ◆ Bridge, Abstract Factory, Singleton
- Nutzen von Patterns
- Zusammenfassung und Ausblick
 - ◆ Arten von Patterns, Abgrenzung
 - ◆ Weiterführende Themen

Arten von Pattern

- Analysis Pattern

- ◆ Wiederkehrende Problemen in der Analysephase eines Softwareprojekts
- ◆ Martin Fowler: "Analysis Patterns", Addison-Wesley, 1997

- Architecture Pattern

- ◆ ... beschreibt mögliche fundamentale Strukturen von Softwaresystemen.
- ◆ ... enthält vordefinierte Subsysteme und ihre Verantwortlichkeiten.
- ◆ ... enthält Regeln und Richtlinien für die Organisation ihrer Beziehungen.
- ◆ Beispiel: Model-View-Controller

- Design Pattern

- ◆ Schema für die Realisation von Subsystemen oder Komponenten eines Softwaresystems

- Idiom (auch: Coding Pattern)

- ◆ Low-level design patterns für eine gegebene Programmiersprache.
- ◆ Beispiel: Wie stelle ich sicher, dass eine Instanz einer Java-Klasse nur innerhalb einer bestimmten anderen Klasse erzeugt werden kann?

Weitere Arten von Pattern

- Organizational Patterns
 - ◆ Struktur und Praxis von Organisationen; also Gruppen, die ein gemeinsames Ziel verfolgen
 - ◆ <http://www.bell-labs.com/cgiuser/OrgPatterns/OrgPatterns?OrganizationalPatterns>
- Anti-Patterns
 - ◆ beschreiben typische Fehler
 - ◆ ... und bestenfalls auch wie man sie wieder beseitigt

Abgrenzung ▶ Pattern versus Algorithmen

- **Konkretheit**
 - ◆ Algorithmen lösen konkrete Probleme im Anwendungsbereich (z.B. Suchen, Sortieren)
 - ◆ Pattern lösen generische Entwurfsprobleme (Dynamik, Wartbarkeit, ...)
- **Vollständigkeit**
 - ◆ Algorithmen sind vollständig → Komplexität genau ermittelbar
 - ◆ Pattern sind „nur“ Schablonen

Abgrenzung

- Framework
 - ◆ Menge von kooperierenden Klassen für einen spezifischen Anwendungsbereich
 - ◆ Erweiterbar durch Unterklassenbildung und Komposition von Instanzen
 - ◆ Hollywood-Prinzip ("Don't call us, we'll call you." --> Template Method Pattern)

- Pattern versus Frameworks
 - ◆ Patterns sind abstrakter
 - ⇒ Frameworks existieren als konkreter, wiederverwendbarer Code
 - ⇒ Patterns sind nur Schablonen für Code
 - ◆ Patterns sind nicht anwendungsabhängig
 - ⇒ Frameworks werden für konkrete Anwendungsbereiche eingesetzt
 - ⇒ Patterns sind anwendungsunabhängig
 - ◆ Frameworks nutzen Patterns

Weiterführende Themen

- Pattern Catalogs
 - ◆ Sammlungen von lose zusammenhängenden Patterns
- Pattern Systems
 - ◆ Sammlungen von stark zusammenhängenden Patterns mit engen Verzahnungen
- Pattern Languages
 - ◆ verfolgen ein gemeinsames Ziel, dass durch die Zusammenarbeit der enthaltenen Patterns erreicht werden kann
 - ◆ inkl. einer Art "Grammatik", die alle mögliche Kombinationen aufzeigt

Pattern ▶ Zusammenfassung

- Betonung von Praktikabilität
 - ◆ Patterns sind bekannte Lösungen für erwiesenermaßen wiederkehrende Probleme
 - ◆ Lösungen, die sich noch nicht in der Praxis bewährt haben, sind keine Pattern.
- Patterns sind kein Allheilmittel
 - ◆ Originalität bei der Anwendung von Patterns ist nach wie vor gefragt.
 - ◆ Es muss immer noch abgewägt werden, welche Patterns eingesetzt werden.
- Gesamteffekt
 - ◆ Aufgabe des Softwarearchitekten verlagert sich von der Erfindung des Rades zur Auswahl des richtigen Rades und seiner kreativen Anwendung