

Entwurfsmuster

Alexander Paar

Institute for Program Structures and Data Organization
Programming Systems Group Prof. Tichy
Universität Karlsruhe (TH)

Inhalt dieser Foliensammlung

- Einführung
- Einführungsbeispiel: das „kleinste“ Muster
- Größeres Beispiel: ein zusammengesetztes Muster
- Zusammenfassung
- Großer Entwurfsmusterkatalog



Definition

Ein Software-Entwurfsmuster beschreibt eine Familie von Lösungen für ein Software-Entwurfsproblem.

Das Ziel eines Entwurfsmusters ist die Wiederverwendbarkeit von Entwurfswissen.

Entwurfsmuster sind für Entwurf (oder das Programmieren im Großen) was Algorithmen für das Programmieren im Kleinen sind.



Definition

- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

Christopher Alexander et al: *A Pattern Language*, Oxford University Press, New York, 1977



Einführungsbeispiel: Einzelstück (Singleton)

Zweck

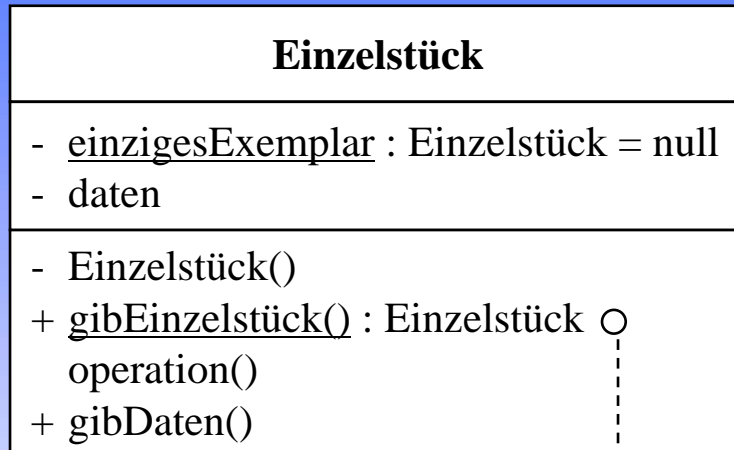
Sichere zu, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.

Motivation

Die Klasse ist selbst für die Verwaltung ihres einzigen Exemplars zuständig. Die Klasse kann durch Abfangen von Befehlen zur Erzeugung neuer Objekte sicherstellen, dass kein weiteres Exemplar erzeugt wird.



Struktur des Einzelstücks



if einzigesExemplar == null
 then einzigesExemplar = new Einzelstück()
 return einzigesExemplar



Einzelstück

Anwendbarkeit

- Wenn es von einer Klasse nur eine einzige Instanz geben darf und diese Instanz den Klienten an einer bekannten Stelle zugänglich gemacht werden soll.
- Wenn es schwierig oder unmöglich ist, festzustellen, welcher Teil der Anwendung die erste Instanz erzeugt.
- Wenn die einzige Instanz durch Unterklassenbildung erweiterbar sein soll und die Klienten ohne Veränderung ihres Quelltextes diese nutzen sollen.



Erweitertes Beispiel: Model/View/Controller (MVC)

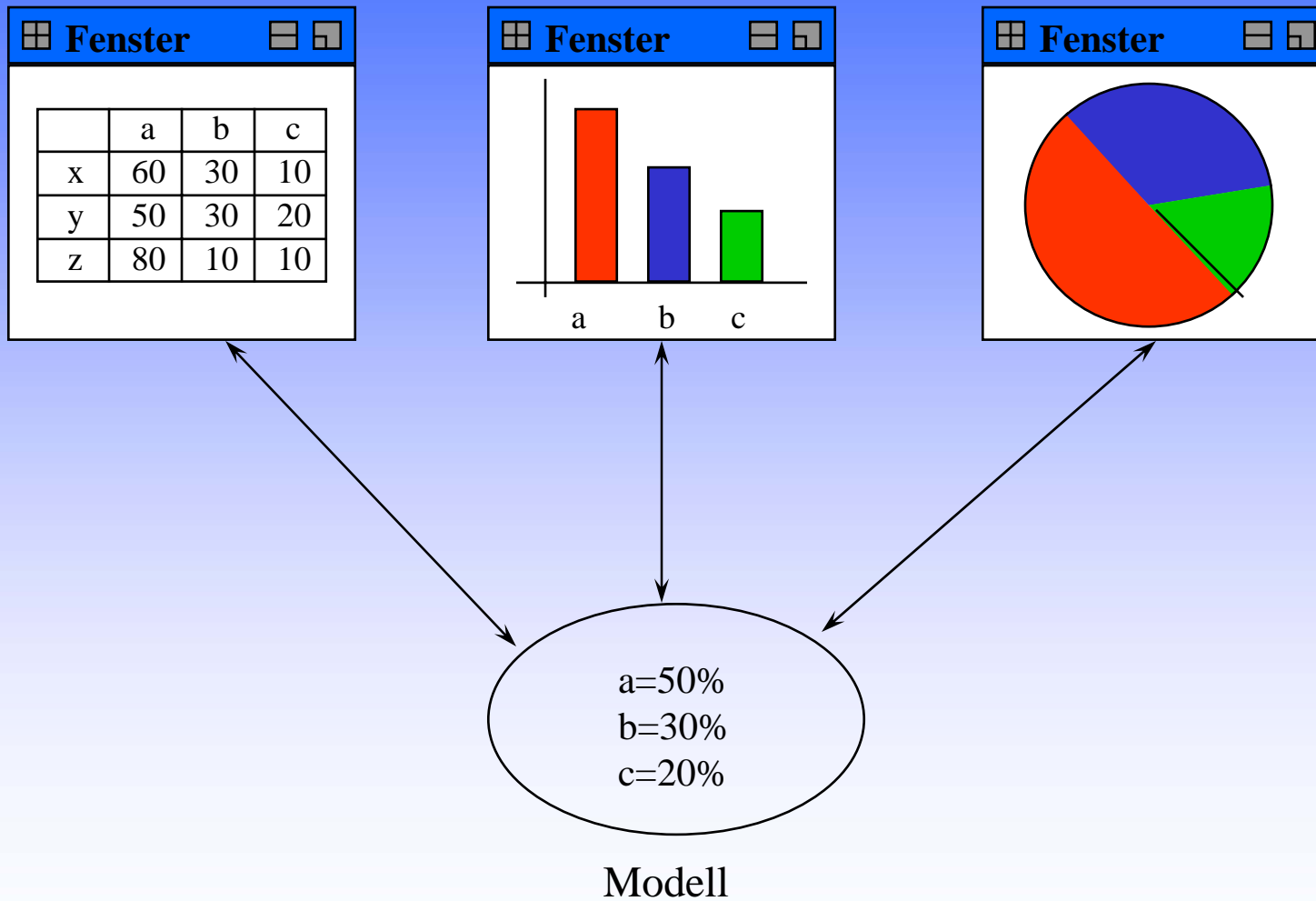
MVC ist eine Klassenkombination zur Konstruktion von Benutzerschnittstellen (entstanden zuerst in Smalltalk).

- *Modell (Model)*: Anwendungsobjekt
- *Sicht (View)*: Darstellung des *Modells* auf dem Bildschirm (evtl. mehrfach)
- *Steuerung (Controller)*: Definiert Reaktion der Benutzerschnittstelle auf Eingaben

MVC ist ein Entwurfsmuster, das die drei Entwurfsmuster *Beobachter*, *Kompositum* und *Strategie* kombiniert.



Muster 1 im MVC: Beobachter (*Observer*)



Muster 1 im MVC: Beobachter

Muster 1: Die Beziehung zwischen Modell und Sichten ist ein Beispiel für das *Beobachter*-Muster.

- Es gibt eine Registrierungs- und Benachrichtigungs-Interaktion zwischen dem *Modell* (dem „Subjekt“) und den *Sichten* (den „Beobachtern“).
- Wenn Daten im *Modell* sich ändern, werden die *Sichten* benachrichtigt. Daraufhin aktualisiert jede *Sicht sich selbst* durch Zugriff auf das *Modell*.
- Die *Sichten* wissen nichts voneinander. Das *Modell* weiß nicht, in welcher Weise die *Sichten* die Daten des *Modells* verwenden (Entkopplung).



Muster 2 im MVC: Kompositum

Muster 2: *Sichten* können geschachtelt sein. Eine zusammengesetzte *Sicht* ist ein Beispiel für das Muster *Kompositum*.

- Zum Beispiel kann ein *Objektinspektor* aus geschachtelten *Sichten* bestehen und kann selbst in der Benutzerschnittstelle eines Debuggers enthalten sein.
- Die Klasse *ZusammengesetzteSicht* ist eine Unterklasse von *Sicht*. Ein Exemplar von *ZusammengesetzteSicht* kann genauso wie ein Exemplar von *Sicht* benutzt werden. Es enthält und verwaltet geschachtelte *Sichten*.



Muster 3 im MVC: Strategie

Muster 3: Die Beziehung zwischen *Sicht* und *Steuerung* ist ein Beispiel für ein *Strategie*-Muster.

- Es gibt mehrere Unterklassen von *Steuerung*, die verschiedene Antwortstrategien implementieren. Zum Beispiel werden Tastatureingaben unterschiedlich behandelt, oder ein Menü wird anstelle von Tastaturkommandos benutzt.
- Eine *Sicht* wählt eine gewisse Unterklasse aus, die die entsprechende Antwortstrategie implementiert. Diese kann auch dynamisch ausgewählt werden (zum Beispiel um ausgeschaltete Zustände zu ignorieren).



Wozu überhaupt Entwurfsmuster?

- **Muster verbessern Kommunikation im Team.**

Entwurfsmuster bilden eine nützliche Terminologie, d.h. bieten Begriffe und Kurzformeln für die Diskussion zwischen Entwicklern über komplexe Konzepte.

- **Muster erfassen wesentliche Konzepte und bringen sie in eine verständliche Form**
 - Muster helfen Entwürfe zu verstehen,
 - dokumentieren Entwürfe kurz und knapp,
 - verhindern unerwünschte Architektur-Drift,
 - verdeutlichen Entwurfswissen.



Wozu überhaupt Entwurfsmuster?

- **Muster dokumentieren und fördern den Stand der Kunst**
 - Muster helfen weniger erfahrenen Entwerfern.
 - Muster vermeiden die Neuerfindung des Rades.
 - Ein Muster ist keine feste Regel, der man blind folgt, sondern ein Vorschlag und ein Satz von Alternativen zur Lösung eines Problems. Anpassung erforderlich.
- **Muster können Code-Qualität, und -Struktur verbessern**
 - Muster fördern gute Entwürfe und guten Code durch Angabe konstruktiver Beispiele.



Die Entwicklung guter Muster ist schwierig und zeitaufwendig

- Herausstellen der wesentlichen Merkmale ist schwierig.
- Die Zusammenstellung ist ein iterativer Prozess und erfordert eine Anwendung und Verfeinerung.
- Erfahrung im Entwurf und Dokumentieren von Mustern ist hilfreich.

Entwurfsmuster sind nicht notwendigerweise objektorientiert.



Beschreibungsstruktur für Entwurfsmuster

- Name (einschließlich Synonyme)
- Aufgabe und Kontext
- Beschreibung der Lösung
 - Struktur (Komponenten, Beziehungen)
 - Interaktionen und Konsequenzen
 - Implementierung
 - Beispielcode



Beobachter (Observer)

Zweck

Definiert eine 1-zu-n Abhängigkeit zwischen Objekten, so dass die Änderung eines Zustandes eines Objektes dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

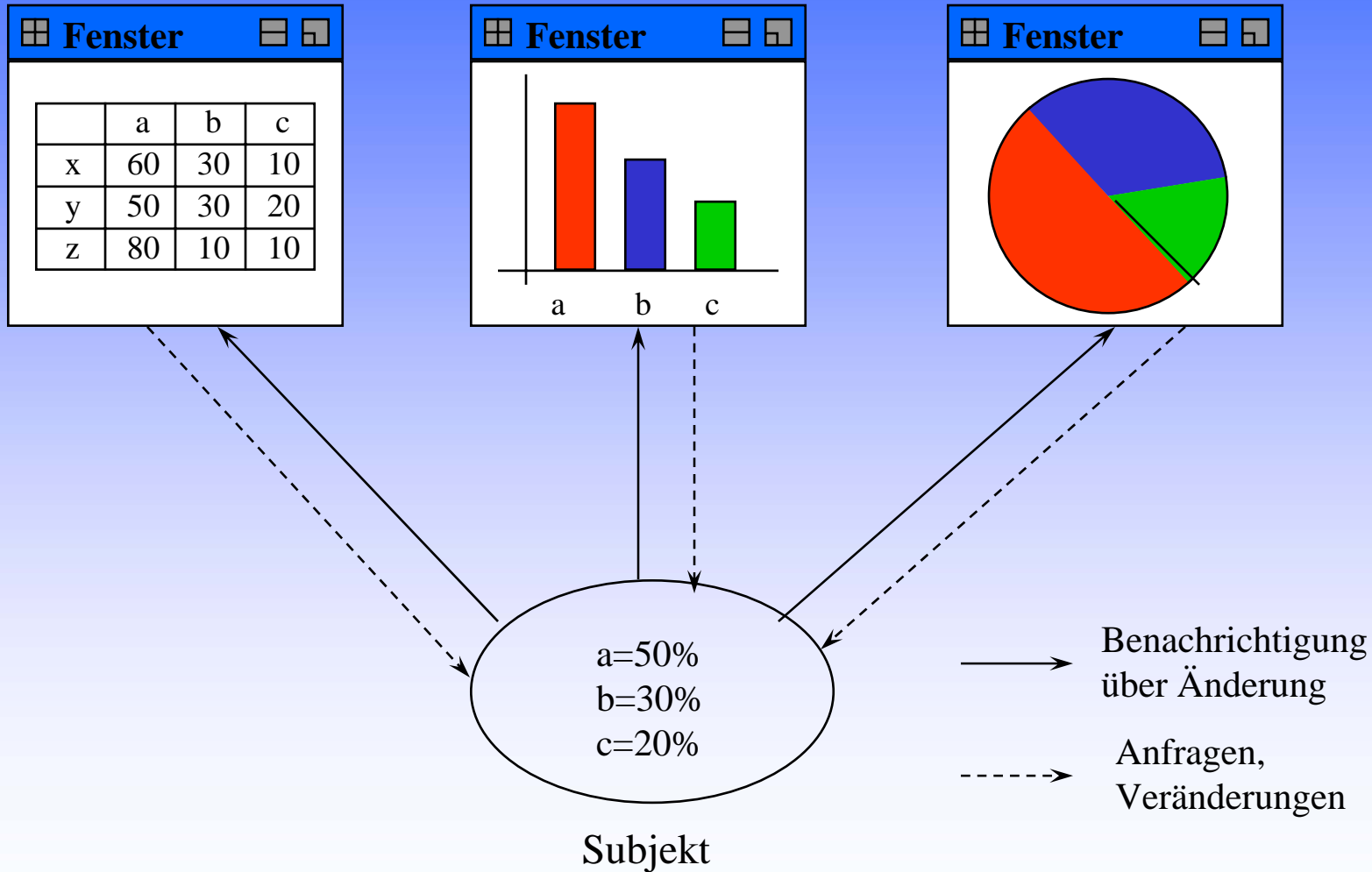
Auch bekannt als

Abhängigkeit, Publizieren-Abonnieren, Subjekt-Beobachter



Beobachter

Beobachter



Beobachter

Motivation

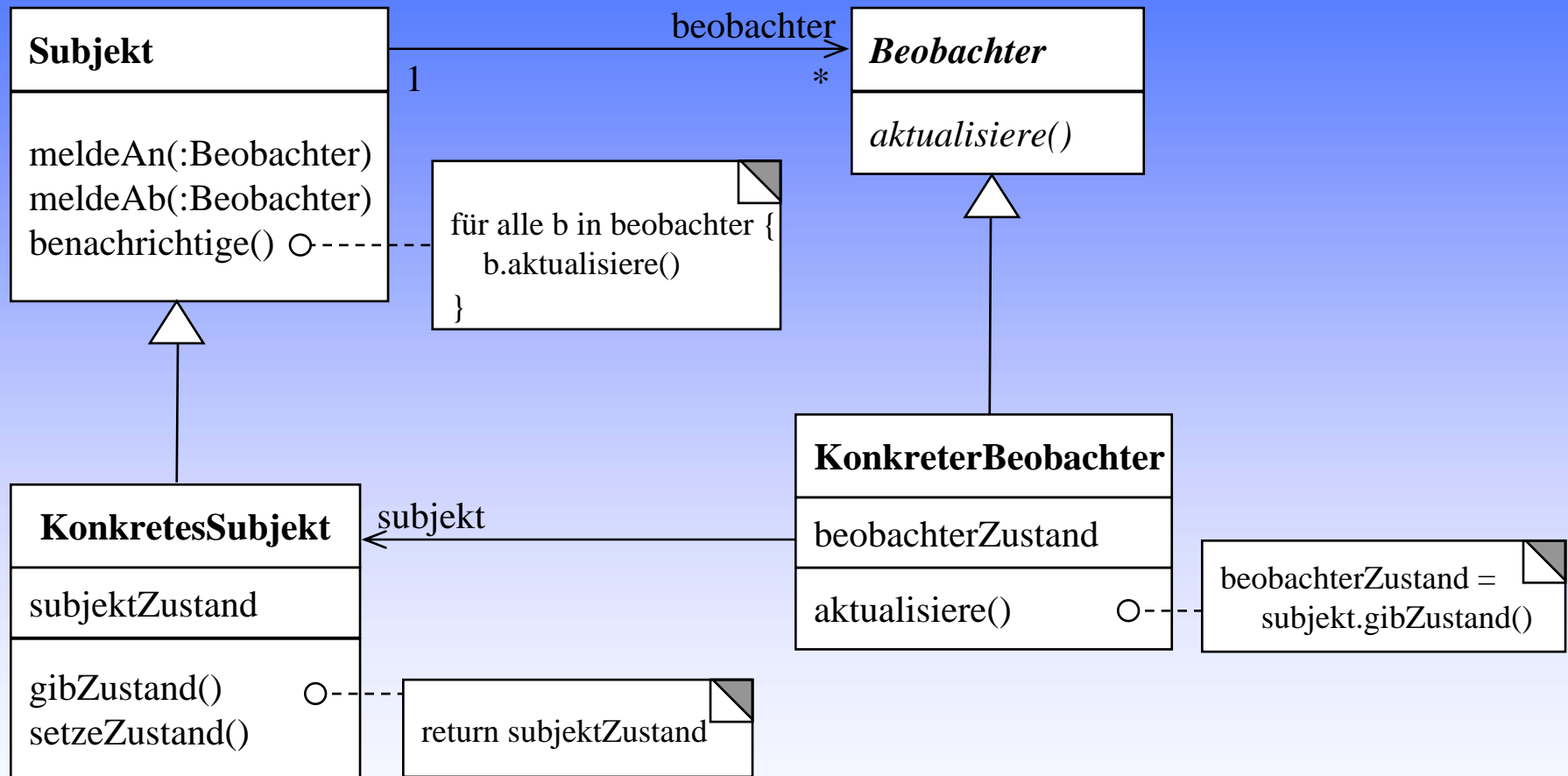
Teilt man ein System in eine Menge von interagierenden Klassen auf, so muss die Konsistenz zwischen den miteinander in Beziehung stehenden Objekten aufrechterhalten werden.

Eine enge Kopplung dieser Klassen ist nicht empfehlenswert, weil dies die individuelle Wiederverwendbarkeit einschränkt.

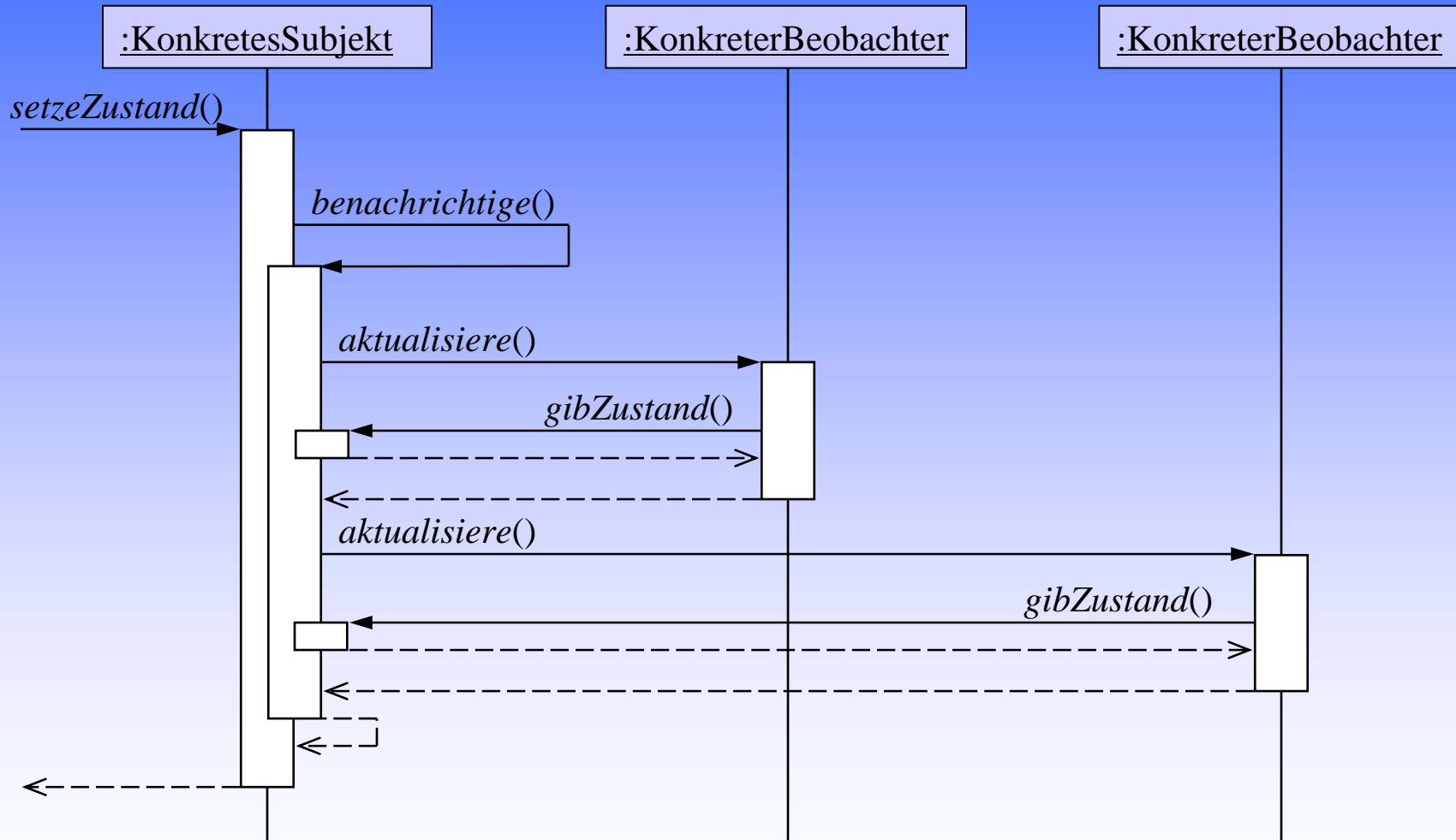
Im MVC-Beispiel wissen die Tabellendarstellung und die Säulendarstellung nichts voneinander. Damit können sie unabhängig voneinander wiederverwendet werden. Trotzdem verhalten sich beide Objekte so, als ob sie einander kennen würden.



Struktur des Beobachters



Interaktionsdiagramm Beobachter



Beobachter

Anwendbarkeit

- Wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt und man nicht weiß, wie viele und welche Objekte geändert werden müssen.
- Wenn ein Objekt andere Objekte benachrichtigen muss, ohne Annahmen über diese Objekte zu treffen.
- Wenn eine Abstraktion zwei Aspekte besitzt, wobei einer von dem anderen abhängt. Die Kapselung dieser Aspekte in separaten Objekten ermöglicht unabhängige Wiederverwendbarkeit.



Beobachter

Konsequenzen

- Subjekte und Beobachter können unabhängig voneinander wiederverwendet werden.
- Beobachter können neu hinzugefügt oder entfernt werden, ohne das Subjekt oder andere Beobachter zu ändern.
- Die abstrakte Kopplung zwischen Subjekt und Beobachter wird durch die Benachrichtigung erreicht. Subjekt und Beobachter gehören verschiedenen Schichten der Benutzthierarchie an, ohne dabei Zyklen zu erzeugen. (Subjekt benutzt nicht die Beobachter, aber umgekehrt.)



Beobachter

Konsequenzen (Fortsetzung)

- Automatischer Rundruf von Änderungen.
- Beobachter entscheiden selbst, ob sie die Benachrichtigung ignorieren oder nicht.
- Der Aufwand der Aktualisierung kann versteckt sein.
 - Eine einfache Benachrichtigung kann zu einer Kaskade von Aktualisierungen bei den Beobachtern führen.
 - Die Botschaft enthält keinen Hinweis *was* geändert wurde. Ein erweitertes Protokoll kann verwendet werden, um den Beobachtern die konkrete Änderung mitzuteilen.



Beobachter

Implementierung

1. Wenn mehr als ein Subjekt beobachtet wird: Verwende Subjekt als Parameter von ***aktualisiere(s :Subjekt)***.
2. Auslösen der Aktualisierung:
 - ***setzeZustand()*** ruft ***benachrichtige()***,
 - oder Klienten rufen ***benachrichtige()*** explizit.
3. Löschen eines Beobachters: Als erstes beim entsprechenden Subjekt abmelden.



Beobachter

Implementierung (Fortsetzung)

4. Subjekte müssen vor der Benachrichtigung konsistent sein. Wenn eine Subjekt-Unterklasse geerbte Operationen aufruft, kann versehentlich die Oberklasse eine Benachrichtigung auslösen bevor das Unterklassenobjekt komplett konsistent ist. Alternative: Schablonenmethode für die Aktualisierung verwenden.
5. Aktualisierung: Pull- und Push-Modell
 - Pull-Modell: Beobachter holt alle Daten direkt vom Subjekt (kann ineffizient sein)
 - Push-Modell: Subjekt schickt Änderungsdaten an die Beobachter in ***aktualisiere()*** (kann Wiederverwendbarkeit reduzieren)



Beobachter

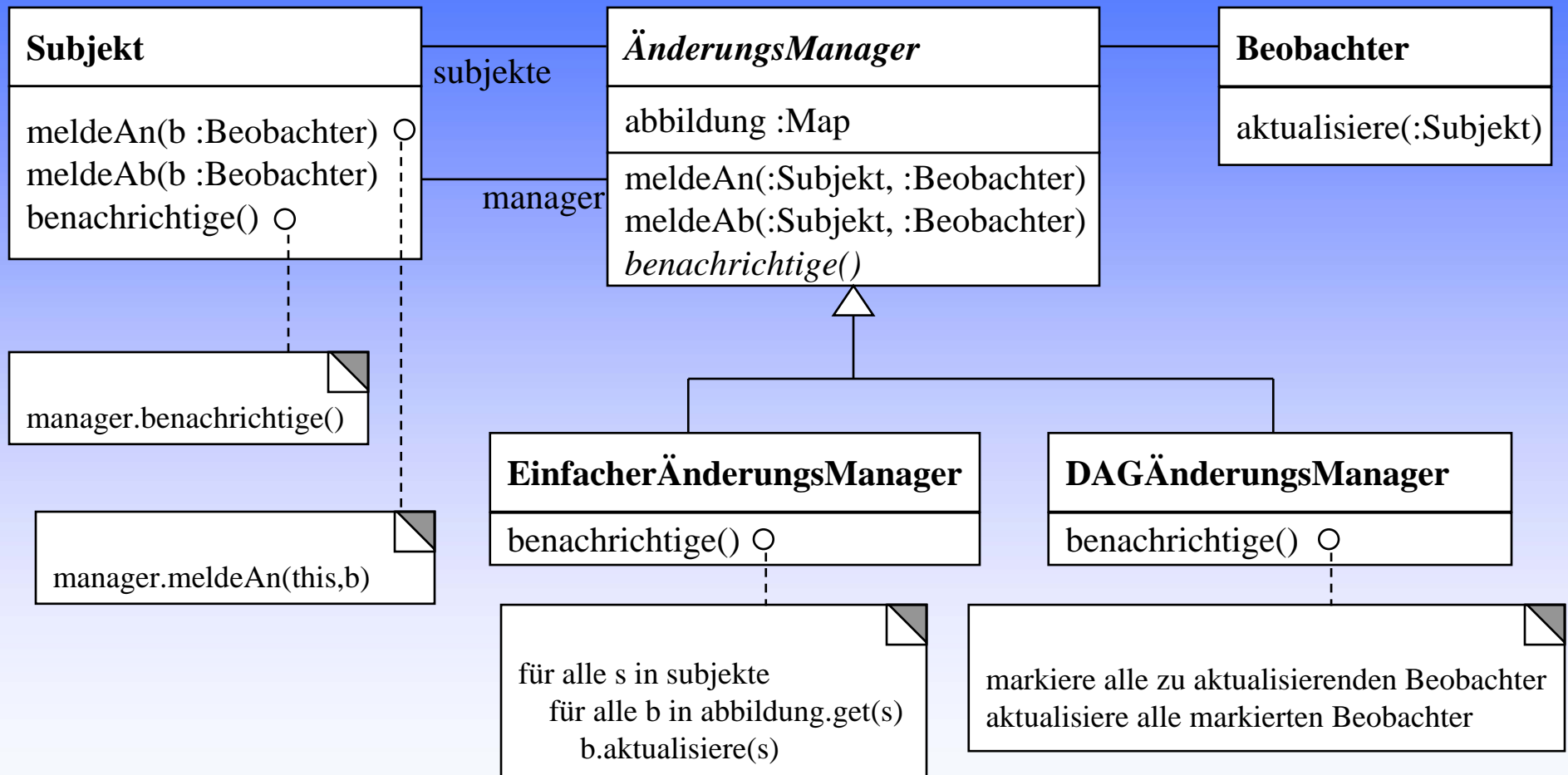
Implementierung (Fortsetzung)

6. ÄnderungsManager zwischen Subjekt und Beobachtern:

- bildet Subjekt auf seine Beobachter ab und bietet eine Schnittstelle zur Verwaltung dieser Abbildung,
- aktualisiert bei Anforderung durch das Subjekt alle abhängigen Beobachter,
- vermeidet mehrfache Aktualisierungen,
- kapselt komplexe Aktualisierungssemantik.



Beobachter mit ÄnderungsManager



Kompositum (Composite)

Zweck

Füge Objekte zu Baumstrukturen zusammen, um Bestands-Hierarchien zu repräsentieren. Das Muster ermöglicht es Klienten, sowohl einzelne Objekte als auch Aggregate einheitlich zu behandeln.



Kompositum

Motivation

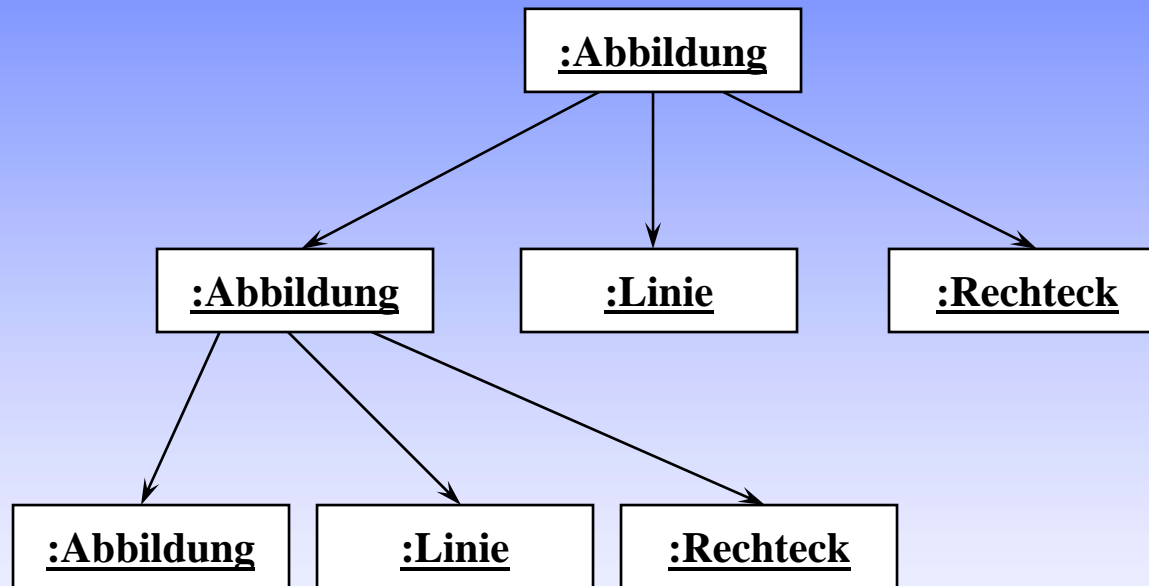
Bestands-Hierarchien treten überall dort auf, wo komplexe Objekte modelliert werden, wie beispielsweise Dateisysteme, graphische Anwendungen, Textverarbeitung, CAD, CIM, Bei diesen Anwendungen werden einfache Objekte zu Gruppen zusammengefasst, welche wiederum zu größeren Gruppen zusammengefügt werden können.

Häufig soll dabei die Behandlung von Objekten und Aggregaten durch das Programm einheitlich sein. Das *Kompositum* isoliert die gemeinsamen Eigenschaften von Objekt und Aggregat und bildet daraus eine Oberklasse.



Beispiel für Kompositum

Zusammengefügte Graphik-Objekte

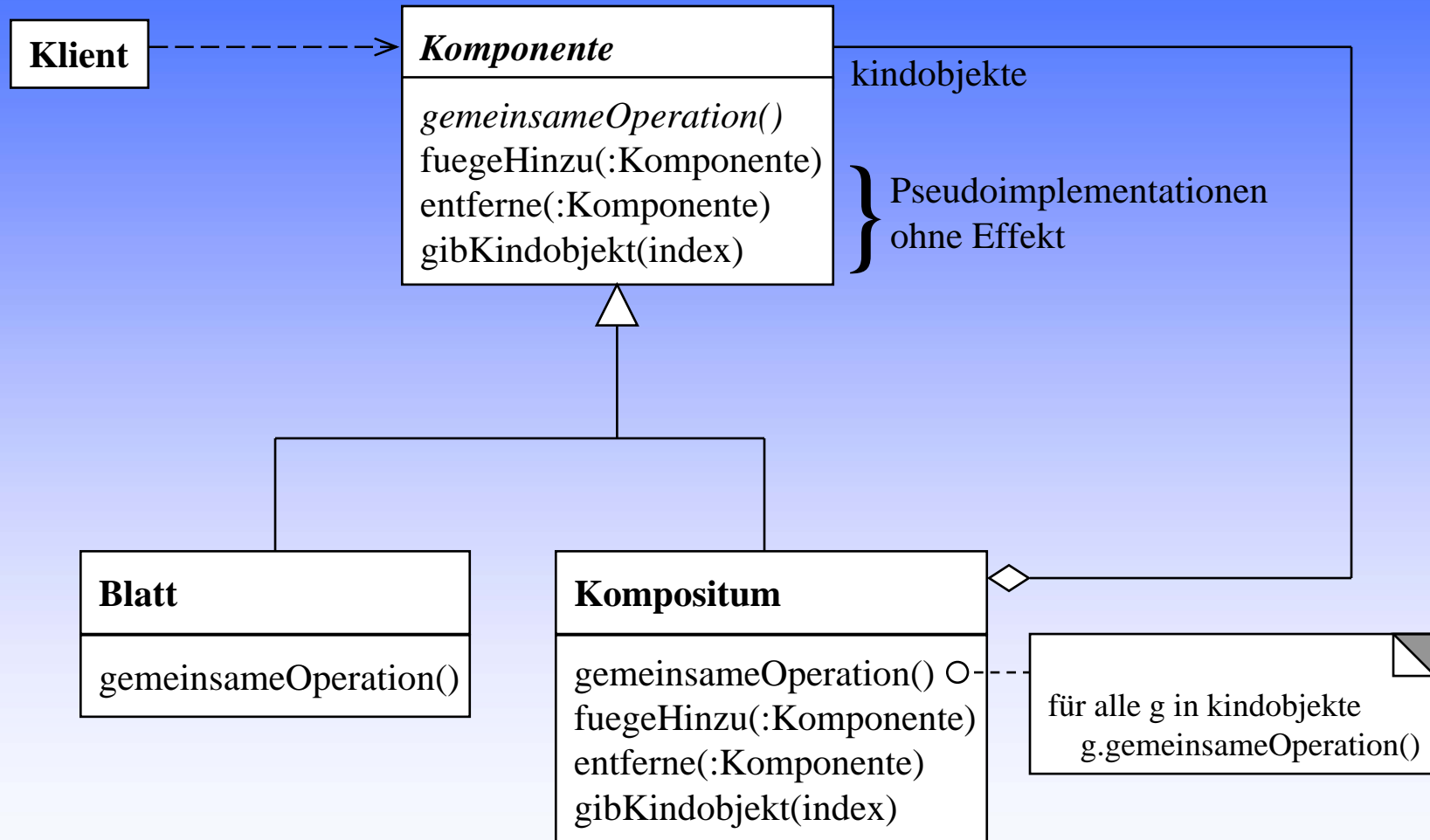


gemeinsame Operationen: zeichne(), verschiebe(), lösche(), skaliere()



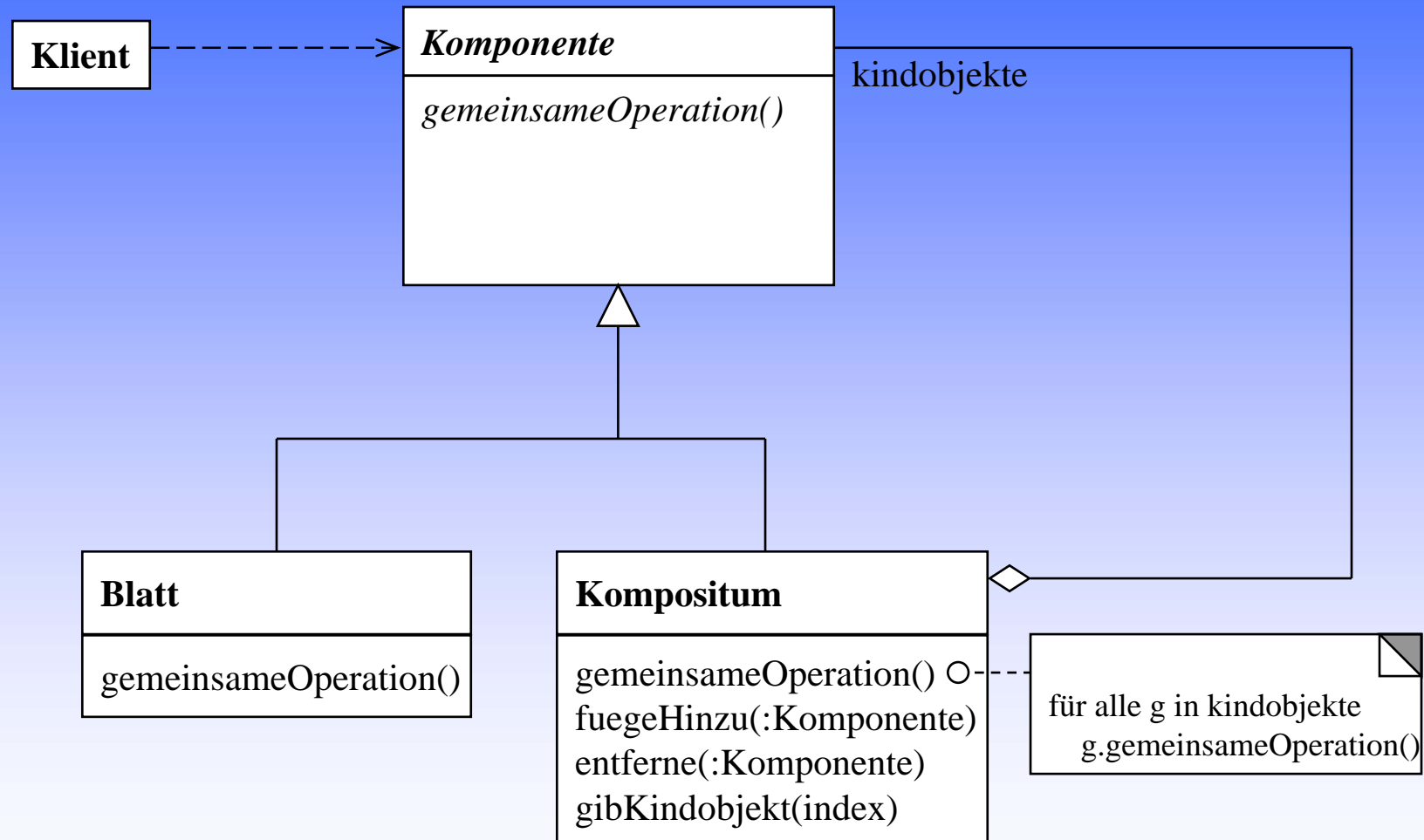
Struktur für Kompositum 1

Kompositum-Operationen in der Komponente



Struktur für Kompositum 2

Kompositum-Operationen im Kompositum



Kompositum

Die Klasse Kompositum enthält und manipuliert die Behälter-Datenstruktur, die die Komponenten speichert.

Anwendbarkeit

- Wenn Bestands-Hierarchien von Objekten repräsentiert werden sollen.
- Wenn die Klienten in der Lage sein sollen, die Unterschiede zwischen zusammengesetzten und einzelnen Objekten zu ignorieren.

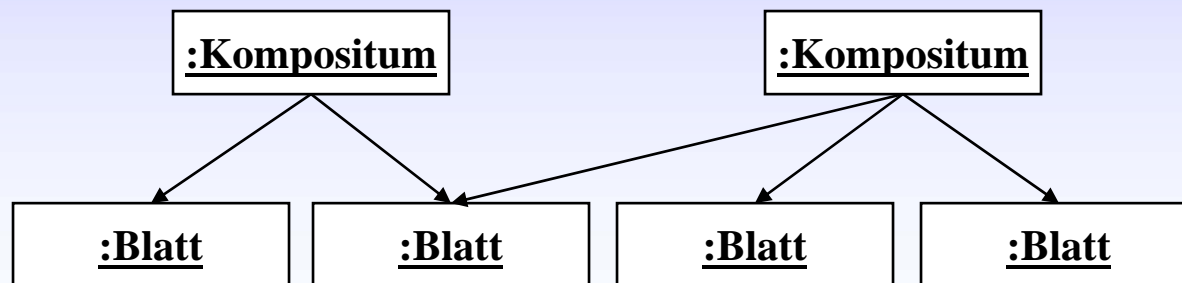


Kompositum

Implementierung

1. Es ist häufig nützlich eine Eltern-Referenz in jeder Komponente zu führen (erleichtert Traversierung). Diese Referenz kann von den **fügeHinz** und **entferne** Methoden des Kompositums gepflegt werden.

Das Teilen von Komponenten kann zu mehreren Eltern und damit zu Zweideutigkeiten führen.



Kompositum

Implementierung (Fortsetzung)

2. Maximieren der Komponenten-Schnittstelle

Die Komponenten-Schnittstelle sollte so viele gemeinsame Methoden des Kompositums und der Blätter wie möglich definieren um Transparenz zu garantieren.

Wenn Methoden des Kompositums in der Komponente definiert werden, sollte ***gibKindobjekt*** bei Blättern nichts zurückgeben – das kann auch durch eine entsprechende Implementierung in der Komponente erreicht werden.

fuegeHinzu und ***entferne*** sollten bei Blättern fehlschlagen (und einen Fehler zurückgeben oder eine Ausnahme generieren).



Kompositum

Implementierung (Fortsetzung)

3. Speichern der Kinder:

Felder, Listen oder Hash-Tabellen – abhängig von der Anwendung und benötigten Effizienz.

Bei einer festen Anzahl Kinder verwende explizite Variablen (und spezialisierte *fuegeHinzu* / *entferne* / *gibKindobjekt* Operationen).

Die Kinder müssen unter Umständen in einer bestimmten Reihenfolge gelassen werden (beispielsweise bei Anordnern (Layout Manager)).



Strategie (Strategy)

Zweck

Definiere eine Familie von Algorithmen, kapsle sie und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von nutzenden Klienten zu variieren.

Auch bekannt als

Policy

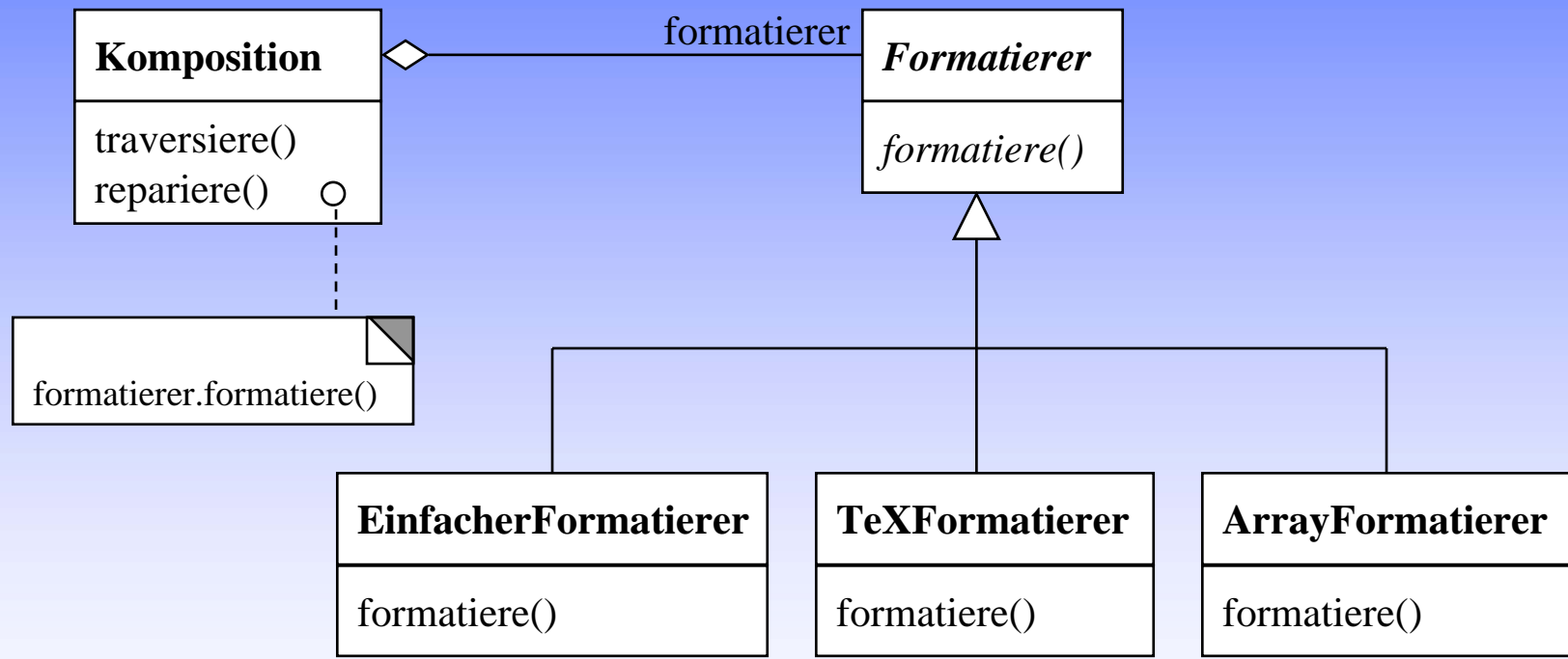
Motivation

Manchmal müssen Algorithmen, abhängig von der notwendigen Performanz, der Anzahl oder des Typs der Daten, variiert werden.

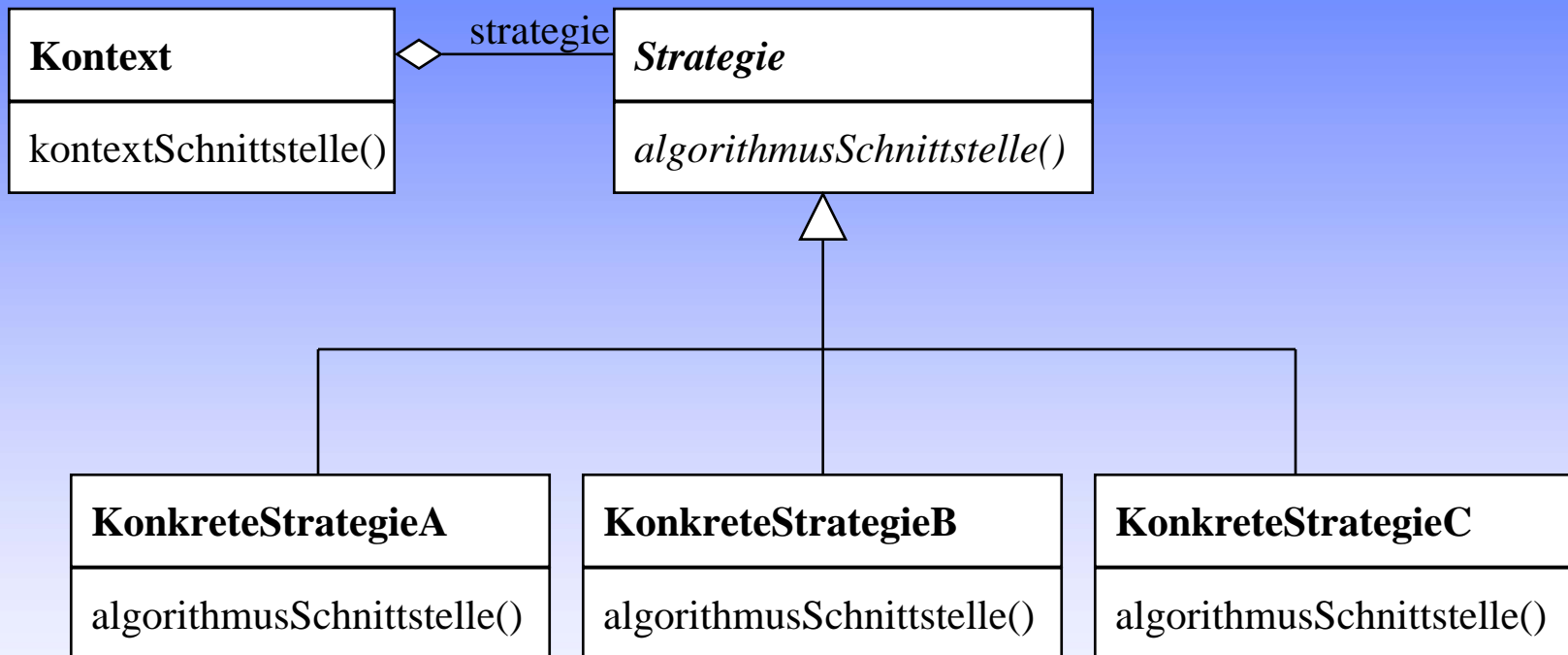


Beispiel für Strategie

Kapselung von Zeilenumbrechalgorithmen in Klassen



Struktur für Strategie



Strategie

Anwendbarkeit

- Wenn sich viele verwandte Klassen nur in ihrem Verhalten unterscheiden. Strategieobjekte bieten die Möglichkeit, eine Klasse mit einer von mehreren möglichen Verhaltensweisen zu konfigurieren.
- Wenn unterschiedliche Varianten eines Algorithmus benötigt werden.
- Wenn ein Algorithmus Datenstrukturen verwendet, die Klienten nicht bekannt sein sollen.
- Wenn eine Klasse unterschiedliche Verhaltensweisen definiert und diese als mehrfache Bedingungsanweisungen in ihren Operationen erscheinen.



Strategie

Anwendbarkeit (Fortsetzung)

- Alternativ zur Ableitung der Klasse *Strategie* kann man auch die Klasse *Kontext* ableiten, um verschiedene Verhaltensmuster zu implementieren. Das Ergebnis sind viele Klassen, die sich nur im Verhalten unterscheiden, welches für jede Klasse fest ist. Das Strategie-Muster erlaubt demgegenüber auch eine dynamische Veränderung des Verhaltens.



Wo finde ich mehr über Entwurfsmuster?

- “Design Patterns”, Gamma et al, Addison Wesley, 1995. Deutsche Ausgabe: “Entwurfsmuster”, Riehle.
- “A System of Patterns”, Buschmann et al, Wiley, 1996
- “Pattern Languages of Program Design”, conference proceedings, Addison-Wesley, 1995, 1996, 1998, ...
- “Pattern Hatching”, John Vlissides, Addison-Wesley, 1998. Deutsch: “Entwurfsmuster anwenden”, 1999.
- “Patterns in Java”, Mark Grand, Wiley, 1998.
- “Pattern-Oriented Software Architecture”, Schmidt et al, Wiley, 2000.



Zusammenfassung

Entwurfsmuster

- bieten ein Vokabular zur effizienten Kommunikation zwischen Teammitgliedern,
- erfassen den „Stand der Kunst“,
- dokumentieren Entwürfe,
- machen aus Anfängern gute Entwerfer,
- machen gute Entwerfer noch besser,
- verbessern Qualität und Produktivität,
- sind schwierig zu finden und zu beschreiben.



Entwurfsmuster - Kategorien

1. Entkoppelungs-Muster
2. Varianten-Muster
3. Zustandshandhabungs-Muster
4. Steuerungs-Muster
5. Virtuelle Maschinen
6. Bequemlichkeits-Muster



Entwurfsmuster-Kategorien nach Verwendungszweck

1. Entkopplungs-Muster: Entkopplungs-Muster teilen ein System in mehrere Einheiten, so dass einzelne Einheiten unabhängig voneinander erstellt, verändert, ausgetauscht und wiederverwendet werden können. Der Vorteil von Entkopplung ist, dass ein System durch lokale Änderungen verbessert, angepasst und erweitert werden kann, ohne das ganze System zu modifizieren.

Mehrere der Entkopplungsmuster enthalten ein Kopplungsglied, das entkoppelte Einheiten über eine Schnittstelle kommunizieren lässt. Diese Kopplungsglieder sind auch für das Koppeln unabhängig erstellter Einheiten brauchbar.



Entwurfsmuster-Kategorien nach Verwendungszweck

2. Varianten-Muster: In Mustern dieser Gruppe werden Gemeinsamkeiten von verwandten Einheiten aus ihnen herausgezogen und an einer einzigen Stelle beschrieben. Aufgrund ihrer Gemeinsamkeiten können unterschiedliche Komponenten im gleichen Programm danach einheitlich verwendet werden, und Wiederholungen desselben Codes werden vermieden.

3. Zustandshandhabungs-Muster: Die Muster dieser Kategorie bearbeiten den Zustand von Objekten, unabhängig von deren Zweck.



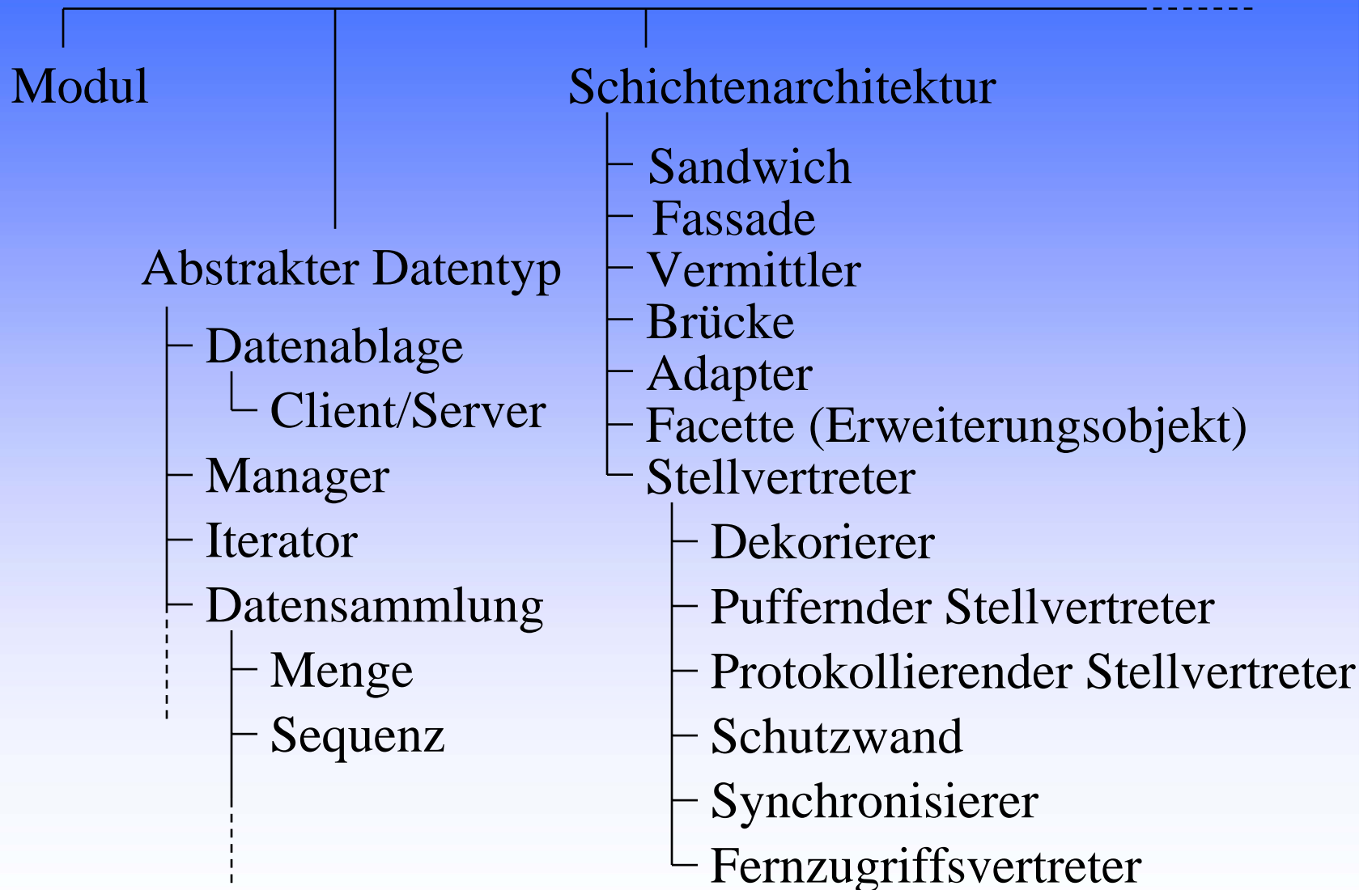
Entwurfsmuster-Kategorien nach Verwendungszweck

4. Steuerungs-Muster: Muster dieser Gruppe steuern den Kontrollfluss. Sie bewirken, dass zur richtigen Zeit die richtigen Methoden aufgerufen werden.

5. Virtuelle Maschinen: Virtuelle Maschinen erhalten Daten und ein Programm als Eingabe und führen das Programm selbständig an den Daten aus. Virtuelle Maschinen sind in Software, nicht in Hardware implementiert.



1. Entkopplungs-Muster



1. Entkopplungs-Muster (Fortsetzung)

Fließband

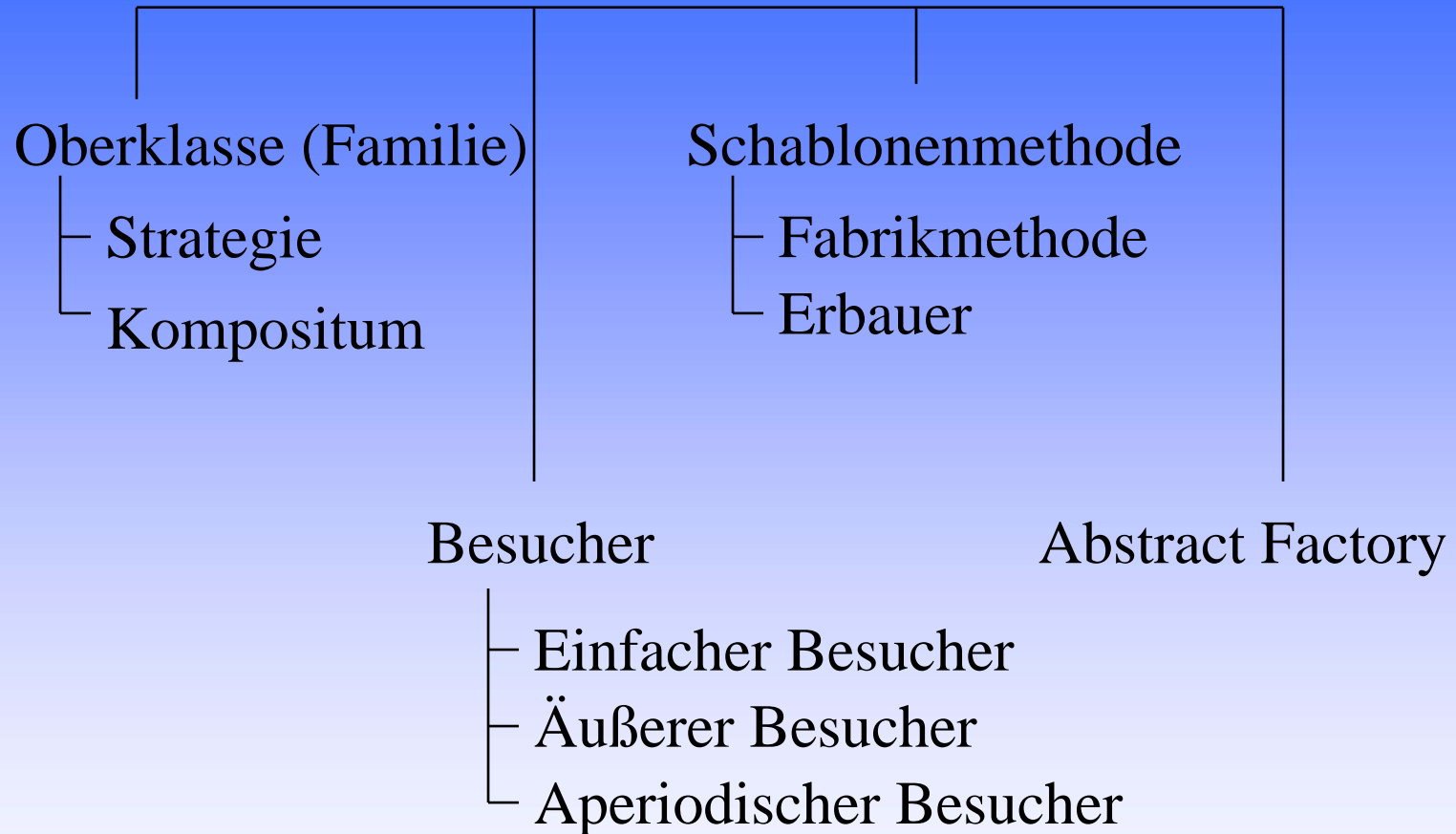
Rahmenarchitektur

Ereignissteuerung

- Ereignisbehandler (auslösen und auffangen)
- Rückruf
- Ereignis-Schleife
- Ereigniskanal
- Propagierer
 - Genauer Propagierer mit/ohne Fehlerbeh.
 - Fauler Propagierer
 - Anpassungsfähiger Propagierer
 - Beobachter



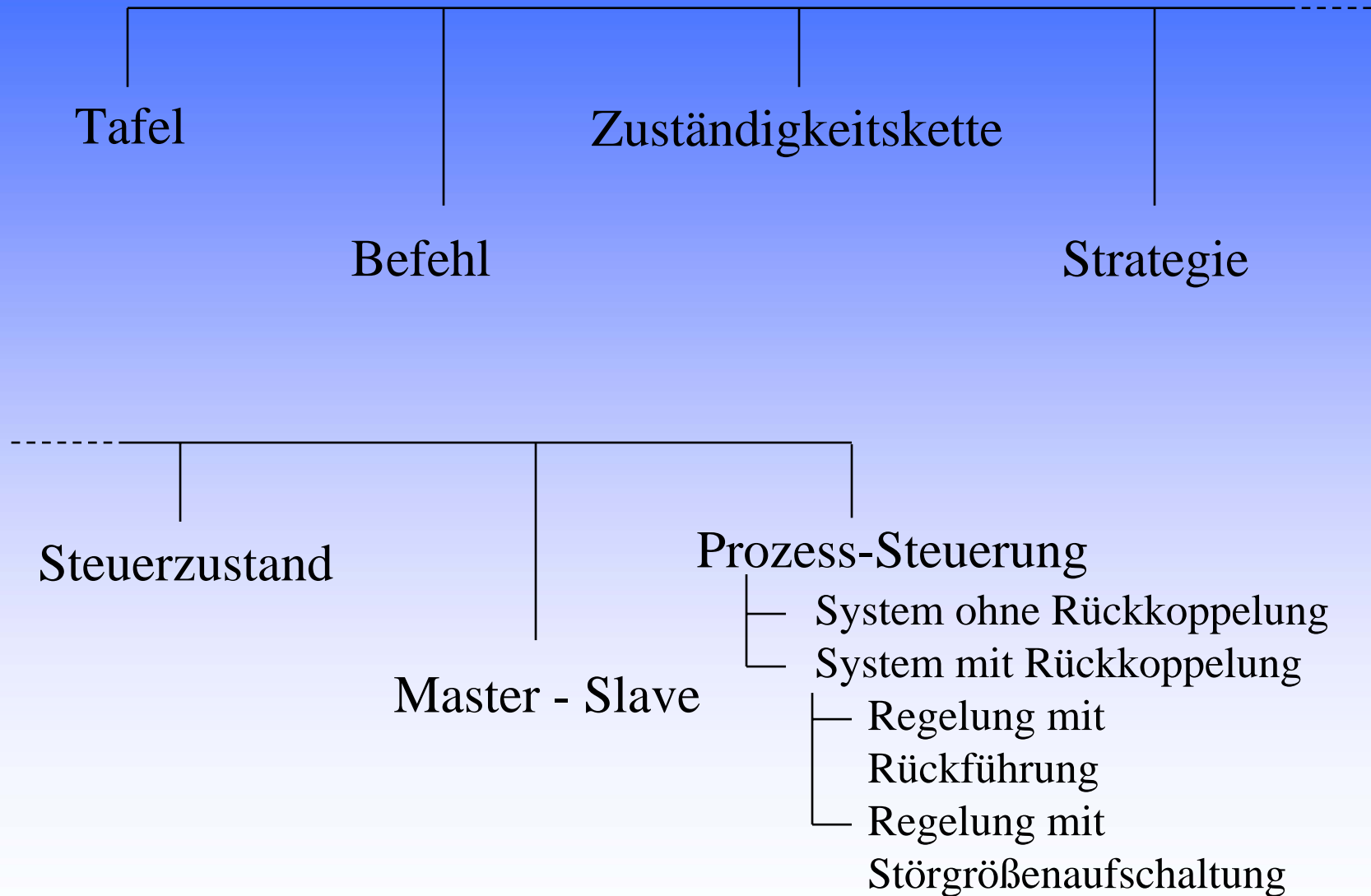
2. Varianten-Muster



3. Zustandshandhabungs-Muster



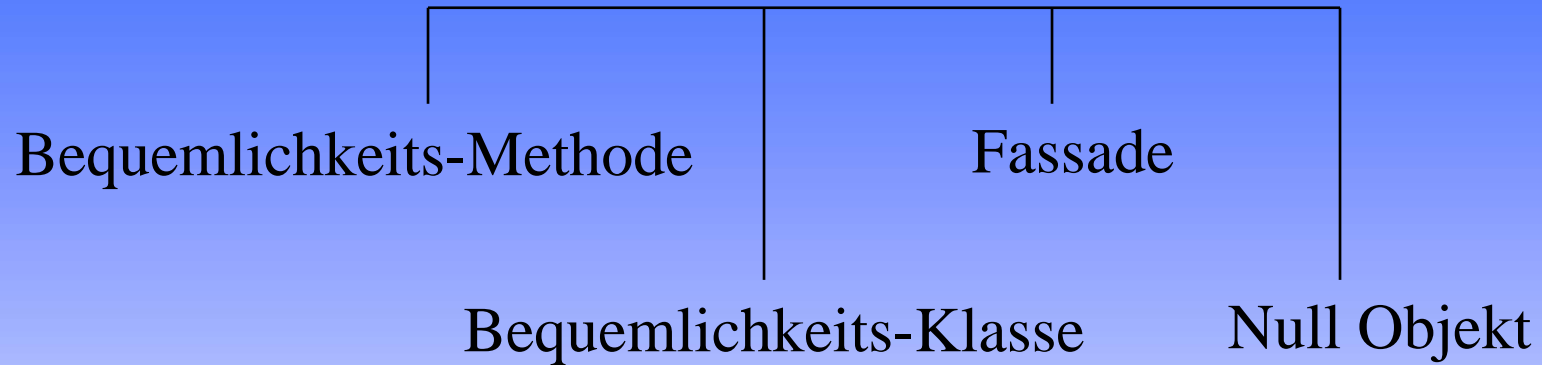
4. Steuerungs-Muster



5. Virtuelle Maschinen



6. Bequemlichkeits-Muster



Entwurfsmuster-Katalog

Es folgen nun die einzelnen Entwurfsmuster-Beschreibungen.



1. Entkoppelungs-Muster

- Abstrakter Datentyp
- Modul
- Datenablage (Repository)
- Iterator
- Schichtenarchitektur
- Vermittler (Mediator)
- Brücke
- Adapter
- Stellvertreter (Proxy)
- Fließband
- Ereigniskanal
- Rahmenprogramm (Framework)



Abstrakter Datentyp

Ein abstrakter Datentyp (ADT) definiert einen neuen Datentyp zusammen mit geeigneten Operationen. Die Implementierung dieses Datentyps ist wie beim Modul hinter einer abstrakten (änderungs-unempfindlichen) Schnittstelle verborgen.

Unterschiede zum Modul:

- Modul ist i.d.R. eine größere Einheit. Es kann z.B. mehrere voneinander abhängige ADTs zusammenfassen. (Beispiel: eine Aggregatsklasse mit zugehörigem Iterator.)
- Von einem ADT kann man beliebig viele Exemplare anlegen; von einem Modul gibt es in jedem Programm nur ein Exemplar.



Modul

Ein Modul ist eine Menge von Programmkomponenten, die gemeinsam entworfen und verändert werden; diese Komponenten werden hinter einer änderungsunempfindlichen Schnittstelle verborgen (“Geheimnisprinzip”).

Ziel der Modularisierung ist eine Entkopplung: modul-interne Komponenten können verändert oder ersetzt werden, ohne die Benutzer des Moduls anpassen zu müssen.

Um effektiv zu sein, müssen die möglichen Änderungen vorausgesehen und in die Modulstruktur und Schnittstellen hineingeplant werden.



Modul

Kandidaten für Veränderung/Verbergung:

- Datenstrukturen und Operationen, Größe der Datenstrukturen, Optimierungen
- maschinennahe Details
- betriebsystemnahe Details
- Ein/Ausgabeformate
- Benutzerschnittstellen
- Dialog- und Fehlermeldungstexte, Maßeinheiten (Internationalisierung)
- Reihenfolge der Verarbeitung, Vorverarbeitung, inkrementelle Verarbeitung
- Zwischenpufferung



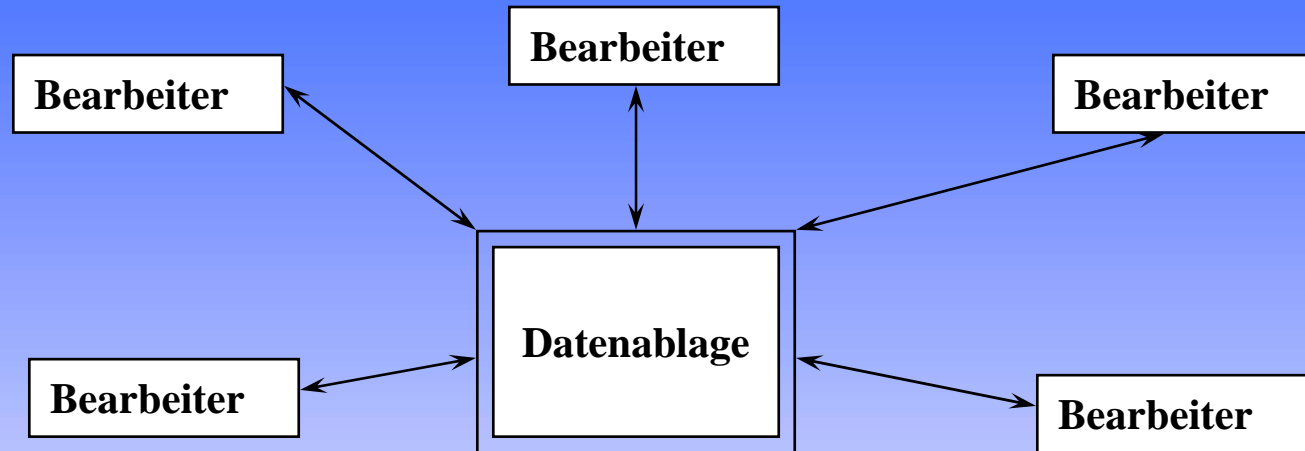
Datenablage (Repository)

Zweck

Eine Menge unabhängiger Komponenten kommunizieren über eine zentrale Ablage, in dem sie Elemente in dieser Datenstruktur ablegen oder aus ihr herausholen.



Struktur der Datenablage



Beispiele für große Datenablagen: Datenbanken, Hypertextsysteme. Zusätzliche Mechanismen gegenüber einfachen Datenablagen: Persistenz, Zugriffskontrolle, Transaktionsverwaltung.



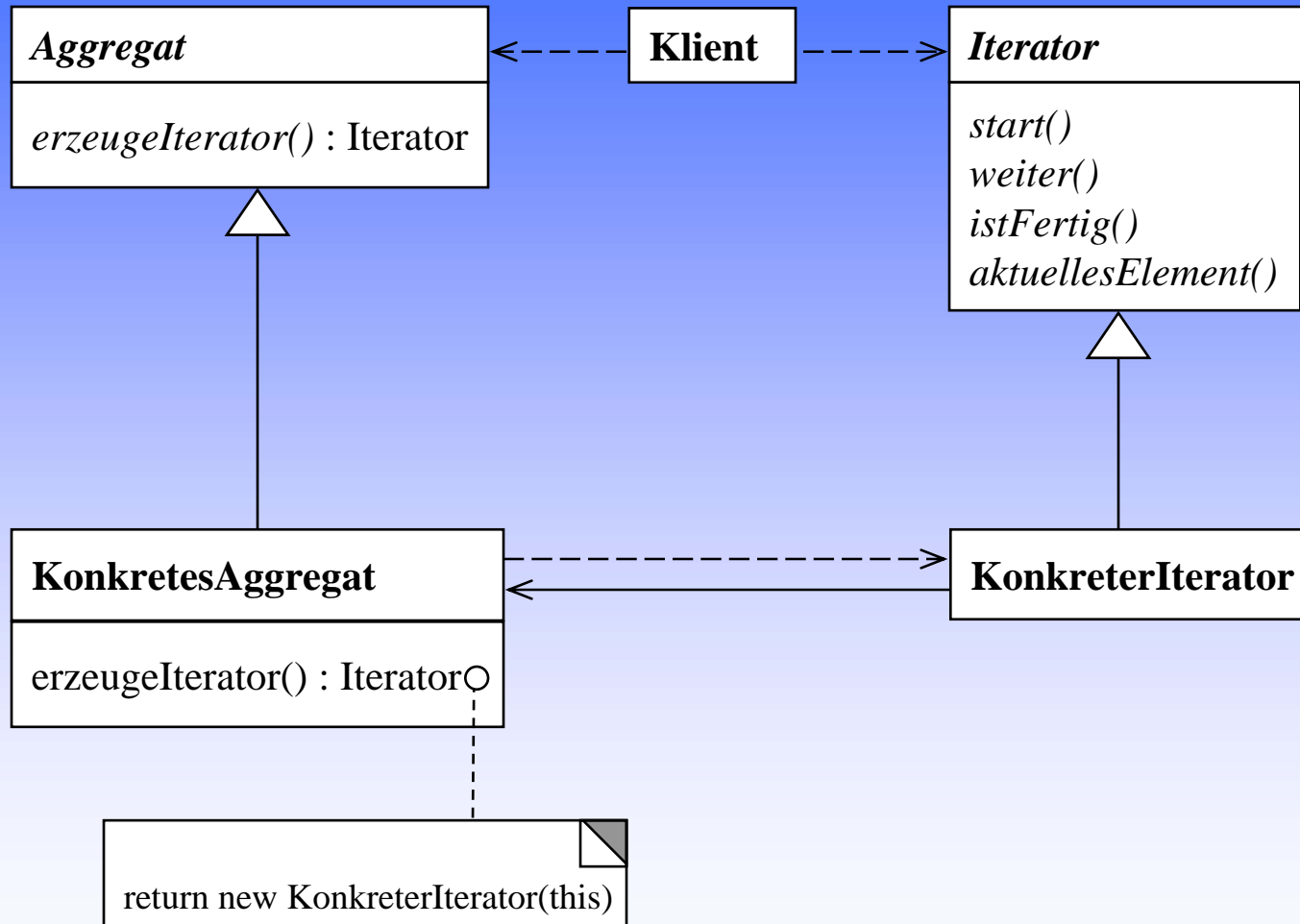
Iterator (Iterator)

Zweck

Ermögliche den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen.

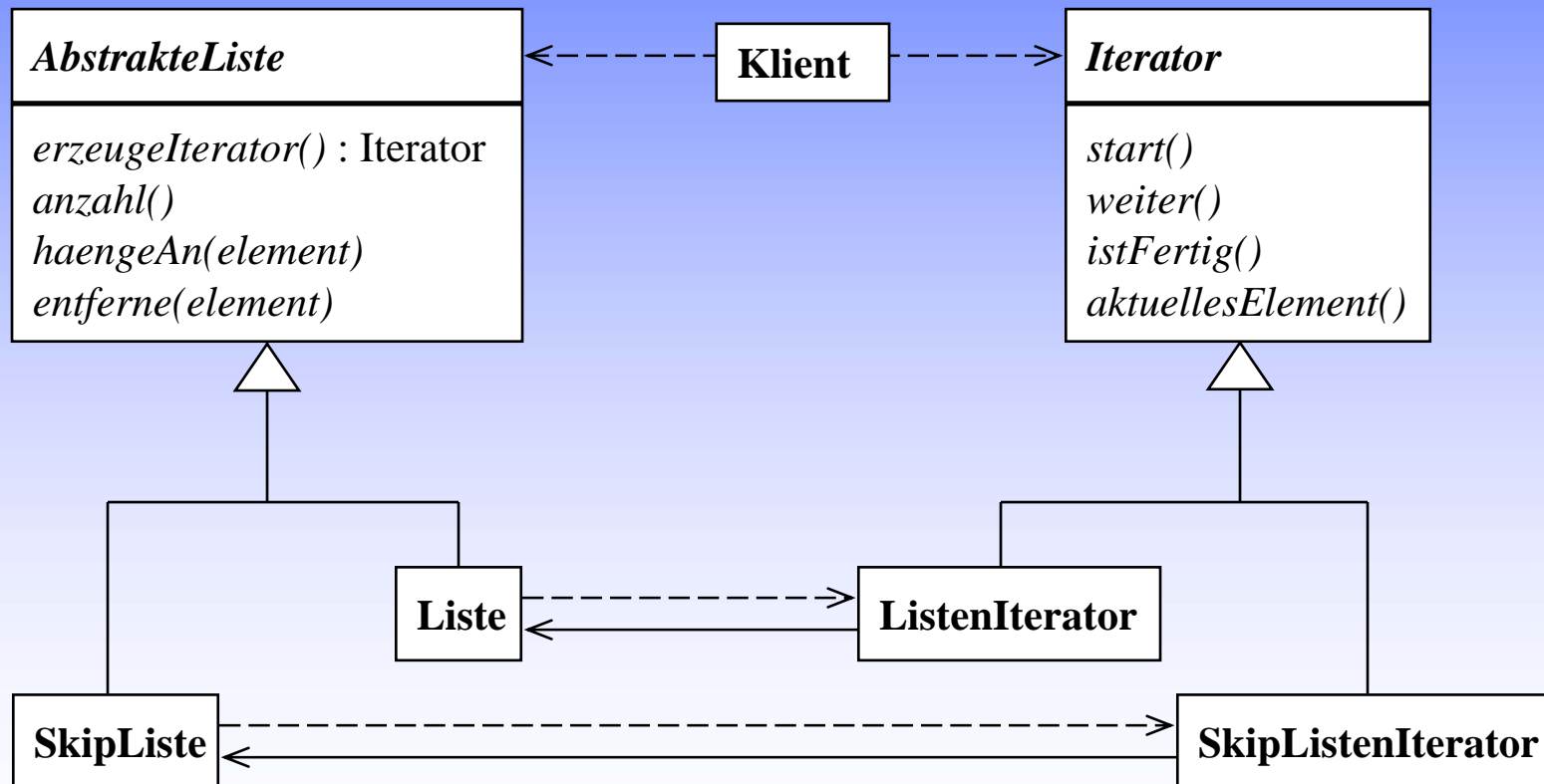


Struktur des Iterators



Beispiel für Iterator

Trennen von Listenimplementierungen und
Listentraversierungs-Implementierungen



Iterator

Anwendbarkeit

- Um den Zugriff auf den Inhalt eines zusammengesetzten Objekts zu ermöglichen, ohne dabei seine interne Struktur offenzulegen.
- Um mehrfache gleichzeitige Traversierungen auf zusammengesetzten Objekten zu ermöglichen.
- Um eine einheitliche Schnittstelle zur Traversierung unterschiedlicher zusammengesetzter Strukturen anzubieten (das heißt, um polymorphe Iteration zu ermöglichen).



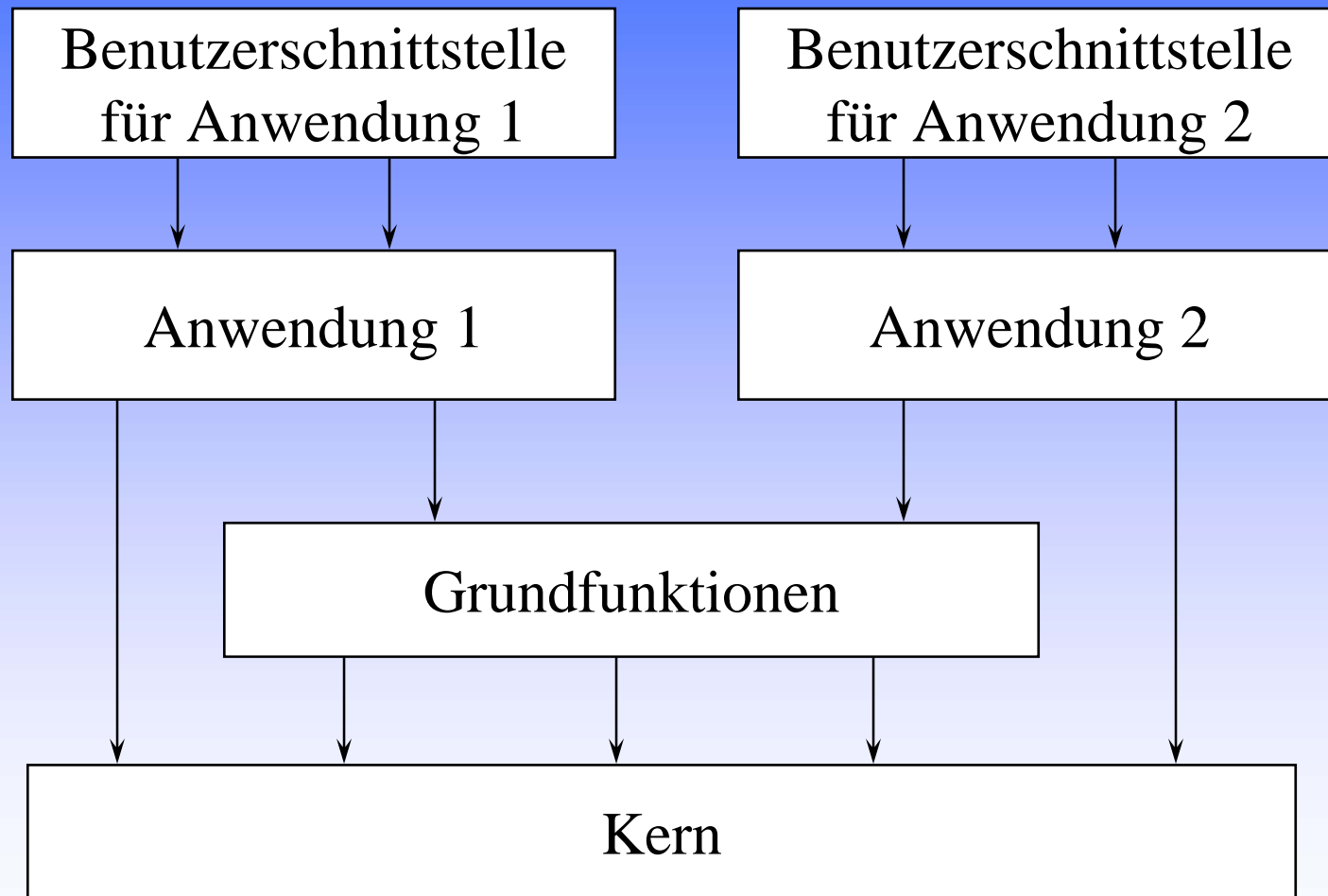
Schichtenarchitektur

Zweck

Gliedere ein System in eine hierarchisch geordnete Menge von Schichten. Eine Schicht besteht aus einer Menge von Software-Komponenten mit einer wohldefinierten Schnittstelle, nutzt die darunterliegenden Schichten als Klient und stellt seine Dienste an darüberliegende Schichten zur Verfügung.



Struktur der Schichtenarchitektur



Schichtenarchitektur

Beispiele

- Mikrokerne (Betriebssysteme)
- Protokolltürme bei der Datenfernübertragung
- Informationssysteme (auf Datenbanken aufbauend)

In manchen Systemen sind Benutzung nur zwischen aufeinanderliegenden Schichten erlaubt. Eine weitere Variante ist, dass jede Schicht neben den eigenen Komponenten nur eine sorgfältig bestimmte Untermenge der Komponenten der darunterliegenden Schicht weiterexportiert.



Schichtenarchitektur

Anwendbarkeit

- Unabhängige Entwicklung und Korrektur, Austausch von Schichten
- Schrittweiser Aufbau und schrittweises Testen
- Wiederverwendung von tieferen Schichten in anderen Konfigurationen.



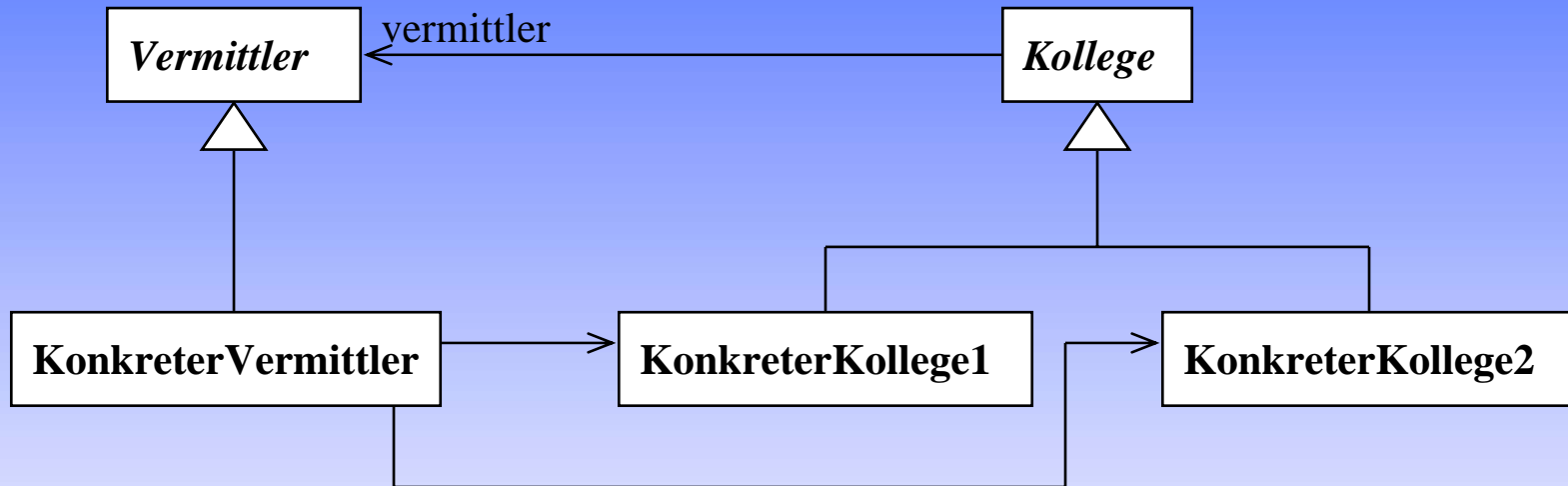
Vermittler (Mediator)

Zweck

Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es, das Zusammenspiel der Objekte unabhängig zu variieren.



Struktur des Vermittlers



Beispiel für Vermittler

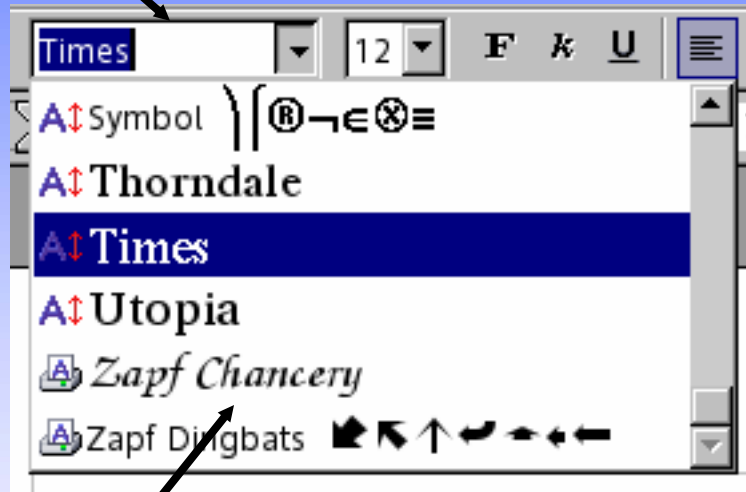
- Es gibt oft Abhängigkeiten zwischen den Elementen (Knöpfen, Menüs, Eingabefelder, etc.) einer Dialogbox. So muß z.B. ein Knopf deaktiviert sein, wenn ein bestimmtes Texteingabefeld leer ist.
- Unterschiedliche Dialogboxen besitzen unterschiedliche Abhängigkeiten zwischen Elementen.
- Individuelle Anpassung in Unterklassen ist mühsam und schlecht wiederverwendbar (zu viele Klassen).

➔ Kapseln des Gesamtverhaltens in einem Vermittlerobjekt.



Beispiel für Vermittler

Eingabefeld



ListBox

Beim Tippen rollt
Auswahlliste auf.

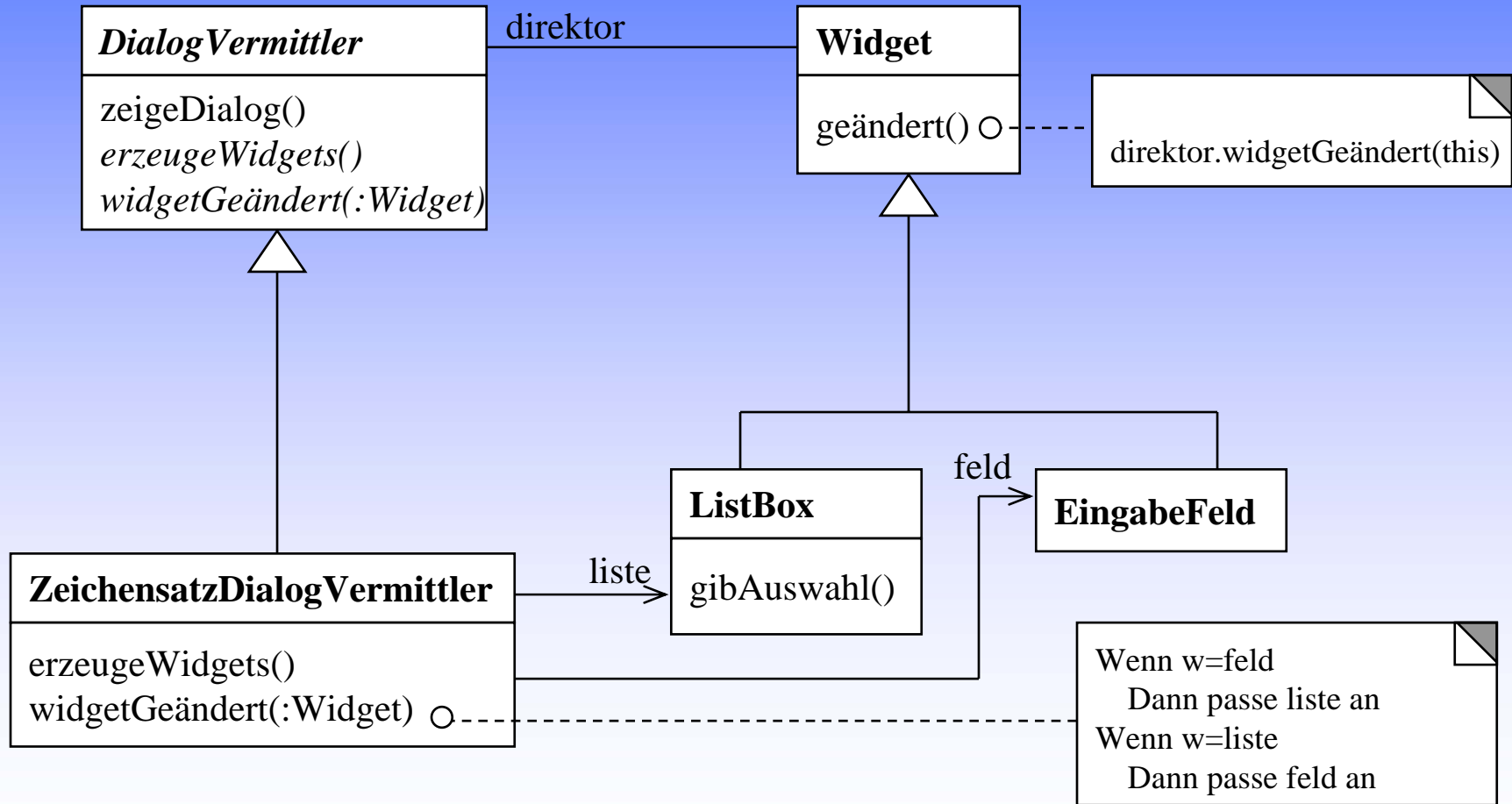
Tippen eines Buchstabens
im Eingabefeld
zeigt ersten Eintrag mit
gleichem Anfang.

Ausgewähltes Element
erscheint im Eingabefeld.



Beispiel für Vermittler

Fenster mit Zeichensatz-Dialog



Vermittler

Anwendbarkeit

- Wenn eine Menge von Objekten vorliegt, die in wohl-definierter, aber komplexer Weise miteinander zusammenarbeiten. Die sich ergebenden Abhängigkeiten sind unstrukturiert und schwer zu verstehen.
- Wenn die Wiederverwertung eines Objektes schwierig ist, weil es sich auf viele andere Objekte bezieht und mit ihnen zusammenarbeitet.
- Wenn ein auf mehrere Klassen verteiltes Verhalten maßgeschneidert werden soll, ohne viele Unterklassen bilden zu müssen.



Brücke (Bridge)

Zweck

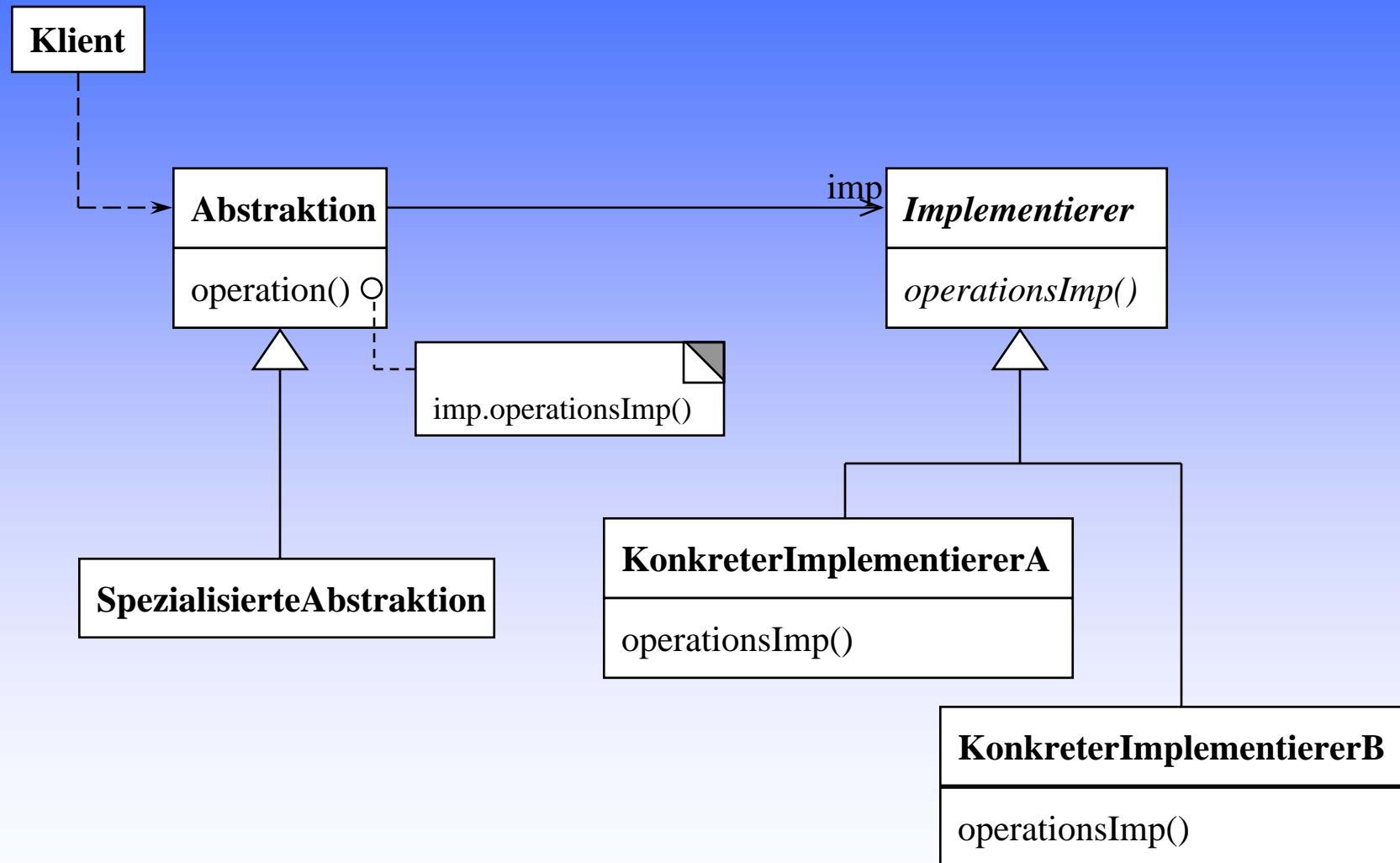
Entkopple eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.

Auch bekannt als

Handle/Body

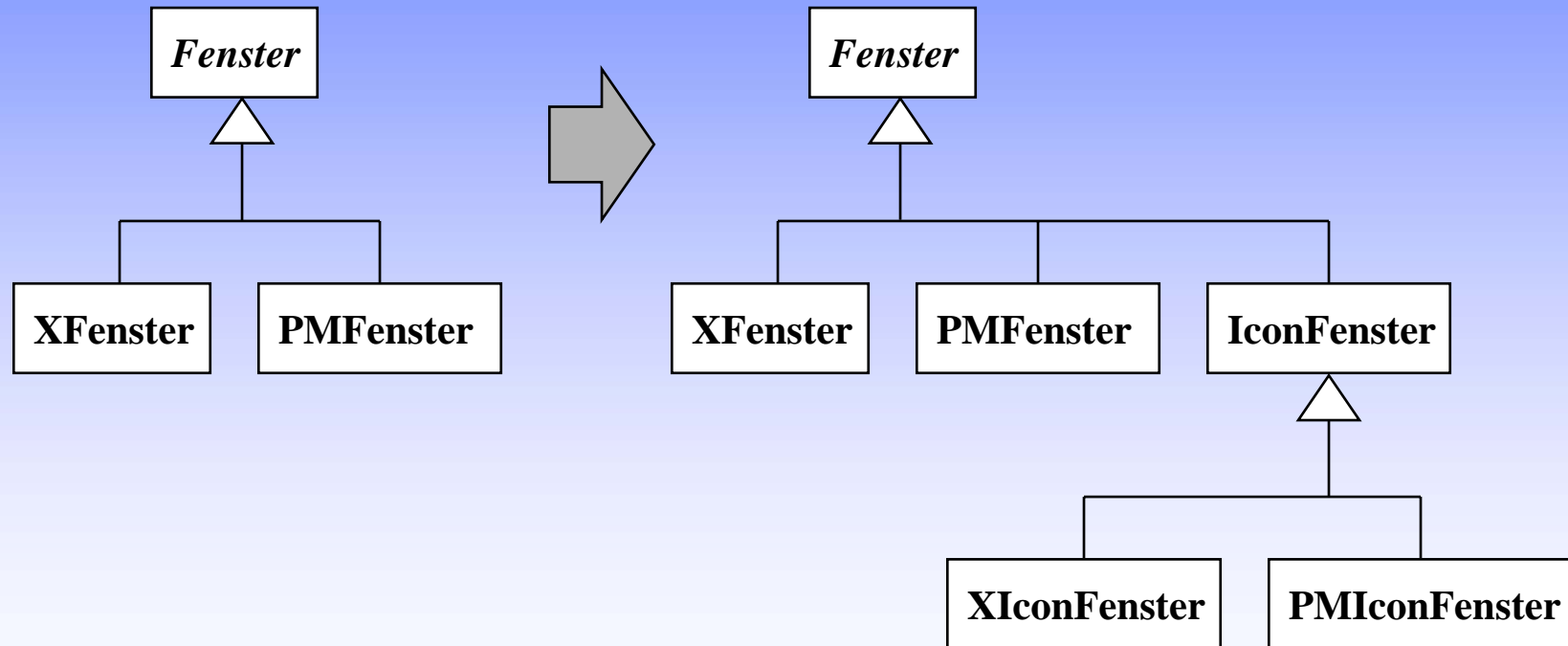


Struktur der Brücke

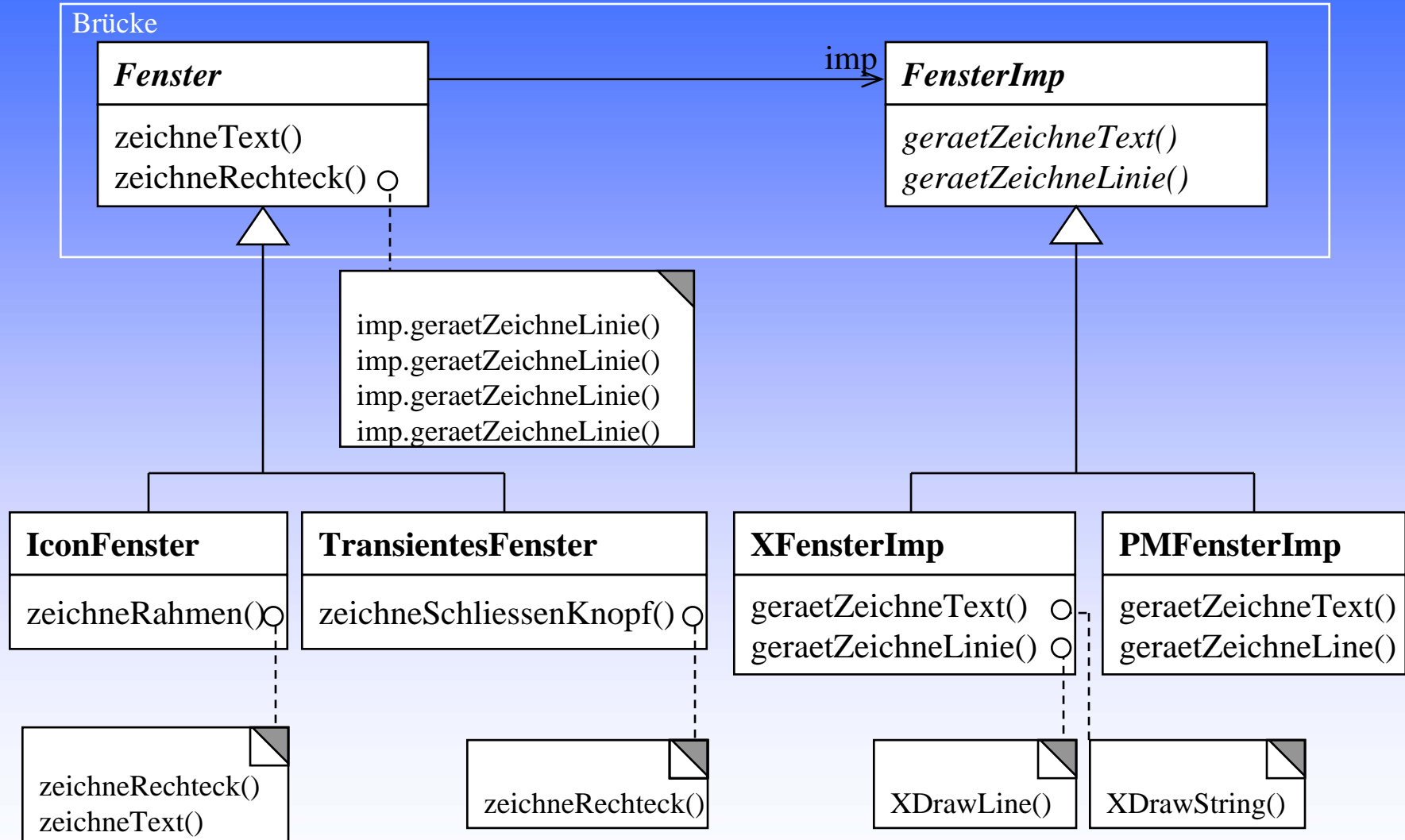


Beispiel für Brücke

Sowohl Implementierung als auch Abstraktion bei Benutzerschnittstellen: Die Bildung von Unterklassen führt zu vielen Klassen, die beide Aspekte vermischen:



Beispiel für Brücke



Brücke

Anwendbarkeit

- Wenn eine dauerhafte Verbindung zwischen Abstraktion und Implementierung vermieden werden soll.
- Wenn sowohl Abstraktion als auch Implementierungen durch Unterklassenbildung erweiterbar sein soll.
- Wenn Änderungen in der Implementierung einer Abstraktion keine Auswirkung auf Klienten haben sollen.
- Wenn die Implementierung einer Abstraktion vollständig vom Klienten versteckt werden soll.
- Wenn eine starke Vergrößerung der Anzahl der Klassen vermieden werden soll (siehe Beispiel).
- Wenn eine Implementierung von mehreren Objekten aus gemeinsam benutzt werden soll.



Adapter (Adapter)

Zweck

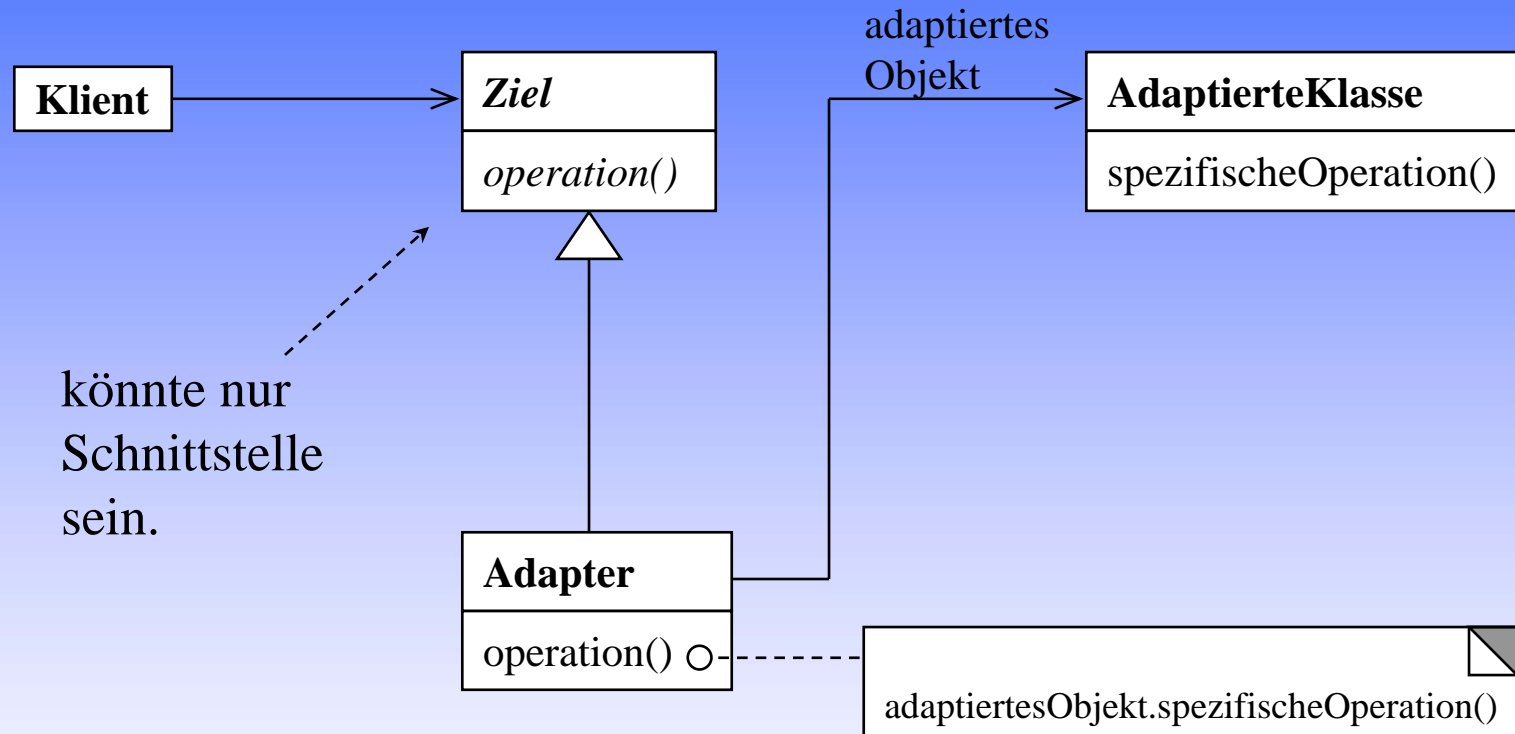
Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster läßt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten dazu nicht in der Lage wären.

Auch bekannt als

Wandler, Umwickler (Wrapper)



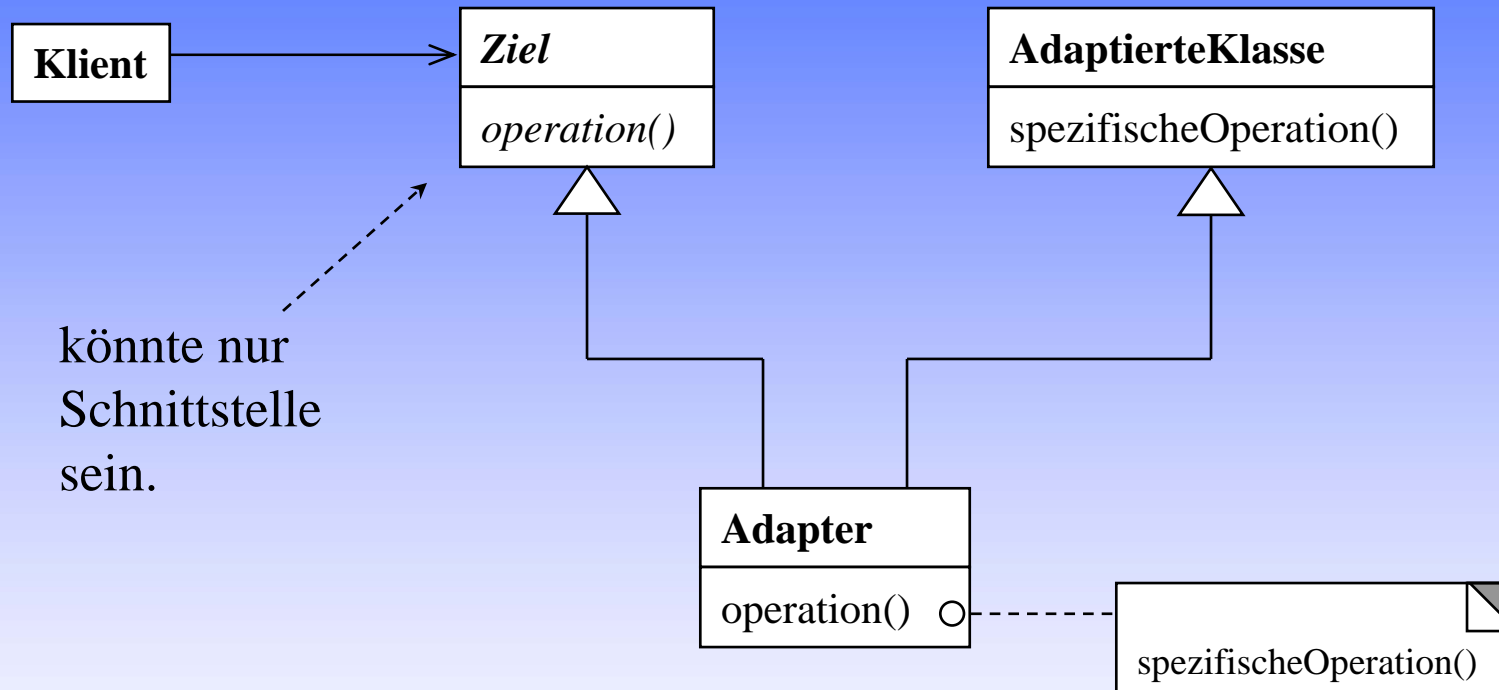
Struktur des Adapters (1)



ohne Mehrfachvererbung (Objektadapter)



Struktur des Adapters (2)



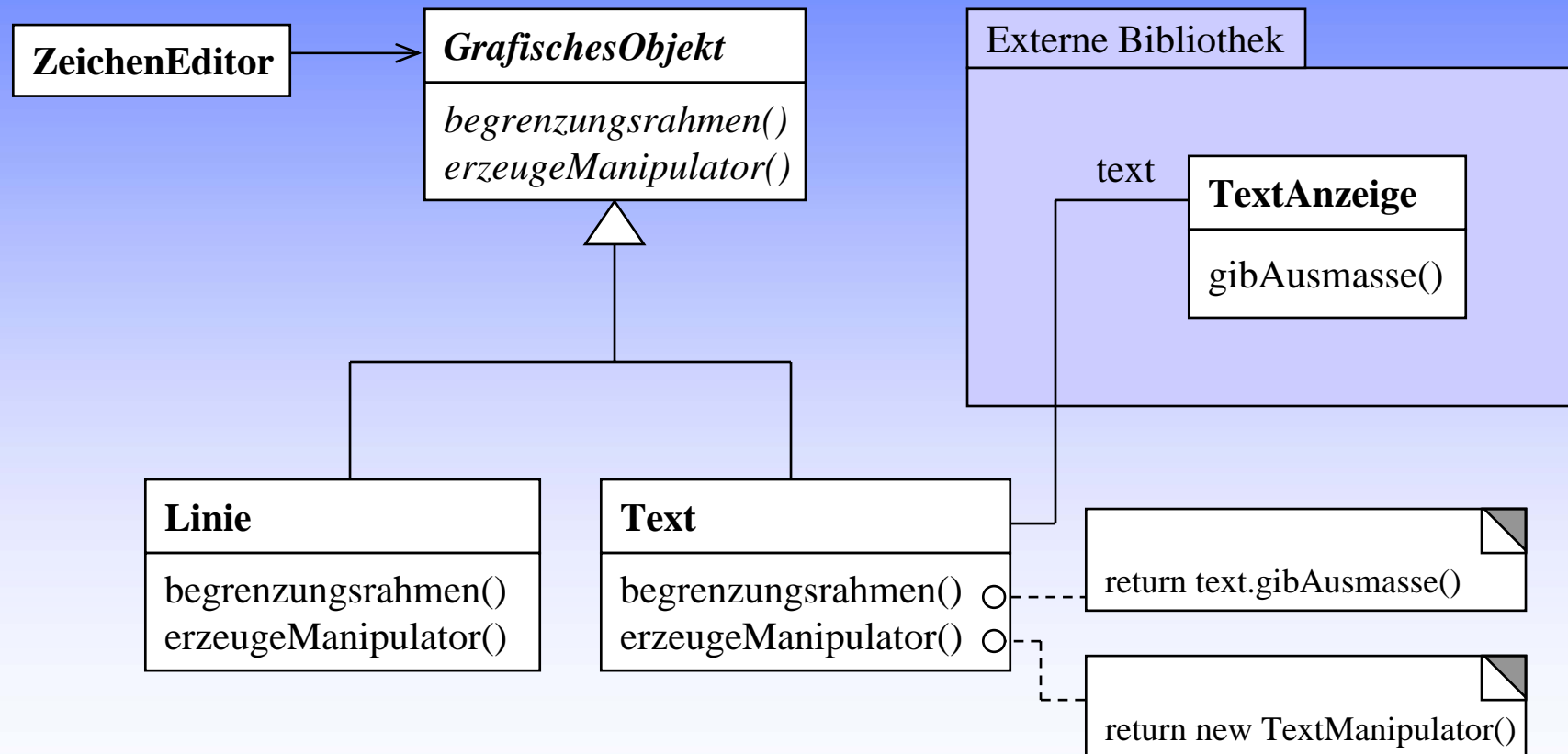
könnte nur
Schnittstelle
sein.

mit Mehrfachvererbung (Klassenadapter)



Beispiel für Adapter

Verwendung einer externen Klassenbibliothek zur Anzeige von Texten in einem Zeicheneditor.



Adapter

Anwendbarkeit

- Wenn eine existierende Klasse verwendet werden soll, deren Schnittstelle aber nicht der benötigten Schnittstelle entspricht.
- Wenn eine wieder verwendbare Klasse erstellt werden soll, die mit unabhängigen oder nicht vorhersehbaren Klassen zusammenarbeitet, d.h. Klassen die nicht notwendigerweise kompatible Schnittstellen besitzen.
- Wenn verschiedene existierende Unterklassen benutzt werden sollen, es aber unpraktisch ist, die Schnittstellen jeder einzelnen Unterklasse durch Ableiten anzupassen. Ein Objektadapter ist in der Lage, die Schnittstelle seiner Oberklasse anzupassen.



Stellvertreter (Proxy)

Zweck

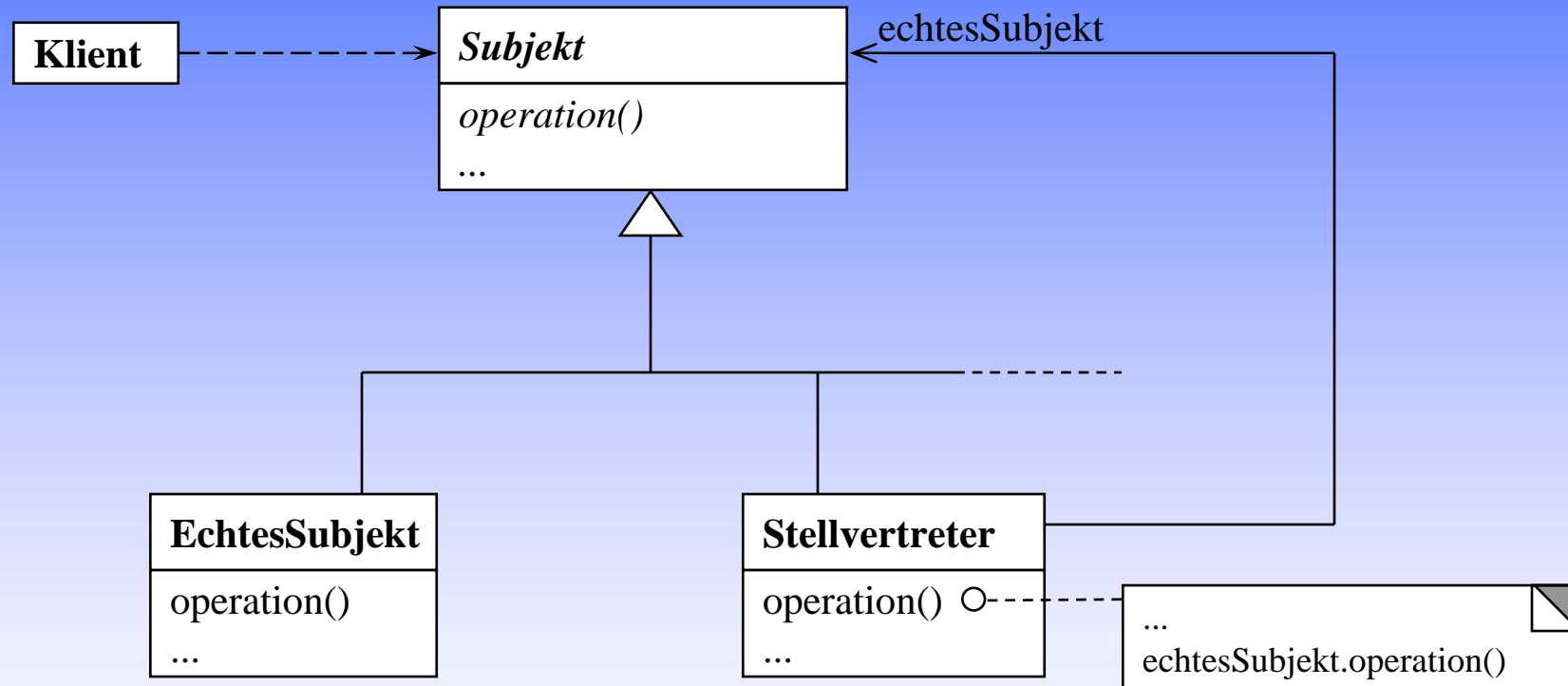
Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.

Auch bekannt als

Surrogat



Struktur des Stellvertreters



Stellvertreter

Anwendbarkeit

Das Stellvertretermuster ist anwendbar, sobald es den Bedarf nach einer anpassungsfähigeren und intelligenteren Referenz auf ein Objekt als einen einfachen Zeiger gibt. Es folgen einige verbreitete Situationen, in denen das Stellvertretermuster anwendbar ist:

1. Ein **protokollierender Stellvertreter** zählt Referenzen auf das eigentliche Objekt, so dass es automatisch freigegeben werden kann, wenn keine Referenzen mehr auf das Objekt existieren. Er kann auch andere Zugriffsinformationen protokollieren und leitet Zugriffe weiter.



Stellvertreter

Anwendbarkeit (Fortsetzung)

2. Ein **puffernder Stellvertreter** lädt ein persistentes Objekt erst dann in den Speicher, wenn es das erste Mal dereferenziert wird. Er kann auch einen Puffer mit mehreren Objekten verwalten, die nach Bedarf zwischen Hintergrund- und Hauptspeicher bewegt werden.
3. Ein **Fernzugriffsvertreter** stellt einen lokalen Stellvertreter für ein Objekt in einem anderen Adressraum dar.



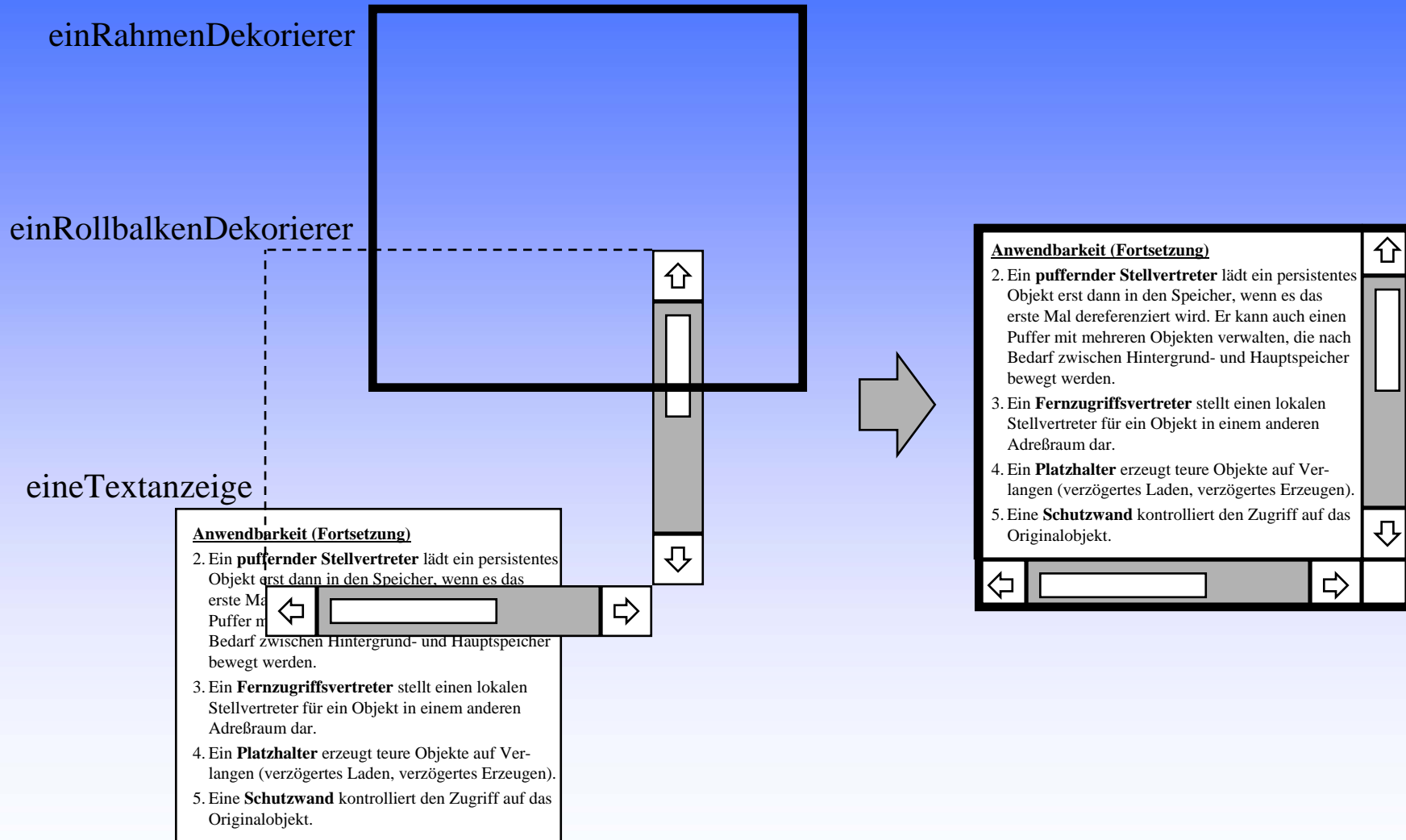
Stellvertreter

Anwendbarkeit (Fortsetzung)

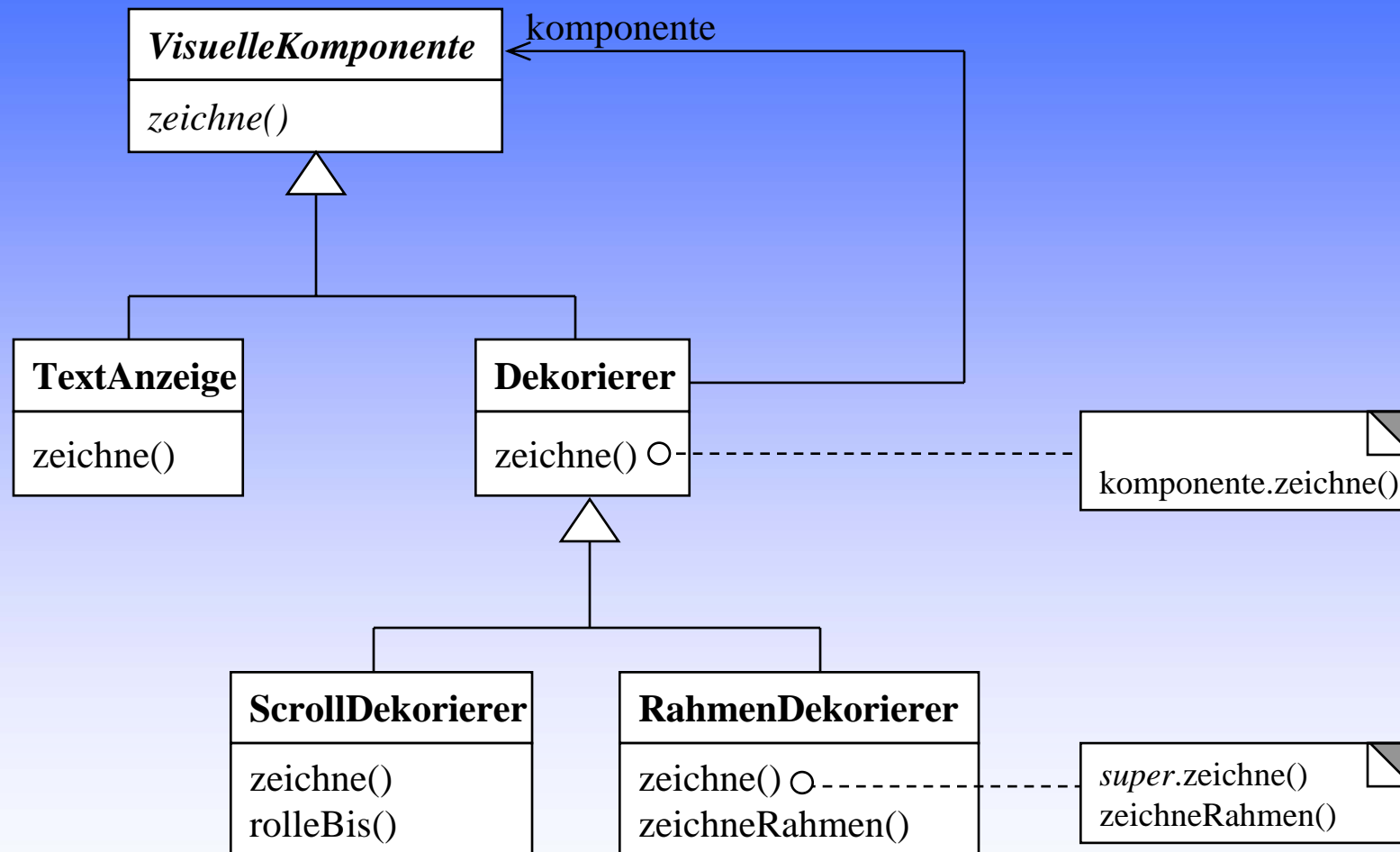
4. Ein **Platzhalter** erzeugt teure Objekte auf Verlangen (verzögertes Laden, verzögertes Erzeugen).
5. Eine **Schutzwand** kontrolliert den Zugriff auf das Originalobjekt. Schutzwände sind nützlich, wenn Objekte über verschiedene Zugriffsrechte verfügen sollen.
6. Ein **Dekorierer** fügt zusätzliche Zuständigkeiten zu einem bestehenden Objekt hinzu (möglicherweise kaskadiert).



Beispiel eines Stellvertreters (Dekorierer)



Beispiel eines Stellvertreters (Dekorierer)



Fließband (Pipes and Filters)

Zweck

Bietet eine Struktur für Systeme, die Datenströme bearbeiten. Jeder Bearbeitungsschritt ist in einer *Filter* Komponente gekapselt. Daten werden über *Kanäle* von einem *Filter* zu einem anderen weitergegeben. Eine Neukombination von Filtern ermöglicht es, Familien von Systemen zu erstellen.

Auch bekannt als

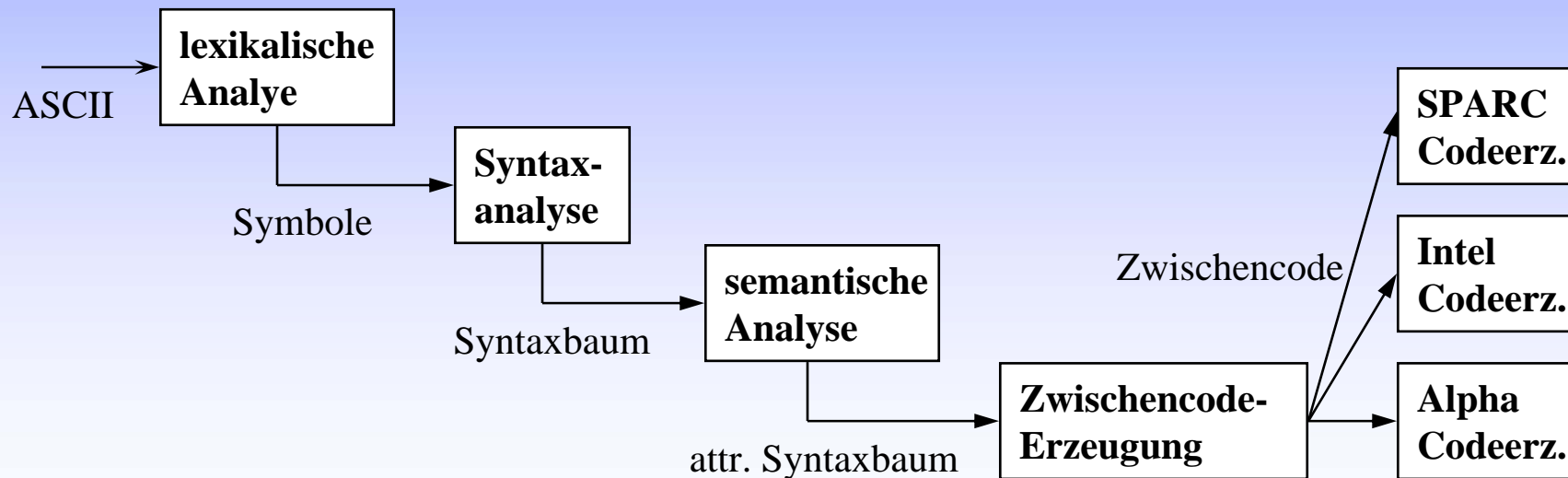
Data Flow, Kanäle und Filter



Struktur für Fließband



Beispiel: Übersetzer



Fließband

Anwendbarkeit

- Wenn ein System Datenströme bearbeiten oder transformieren muss und ein System bestehend aus nur einer Komponente unhandhabbar ist.
- Wenn in zukünftigen Entwicklungen einzelne Komponenten ersetzt oder Arbeitsschritte umgeordnet werden sollen.
- Wenn kleinere Komponenten einfacher in anderen Zusammenhängen wiederverwendet werden können.
- Wenn einzelne Komponenten parallel oder quasi-parallel ablaufen sollen.
- Nachteil: nicht geeignet für interaktive Systeme



Fließband

Beispiel: Erzeugen eines Reimwörterbuches (Wörter sind von hinten her sortiert, d.h., Wörter, die gleich enden, stehen hintereinander.)

```
reversiere < wörterbuch | sort | reversiere > reimwörterbuch
```

wörterbuch: ein vorhandenes Wörterbuch; ein Wort pro Zeile

sort: Systemsortierprogramm von Unix

reversiere: einfaches Programm zum Umkehren von Einzelzeilen

„ | “: Kanal

„ < „: Eingabe von Datei; „ > „: Ausgabe in Datei



Ereigniskanal (Event Channel)

Zweck

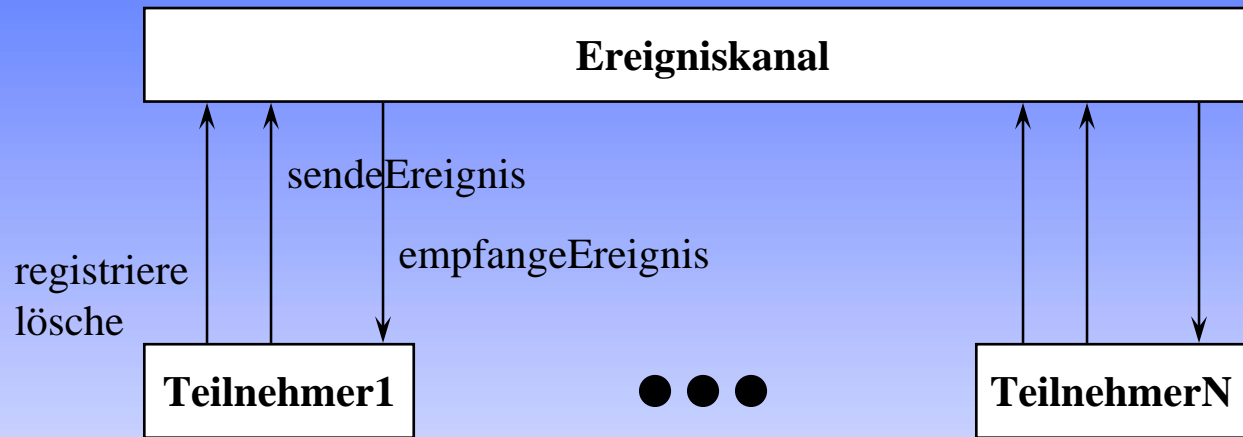
Entkopple Teilnehmer an einem Gesamtsystem vollständig voneinander, so dass sie völlig eigenständig arbeiten können und über die Existenz oder Anzahl anderer Teilnehmer nichts wissen. Interaktionen erfolgen über Ereignisse.

Auch bekannt als

Ereignissteuerung



Struktur des Ereigniskanal

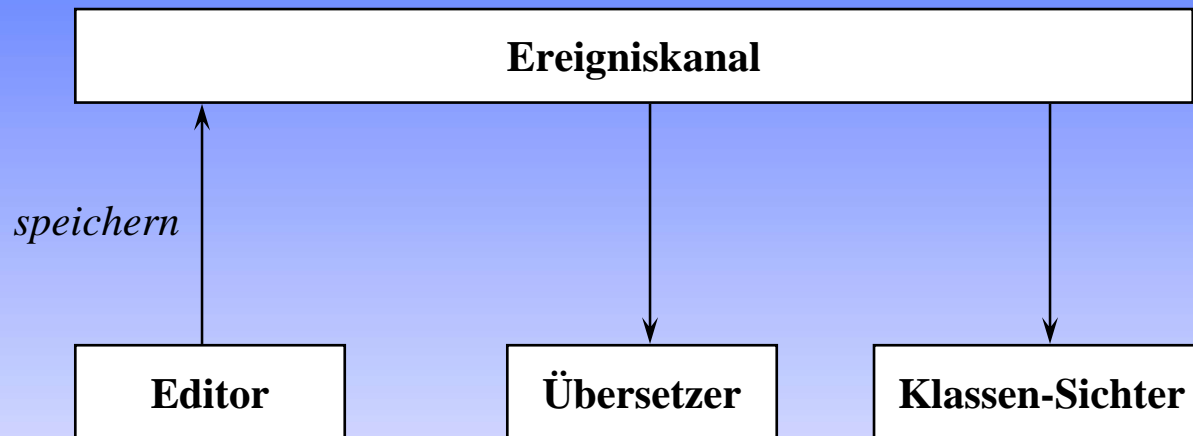


Teilnehmer registrieren sich am Ereigniskanal, in dem sie angeben, bei welchen Ereignissen sie benachrichtigt werden sollen. Wenn ein Teilnehmer ein Ereignis (evtl. mit Daten) an den Ereigniskanal sendet, leitet dieser das Ereignis an die dafür registrierten Teilnehmer weiter.



Beispiel eines Ereigniskanals

Eine Programmierumgebung



Wenn der Editor eine Datei speichert, wird ein Ereignis (*speichern*) generiert, welches den Übersetzer startet und den Klassen-Sichter dazu bringt, seine Anzeige zu aktualisieren.



Rahmenarchitektur (Framework)

Zweck

Biete ein (nahezu) vollständiges Programm, das durch Einfüllen geplanter „Lücken“ erweitert werden kann. Es enthält die vollständige Anwendungslogik, meistens sogar ein komplettes Hauptprogramm. Von einigen der Klassen in dem Programm können Benutzer Unterklassen bilden und dabei Methoden überschreiben oder vordefinierte abstrakte Methoden implementieren.

Das Rahmenprogramm sieht vor, dass die vom Benutzer gelieferten Erweiterungen richtig aufgerufen werden.



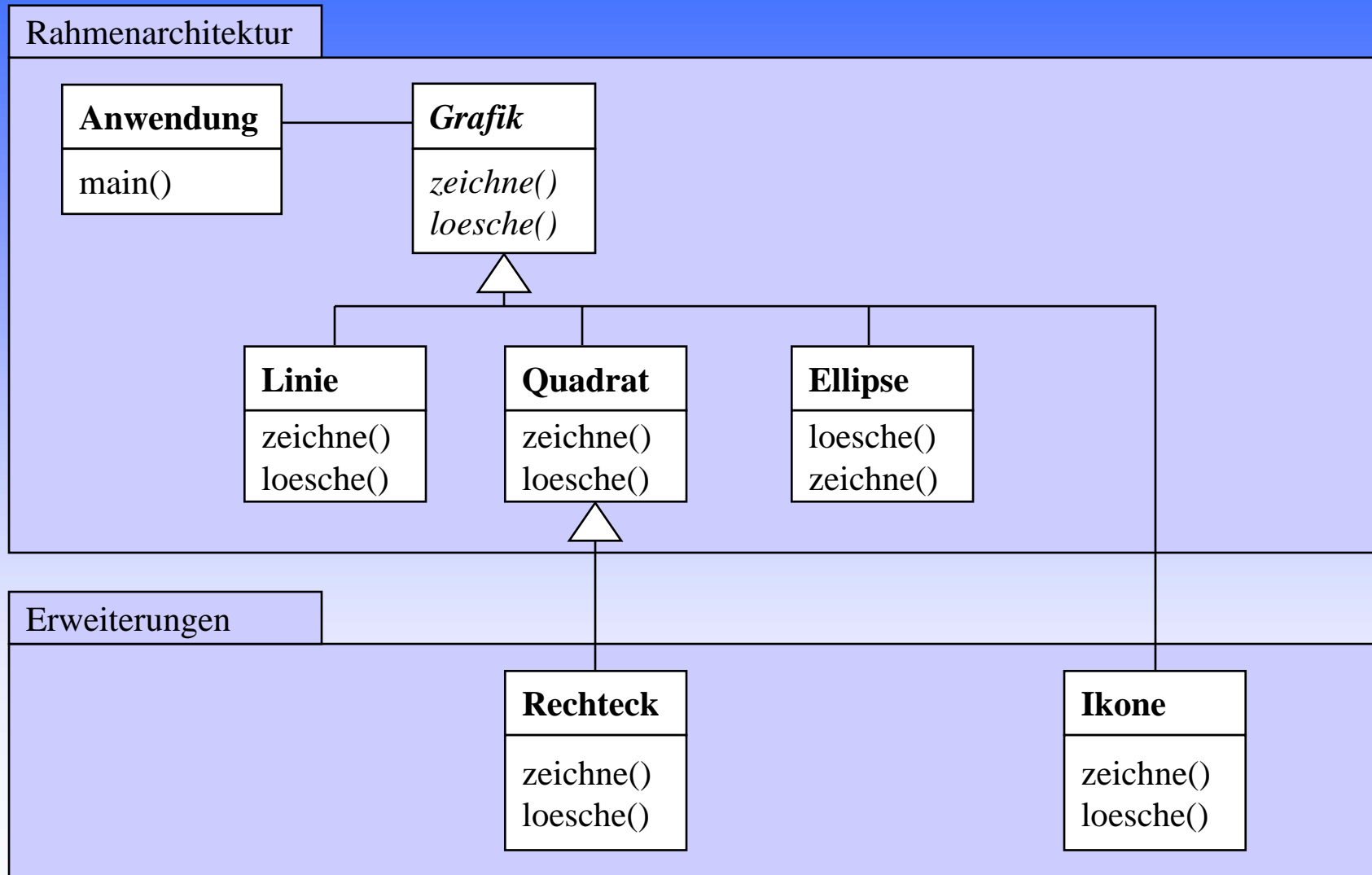
Beispiel für ein Rahmenprogramm

Ein Zeichen-Rahmenprogramm sieht eine Klasse *Grafik* mit einigen Unterklassen (z.B. *Linie*, *Quadrat*, *Ellipse*, usw.) vor. Der Benutzer darf neue Unterklassen von *Grafik* und seinen Unterklassen bilden, z.B. *Rechteck* oder *Ikone*, muss dazu aber eine Methode **zeichne** bereitstellen.

Das Rahmenprogramm sorgt dann dafür, dass Objekte der neuen Klassen richtig erzeugt, auf dem Zeichenbrett positioniert, verschoben, gesichert, usw. werden können.

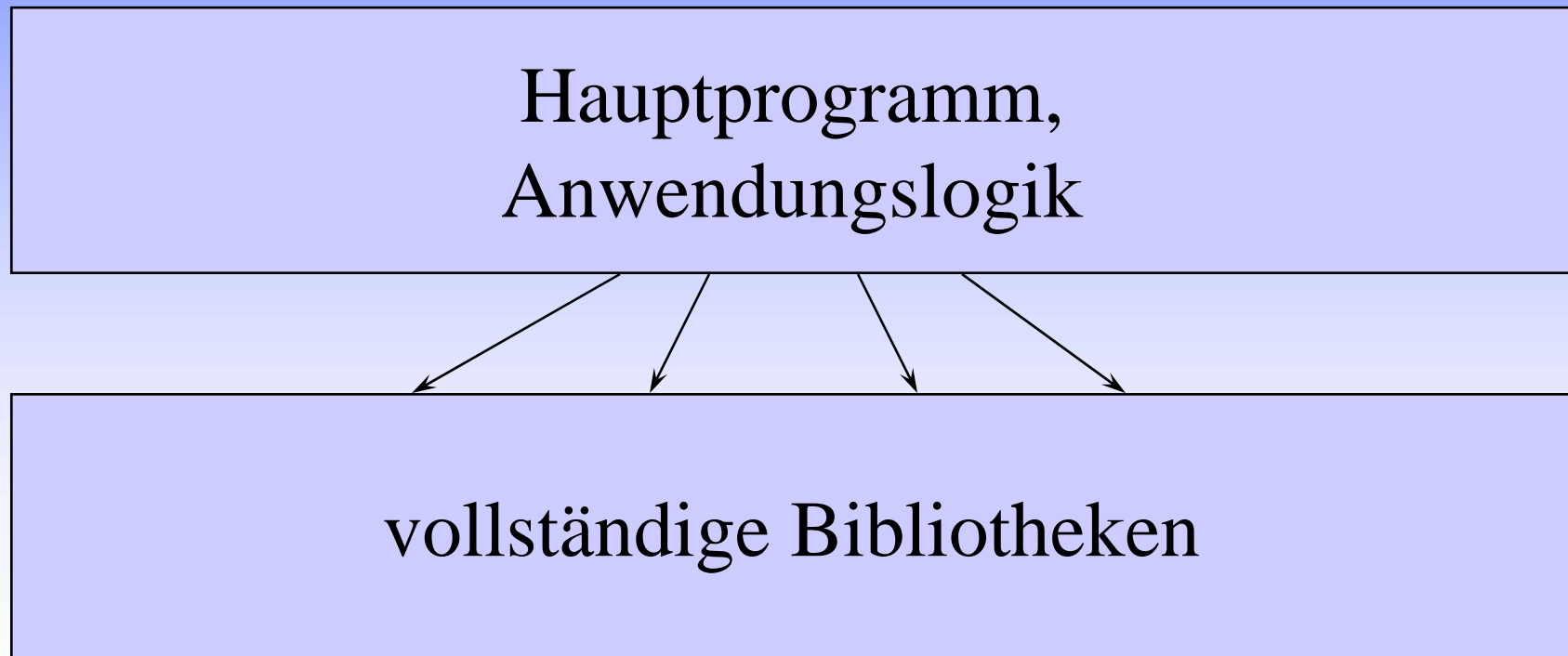


Zeichen-Rahmenarchitektur



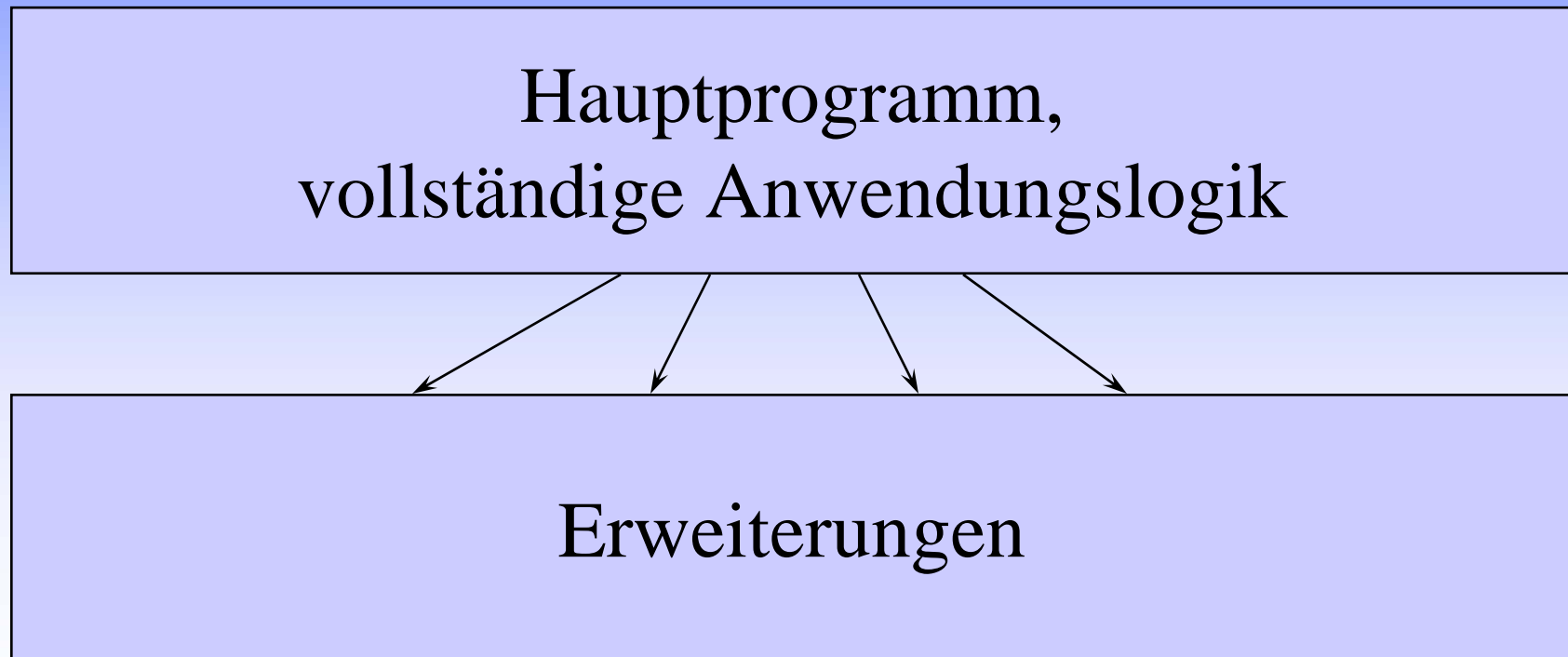
Herkömmliche Systemstruktur

- Hersteller liefert Bibliotheken,
- Benutzer schreibt Hauptprogramm und Anwendungslogik



Rahmenarchitektur

Ein Rahmenprogramm befolgt das „Hollywood-Prinzip“:
“Don’t call us - we’ll call you”. Das Hauptprogramm besteht
bereits und ruft die Erweiterungen der Benutzer auf.



Rahmenarchitektur

Anwendbarkeit

- Wenn eine Grundversion der Anwendung schon funktionsfähig sein soll.
- Wenn Erweiterungen möglich sein sollen, die sich konsistent verhalten (Anwendungslogik im Rahmenprogramm).
- Wenn komplexe Anwendungslogik nicht neu programmiert werden soll.

Die Muster Fabrikmethode, abstrakte Fabrik und Schablonenmethode werden häufig in Rahmenprogrammen benötigt.



2. Varianten-Muster

Gemeinsamkeiten von verwandten Einheiten werden aus ihnen herausgezogen und an einer einzigen Stelle beschrieben. Aufgrund ihrer Gemeinsamkeiten können unterschiedliche Komponenten im gleichen Programm einheitlich verwendet werden, und Code-Wiederholungen werden vermieden.

- Oberklasse
- Besucher
- Schablonenmethode
- Fabrikmethode
- Erbauer
- Abstrakte Fabrik



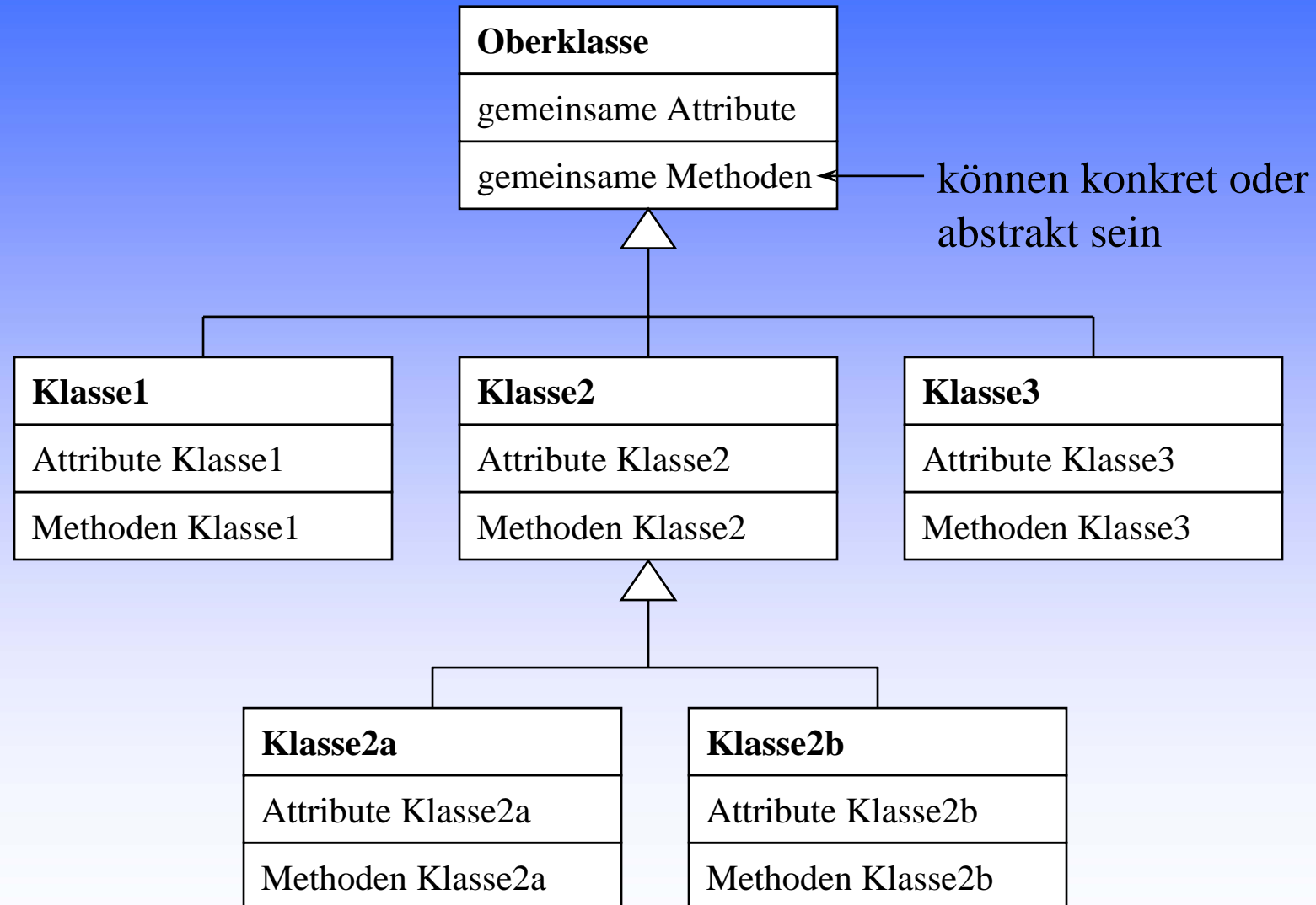
Oberklasse

Zweck

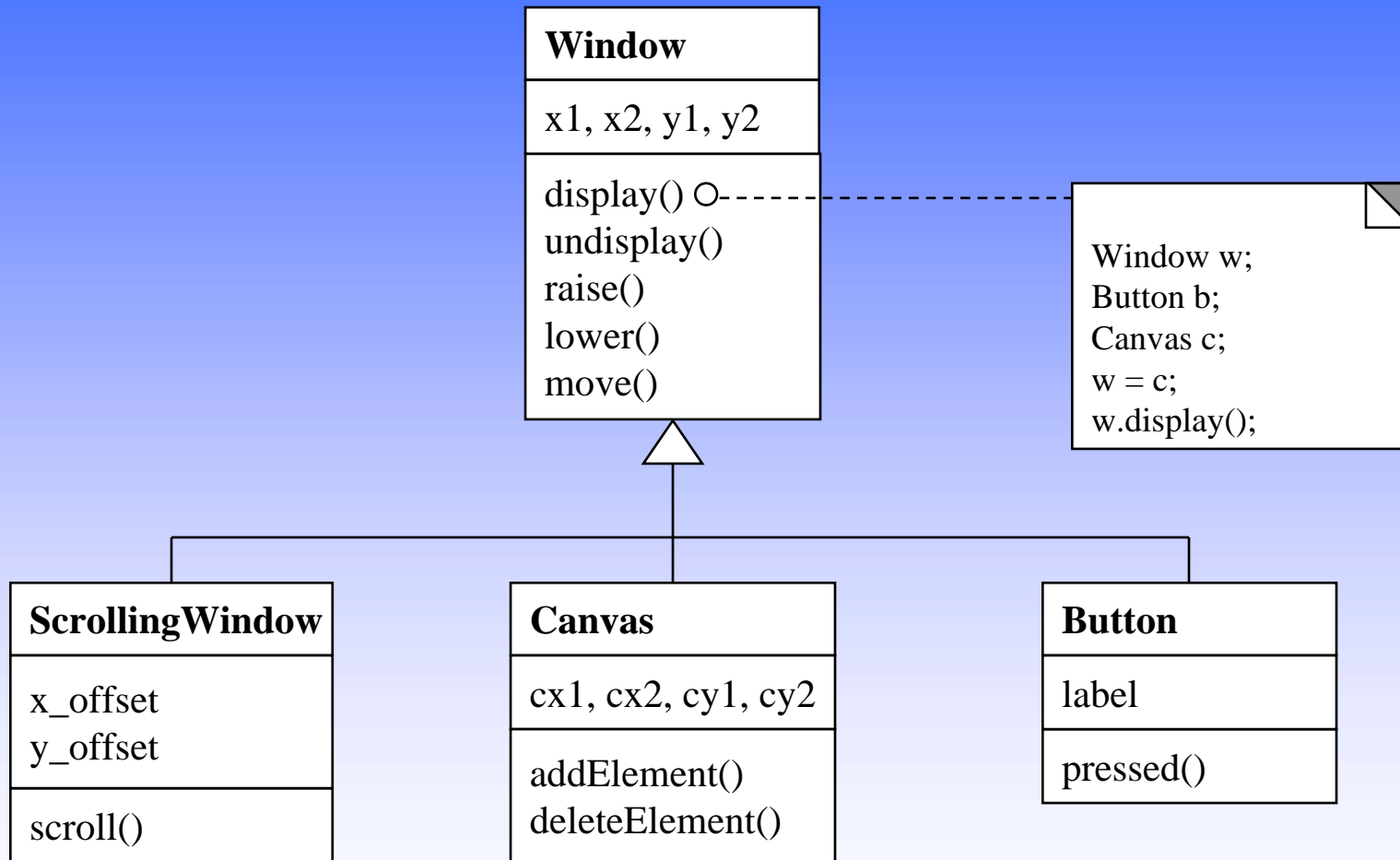
Einheitliche Behandlung von Objekten, die unterschiedlichen Klassen angehören, aber gemeinsame Attribute oder Methoden besitzen.



Struktur der Oberklasse



Beispiel einer Oberklasse



Oberklasse

Anwendbarkeit

- Wenn Objekte verschiedener Klassen gemeinsame Attribute, Methoden oder Schnittstellen haben.
- Wenn Objekte verschiedener Klassen einheitlich in einem Programm behandelt werden sollen.
- Wenn es möglich sein soll, weitere Klassen hinzuzufügen, ohne den bestehenden Quelltext zu verändern.



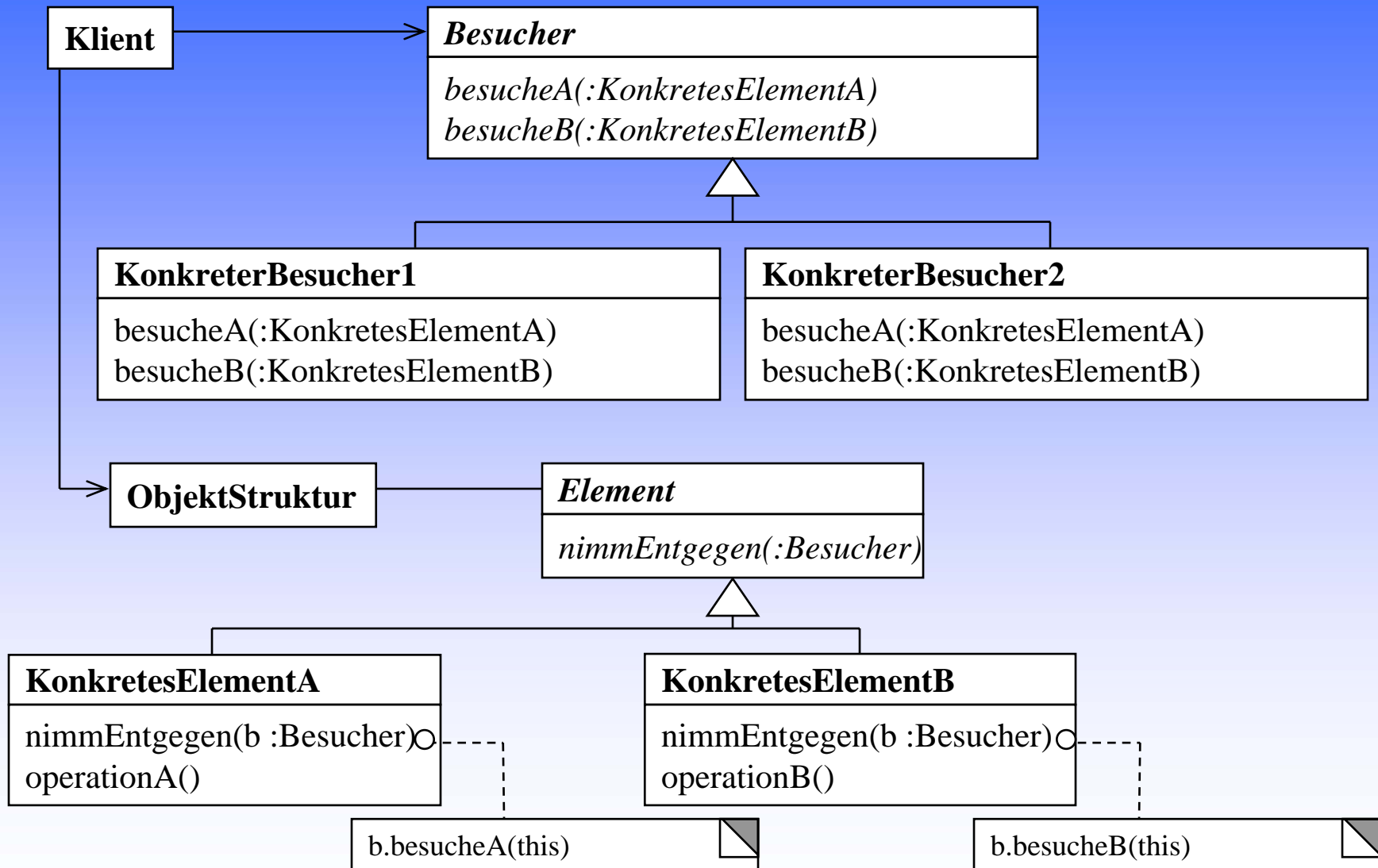
Besucher (Visitor)

Zweck

Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

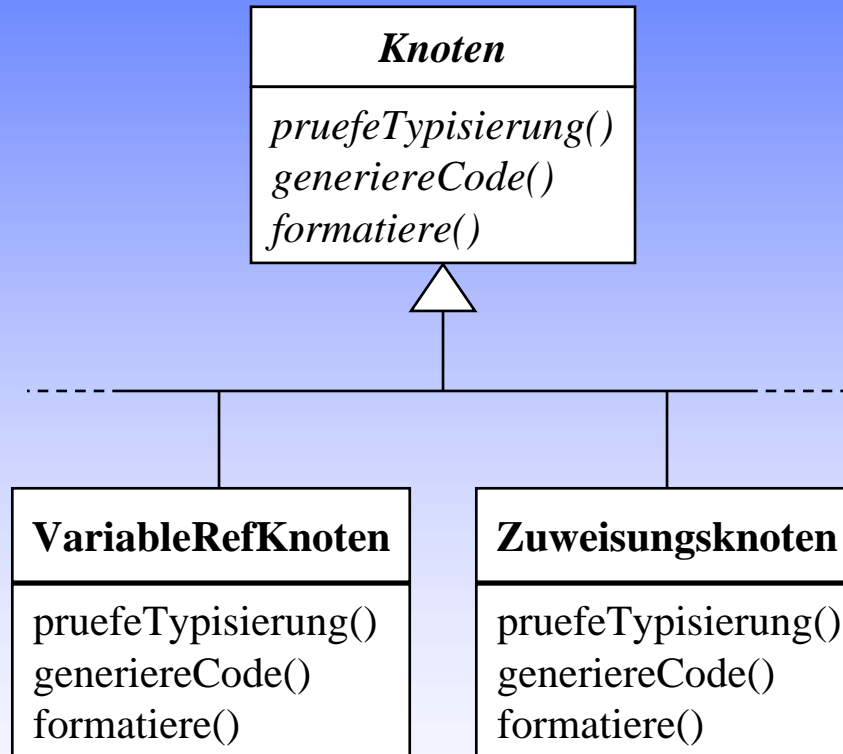


Struktur des Besucher



Beispiel: ohne Besucher

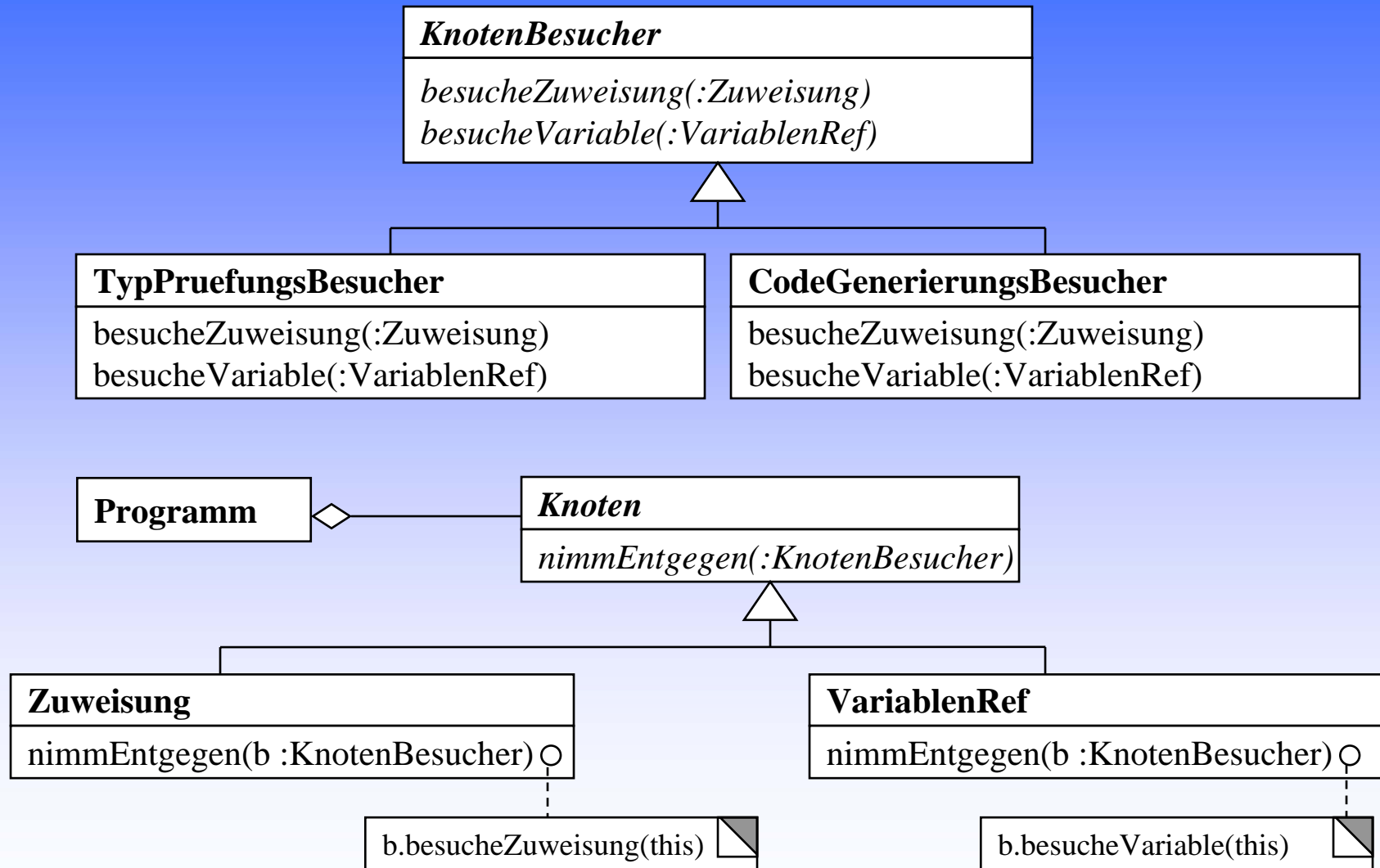
Abstrakte Syntaxbäume in einem Übersetzer



Die einzelnen Operationen sind über viele Klassen verstreut. Bei Einführung neuer Operationen müssen alle diese Klassen erweitert werden.



Beispiel: mit Besucher



Besucher

Anwendbarkeit

- Wenn eine Objektstruktur viele Klassen von Objekten mit unterschiedlichen Schnittstellen enthält und Operationen auf diesen Objekten ausgeführt werden sollen, die von ihren konkreten Klassen abhängen.
- Wenn viele unterschiedliche und nicht miteinander verwandte Operationen auf den Objekten einer Objektstruktur ausgeführt werden müssen und diese Klassen nicht mit diesen Operationen „verschmutzt“ werden sollen.
- Wenn sich die Klassen, die eine Objektstruktur definieren, praktisch nie ändern, aber häufig neue Operationen für die Struktur definiert werden.



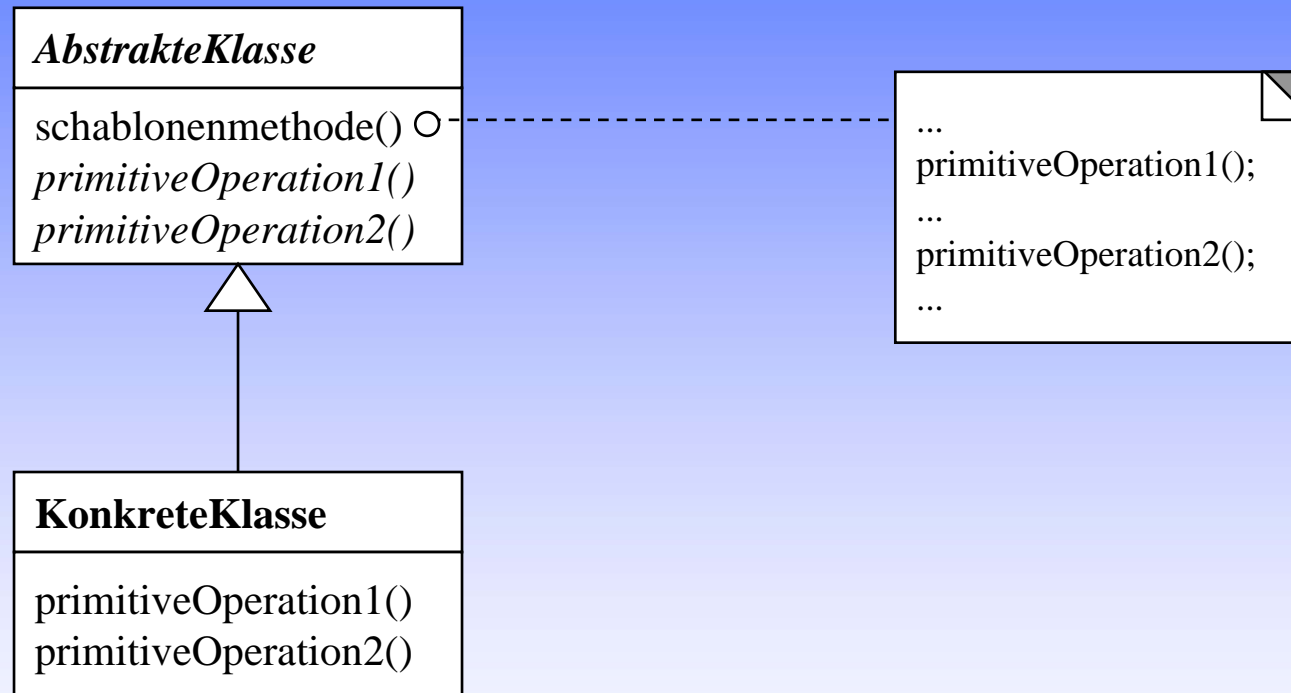
Schablonenmethode (Template Method)

Zweck

Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.



Struktur für Schablonenmethode



Schablonenmethode

Anwendbarkeit

- Um die invarianten Teile eines Algorithmus genau einmal festzulegen und es dann Unterklassen zu überlassen, das variierende Verhalten zu implementieren.
- Wenn gemeinsames Verhalten aus Unterklassen herausfaktoriert und in einer allgemeinen Klasse plaziert werden soll, um die Verdopplung von Code zu vermeiden.
- Um die Erweiterungen durch Unterklassen zu kontrollieren. Eine Schablonenmethode lässt sich so definieren, dass sie „Einschubmethoden“ (hooks) an bestimmten Stellen aufruft und damit Erweiterungen nur an diesen Stellen zulässt.



Fabrikmethode (Factory Method)

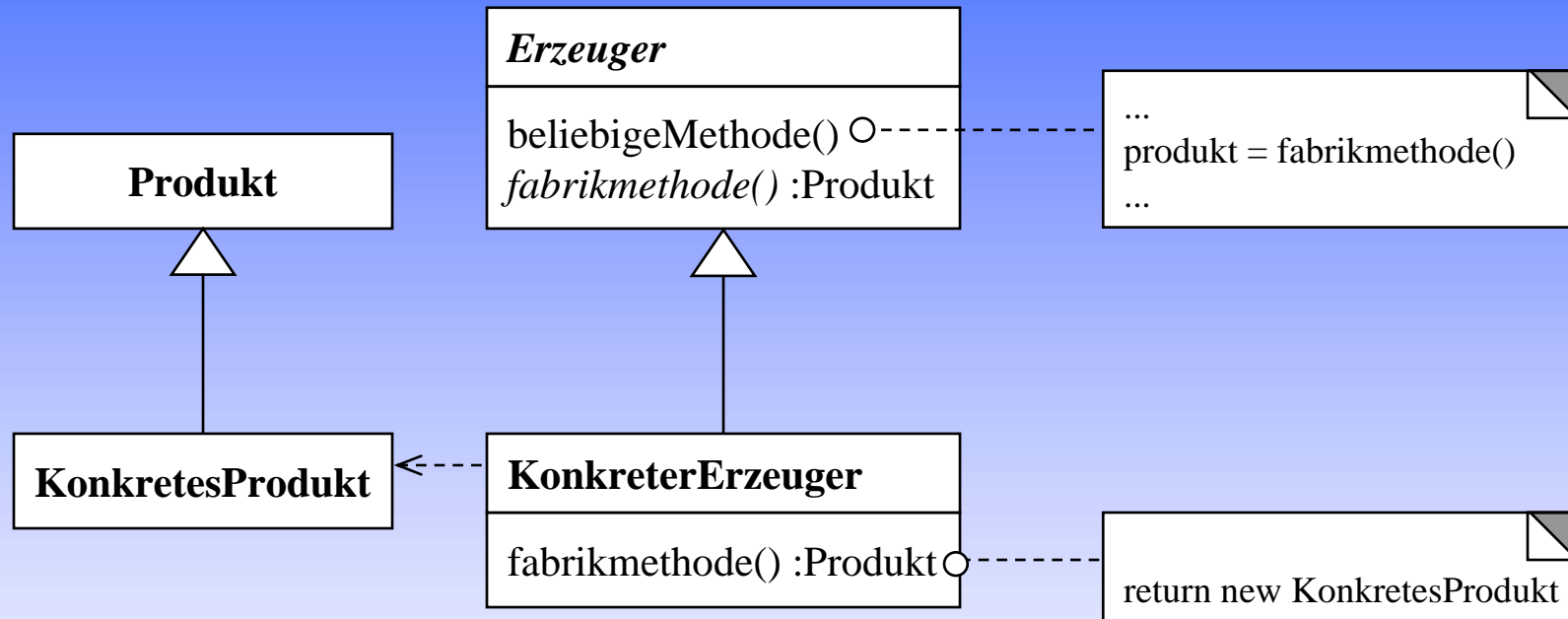
Zweck

Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist.

Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.



Struktur der Fabrikmethode



Fabrikmethode

Anwendbarkeit

- Wenn eine Klasse die Klasse von Objekten, die sie erzeugen muss, nicht im voraus kennen kann.
- Wenn eine Klasse möchte, dass ihre Unterklasse die von ihr zu erzeugenden Objekte festlegen.
- Wenn Klassen Zuständigkeiten an eine von mehreren Hilfsunterklassen delegieren sollen und das Wissen, an welche Hilfsunterklasse die Zuständigkeit delegiert wird, lokalisiert werden soll.

Eine Fabrikmethode ist die Einschubmethode bei einer Schablonenmethode für Objekterzeugung.



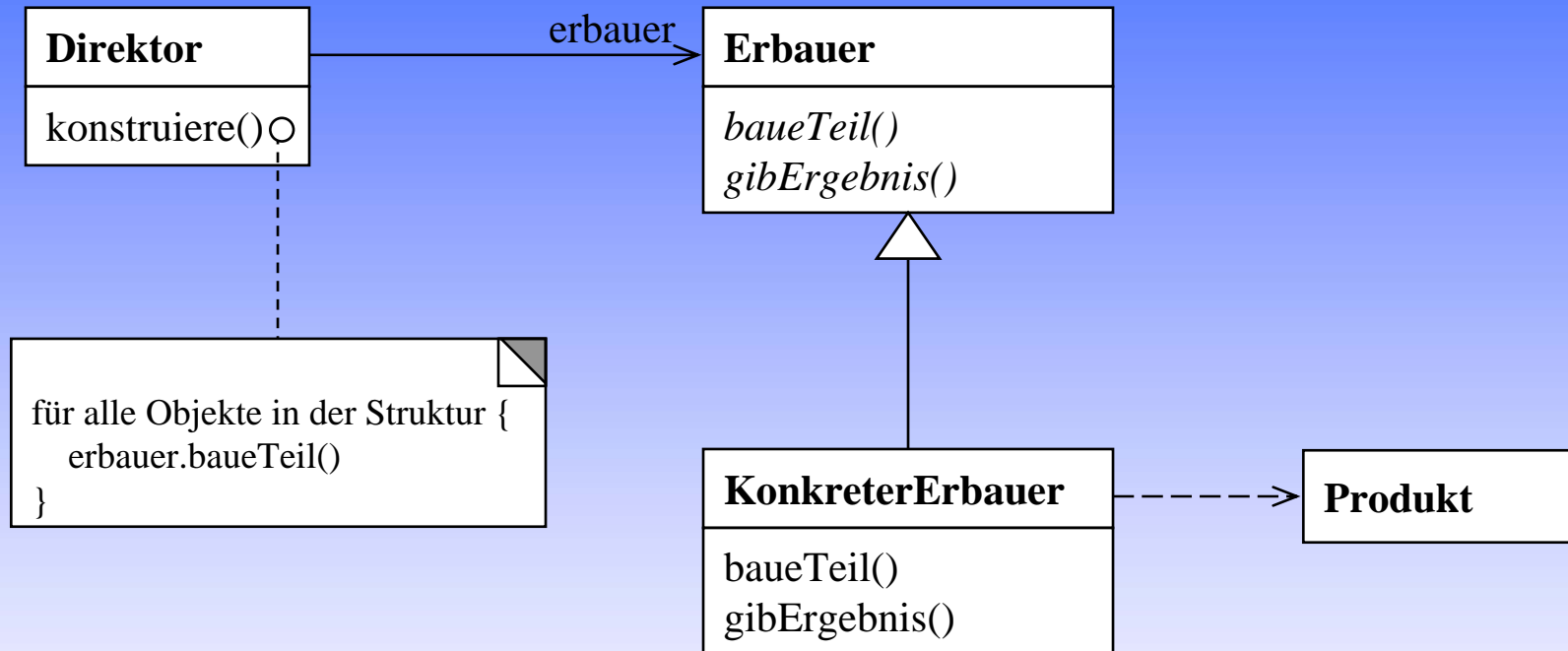
Erbauer (Builder)

Zweck

Trenne die Konstruktion eines komplexen Objekts (bestehend aus mehreren Teilen) von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.

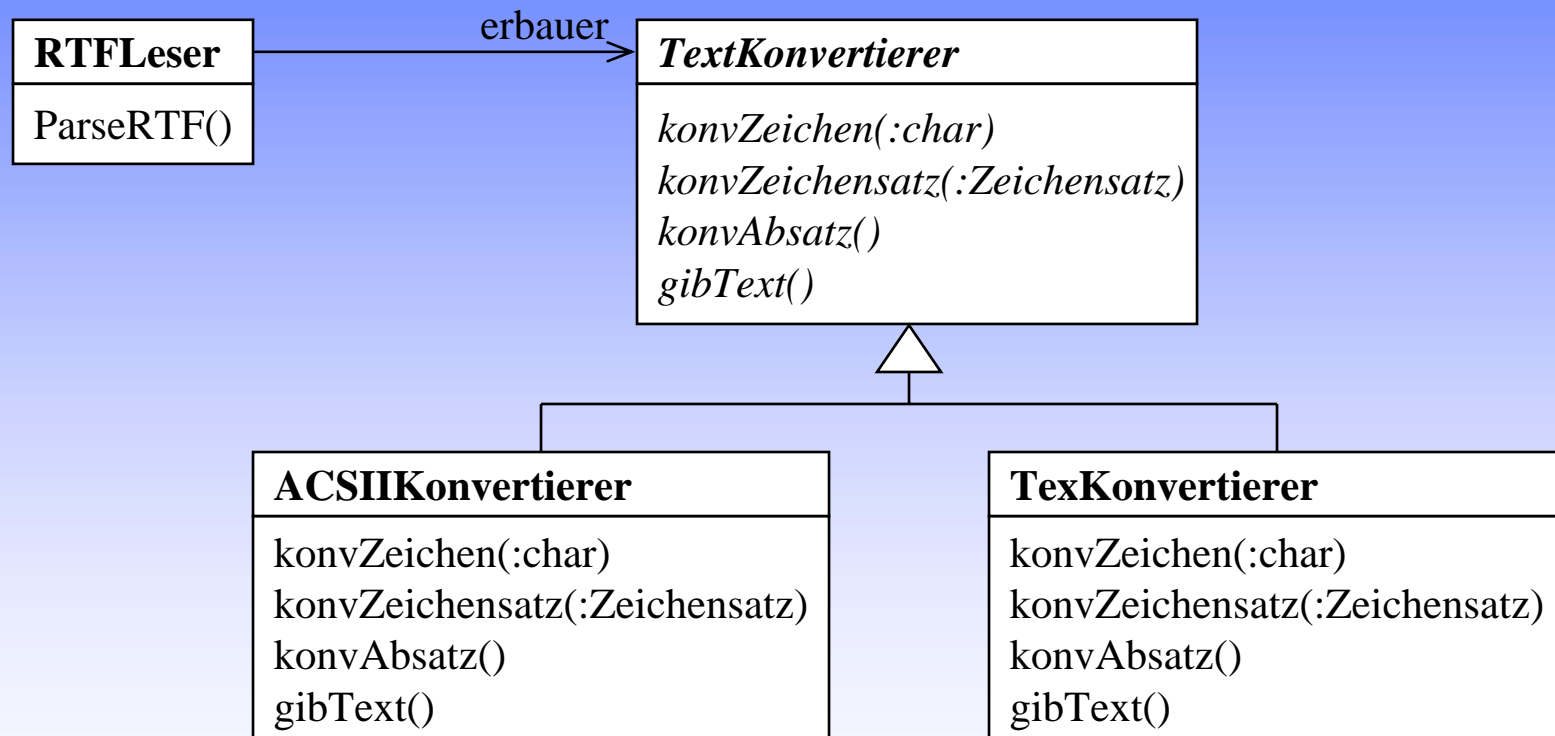


Struktur des Erbauers



Beispiel eines Erbauers

Transformation zwischen Textformaten



Erbauer

Anwendbarkeit

- Der Algorithmus zum Erzeugen eines komplexen Objekts soll unabhängig von den Teilen sein, aus denen das Objekt besteht und wie sie zusammengesetzt werden.
- Der Konstruktionsprozess muss verschiedene Repräsentationen des zu konstruierenden Objekts erlauben.



Erbauer

Interaktionen

- Der Klient erzeugt das Direktorobjekt und konfiguriert es mit dem gewünschten Erbauerobjekt.
- Der Direktor informiert den Erbauer, wenn ein Teil des Produkts gebaut werden soll.
- Der Erbauer bearbeitet die Anfragen des Direktors und fügt Teile zum Produkt hinzu.
- Der Klient erhält das Produkt vom Erbauer.



Erbauer vs. Fabrikmethode

Der Erbauer trennt den Konstruktionsalgorithmus und die Schnittstelle zum Bauen der einzelnen Teile (Direktor und Erbauerklassen). Daher ist es möglich, die Erbauerklasse und damit die Repräsentation der Einzelteile und des gesamten Produkts zu variieren (auch dynamisch).



Abstrakte Fabrik (Abstract Factory)

Zweck

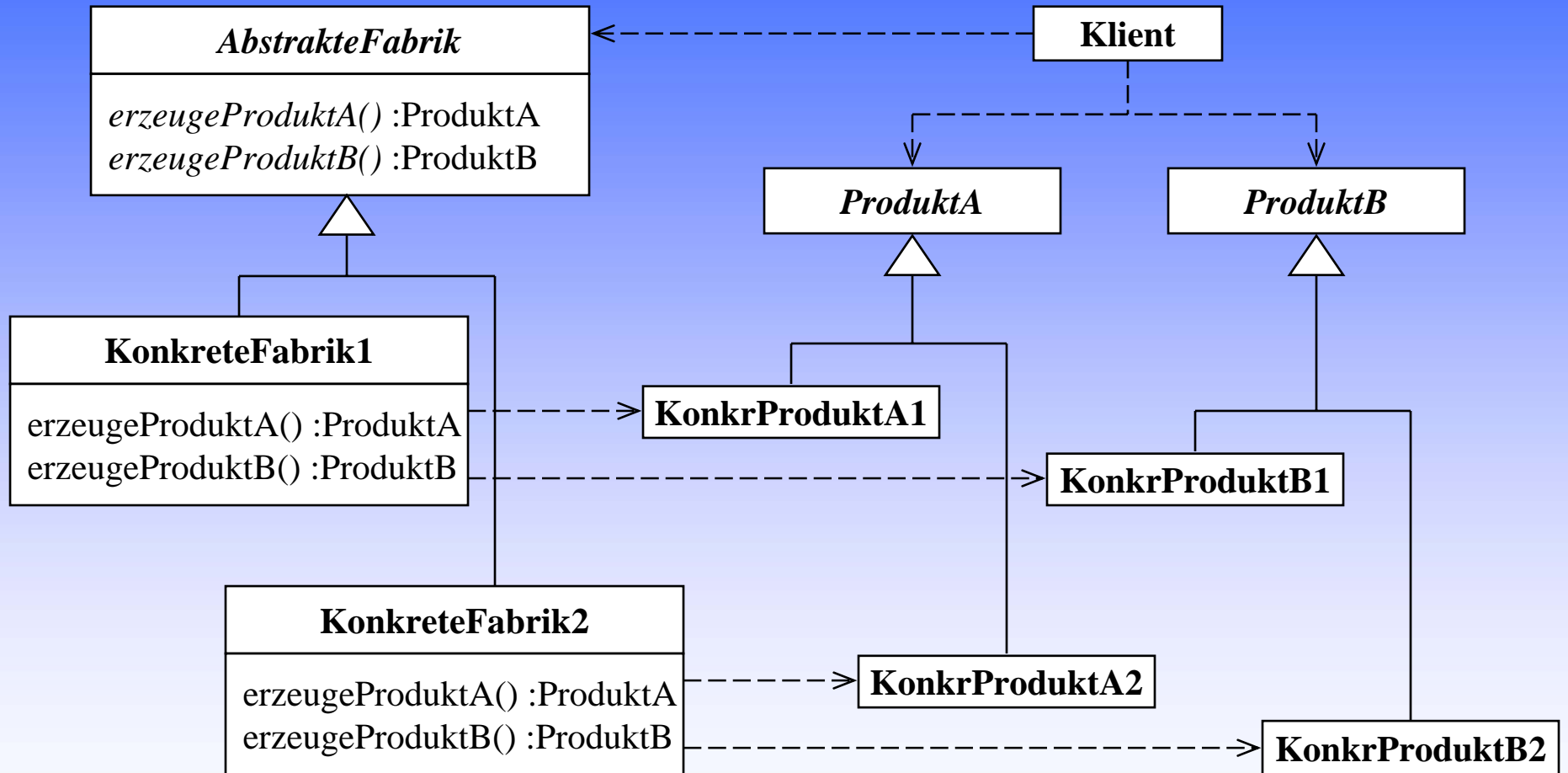
Bietet eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.

Auch bekannt als

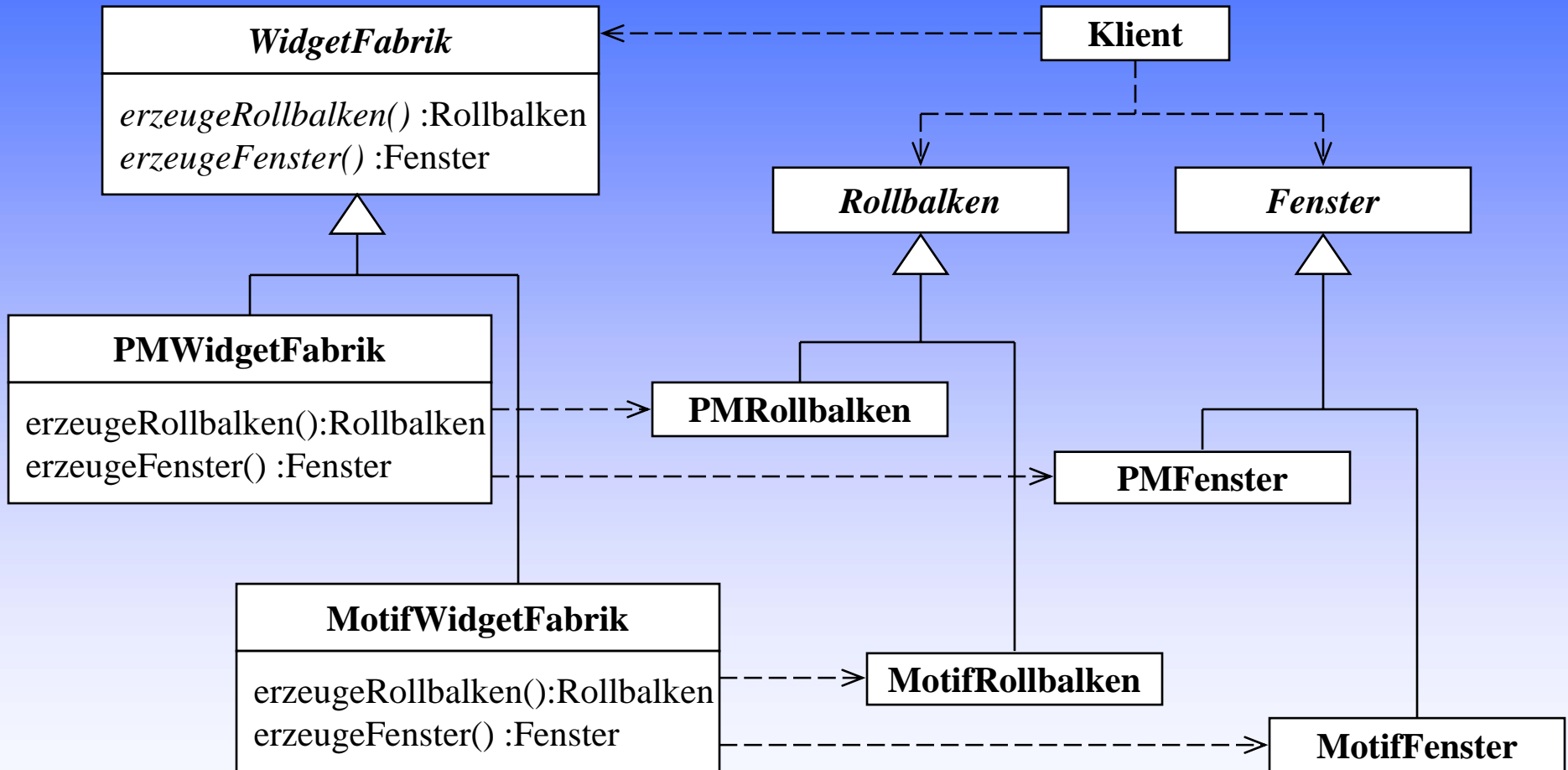
Kit



Struktur der Abstrakten Fabrik



Beispiel einer Abstrakten Fabrik



Abstrakte Fabrik

Anwendbarkeit

- Wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden.
- Wenn ein System mit einer von mehreren Produktfamilien konfiguriert werden soll.
- Wenn eine Familie von verwandten Produktobjekten zusammen verwendet werden sollen, und dies erzwungen werden muss.
- Bei einer Klassenbibliothek, die nur die Schnittstellen, nicht aber die Implementierungen offen legt.



Erbauer vs. Abstrakte Fabrik

Das Abstrakte-Fabrik-Muster ist dem Erbauermuster in der Hinsicht ähnlich, dass es ebenfalls komplexe Objekte konstruieren kann. Der Hauptunterschied ist, dass das Erbauermuster sich auf den schrittweisen Konstruktionsprozess eines komplexen Objekts konzentriert. Die Betonung der abstrakten Fabrik liegt auf Familien von Produktobjekten (ob nun einfach oder komplex). Der Erbauer gibt das Produkt als letzten Schritt zurück, während die Abstrakte Fabrik das Produkt unmittelbar zurückgibt.



3. Zustandshandhabungs-Muster

- Memento
- Prototyp
- Fliegengewicht
- Einzelstück (siehe Einführungsbeispiel)



Memento (Memento)

Zweck

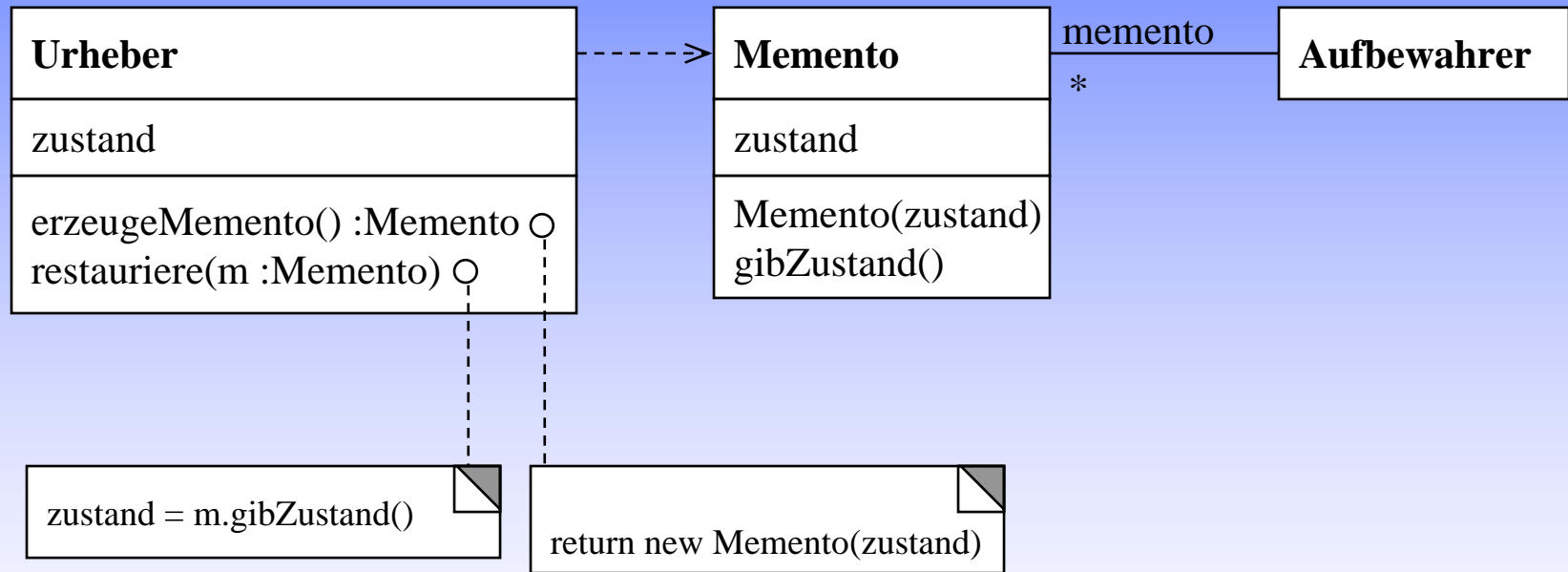
Erfasse und externalisiere den internen Zustand eines Objekts, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.

Auch bekannt als

Token

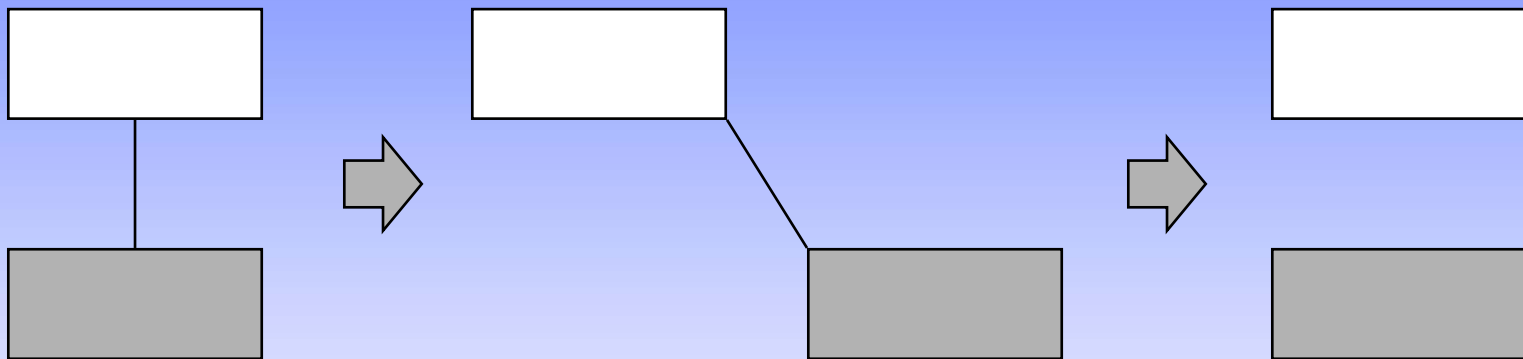


Struktur des Mementos



Beispiel eines Mementos

Komplexe Haltepunkte und Undo-Mechanismen
Z.B. in einem grafischen Editor:



Rechtecke bleiben
verbunden, wenn ein
Rechteck bewegt wird.

Mögliches falsches
Ergebnis nach einem
Undo, falls nur Entfernung
zum Ursprung des Rechtecks
gespeichert wurde.



Memento

Anwendbarkeit

- Wenn eine Momentaufnahme (eines Teils) des Zustands eines Objekts zwischengespeichert werden muss, so dass es zu einem späteren Zeitpunkt in diesen Zustand zurückversetzt werden kann, *und*
- wenn eine direkte Schnittstelle zum Ermitteln des Zustands die Implementierungsdetails offenlegen und die Kapselung des Objekts aufbrechen würde.



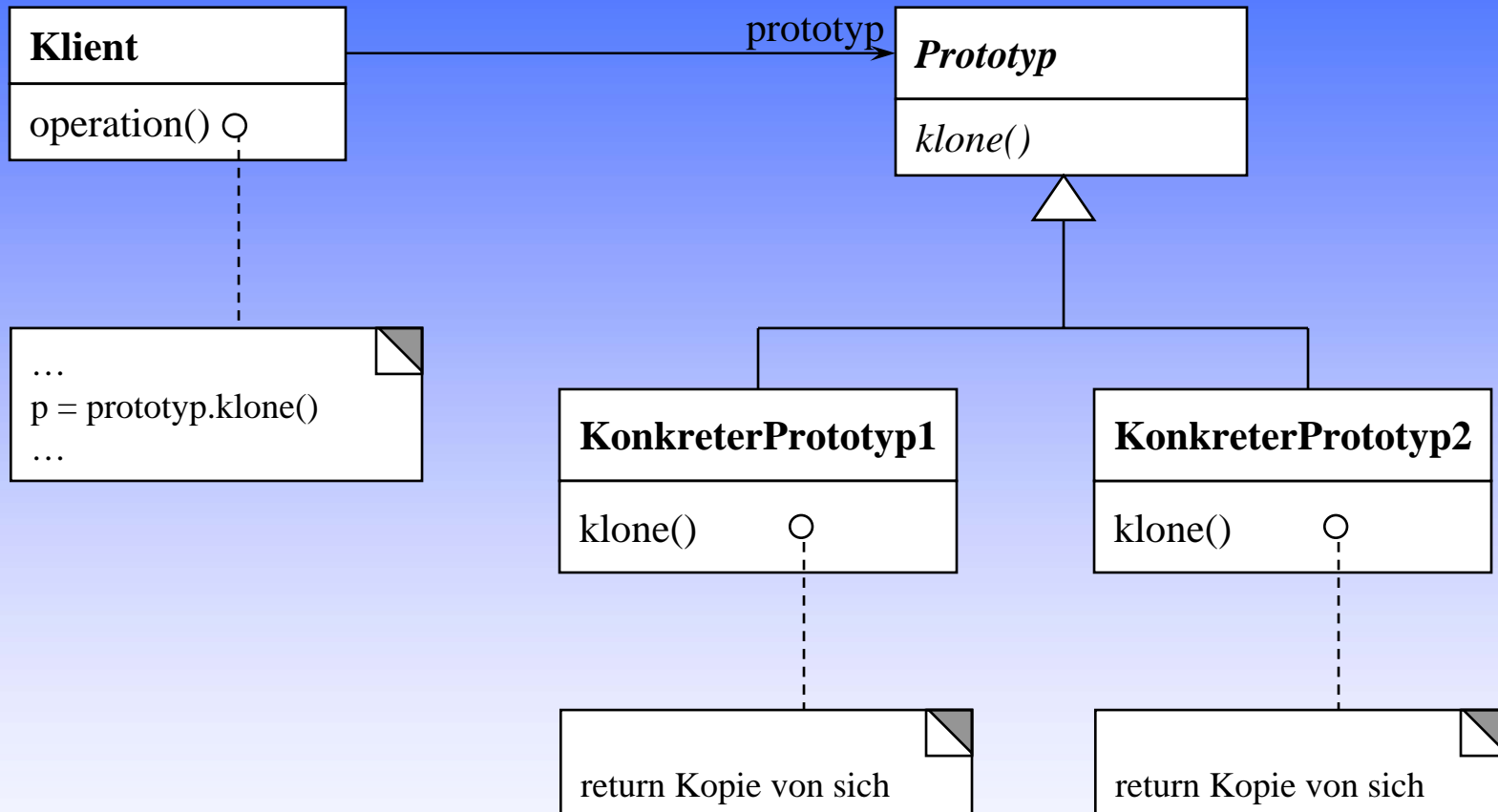
Prototyp (Prototype)

Zweck

Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines typischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototyps.



Struktur des Prototyps



Prototyp

Anwendbarkeit

Das Prototypmuster wird verwendet, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden, *und*

- wenn die Klassen zu erzeugender Objekte erst zur Laufzeit spezifiziert werden, z.B. durch dynamisches Laden, *oder*
- um eine Klassenhierarchie von Fabriken zu vermeiden, die parallel zur Klassenhierarchie der Produkte verläuft, *oder*
- wenn Exemplare einer Klasse nur wenige Zustandskombinationen haben können. Es ist möglicherweise bequemer, eine entsprechende Anzahl von Prototypen einzurichten und sie zu klonen statt die Objekte einer Klasse jedesmal von Hand mit dem richtigen Zustand zu erzeugen.



Prototyp vs. Fabrikmethode

- Benutze Prototyp, falls der Aufbau eines Objekts wesentlich mehr Zeit erfordert als eine Kopie anzulegen.
- Benutze Prototypen nicht, wenn die Kopien so groß werden, dass es zu Speicherengpässen kommt.



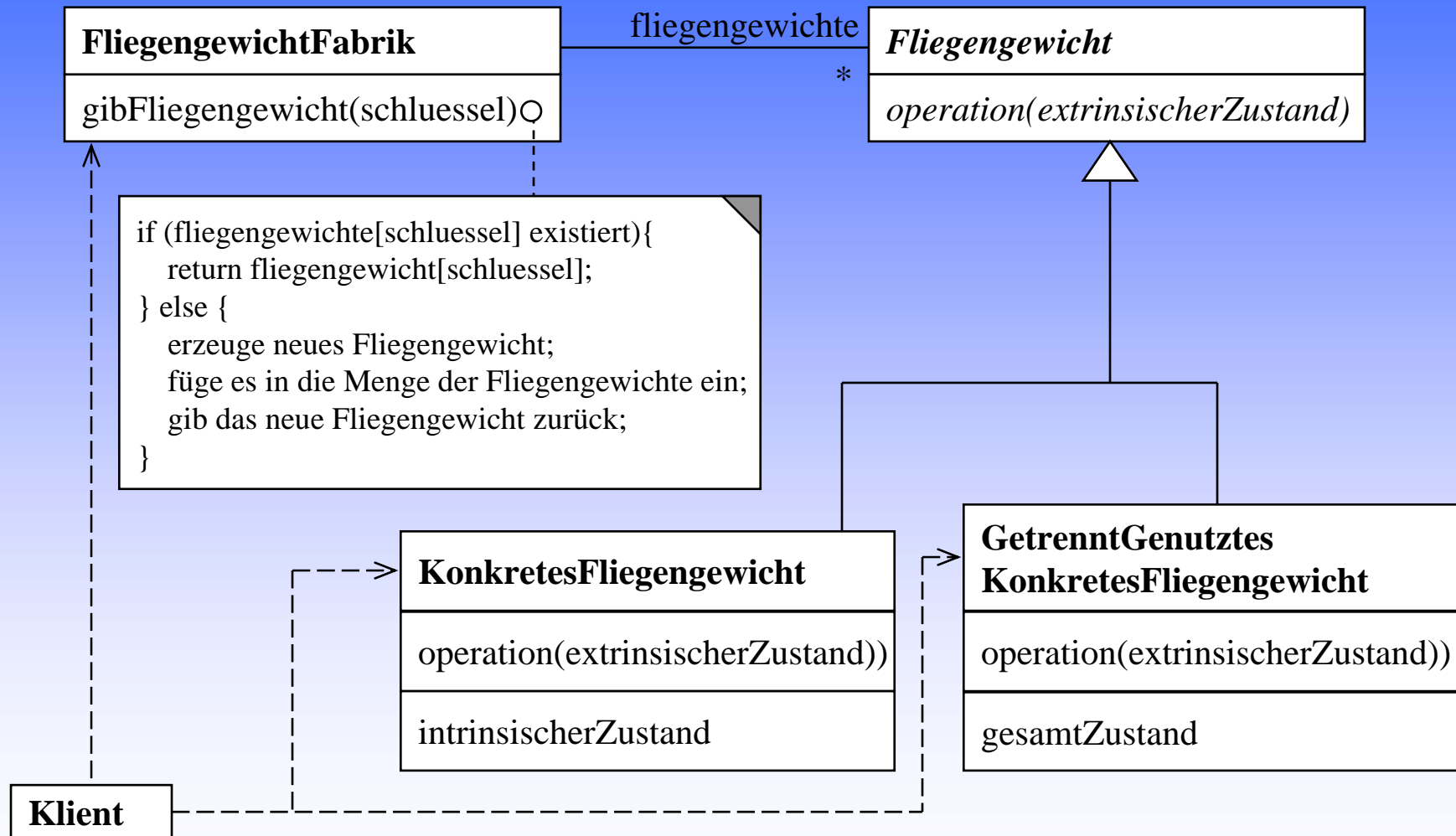
Fliegengewicht (Flyweight)

Zweck

Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient speichern zu können.

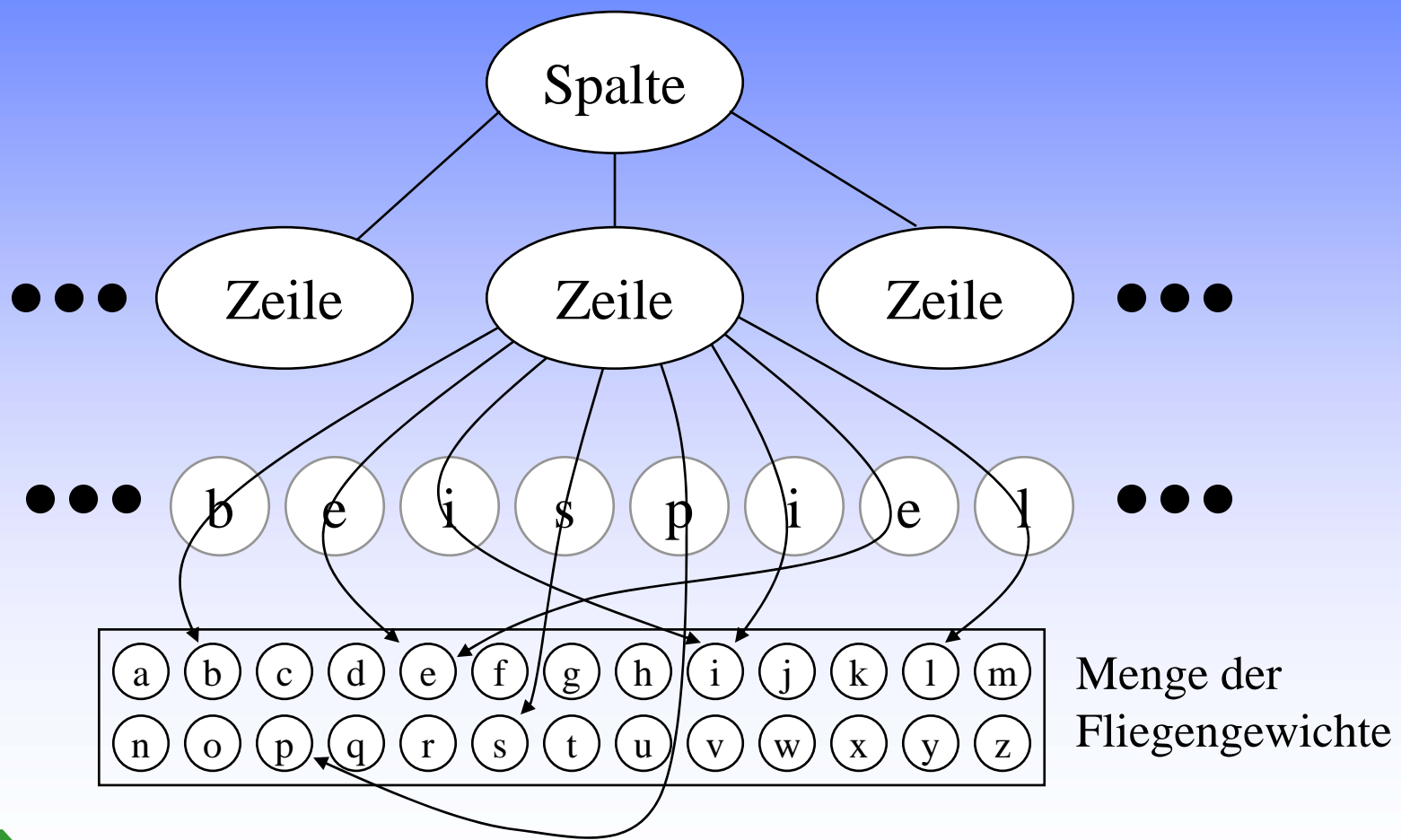


Struktur des Fliegengewichts



Beispiel eines Fliegengewichts

Objektmodellierung bis hinunter zu einzelnen Zeichen in einem Texteditor:



Beispiel eines Fliegengewichts

- Die einzelnen Zeichen können durch einen Code repräsentiert werden (innerer Zustand).
- Die Informationen über Schriftart, Größe und Position können externalisiert werden (äußerer Zustand) und in dem Zeilen- oder Spaltenobjekt oder auch in Teilfolgen von Zeichen gespeichert werden.



Fliegengewicht

Anwendbarkeit

- Wenn die Anwendung eine große Menge von Objekten verwendet, *und*
- wenn Speicherkosten allein aufgrund der Anzahl von Objekten hoch sind, *und*
- wenn ein Großteil des Objektzustands in den Kontext verlagert und damit extrinsisch gemacht werden kann, *und*
- viele Gruppen von Objekten durch relativ wenige gemeinsam genutzte Objekte ersetzt werden, sobald einmal der extrinsische Zustand entfernt wurde, *und*
- die Anwendung nicht von der Identität der Objekte abhängt (sonst Identität trotz konzeptuellem Unterschied).



4. Steuerungs-Muster

- Tafel
- Befehl
- Zuständigkeitskette
- Master/Slave
- Prozesssteuerung
 - System ohne Rückkoppelung
 - System mit Rückkoppelung
 - Regelung mit Rückführung
 - Regelung mit Störgrößenaufschaltung



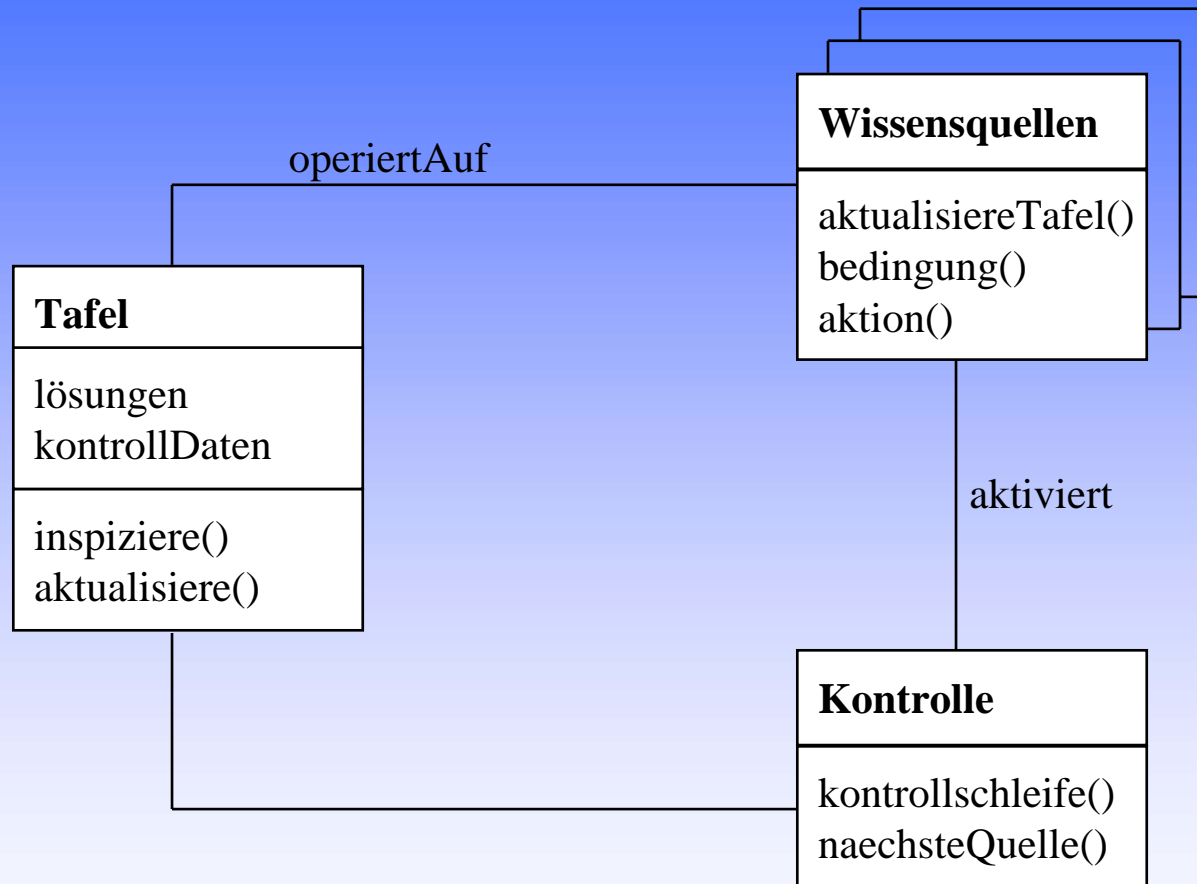
Tafel (Blackboard)

Zweck

Das Tafelmuster ist nützlich bei Problemen, für die keine deterministischen Lösungsstrategien bekannt sind. Mehrere spezialisierte Subsysteme vereinigen ihr Wissen auf der Tafel um eine eventuell partielle oder approximative Lösung zu finden.



Struktur der Tafel



Tafel

- Einfache Form: Kontrolle aktiviert alle Wissensquellen der Reihe nach.
- Komplexere Form: bestimmt eine Folge anwendbarer Wissensquellen (über *bedingung()*), wählt davon aus, und führt diese dann aus.



Tafel

Anwendbarkeit

- Wenn mehrere Transformationen („Wissensquellen“) auf einer gemeinsamen Datenstruktur („Tafel“) operieren.
- Wenn das Auslösen der Transformationen vom Inhalt der Datenstruktur gesteuert wird.
- Wenn die Auswahl der anwendbaren Transformationen gesteuert werden soll.



Befehl (Command)

Zweck

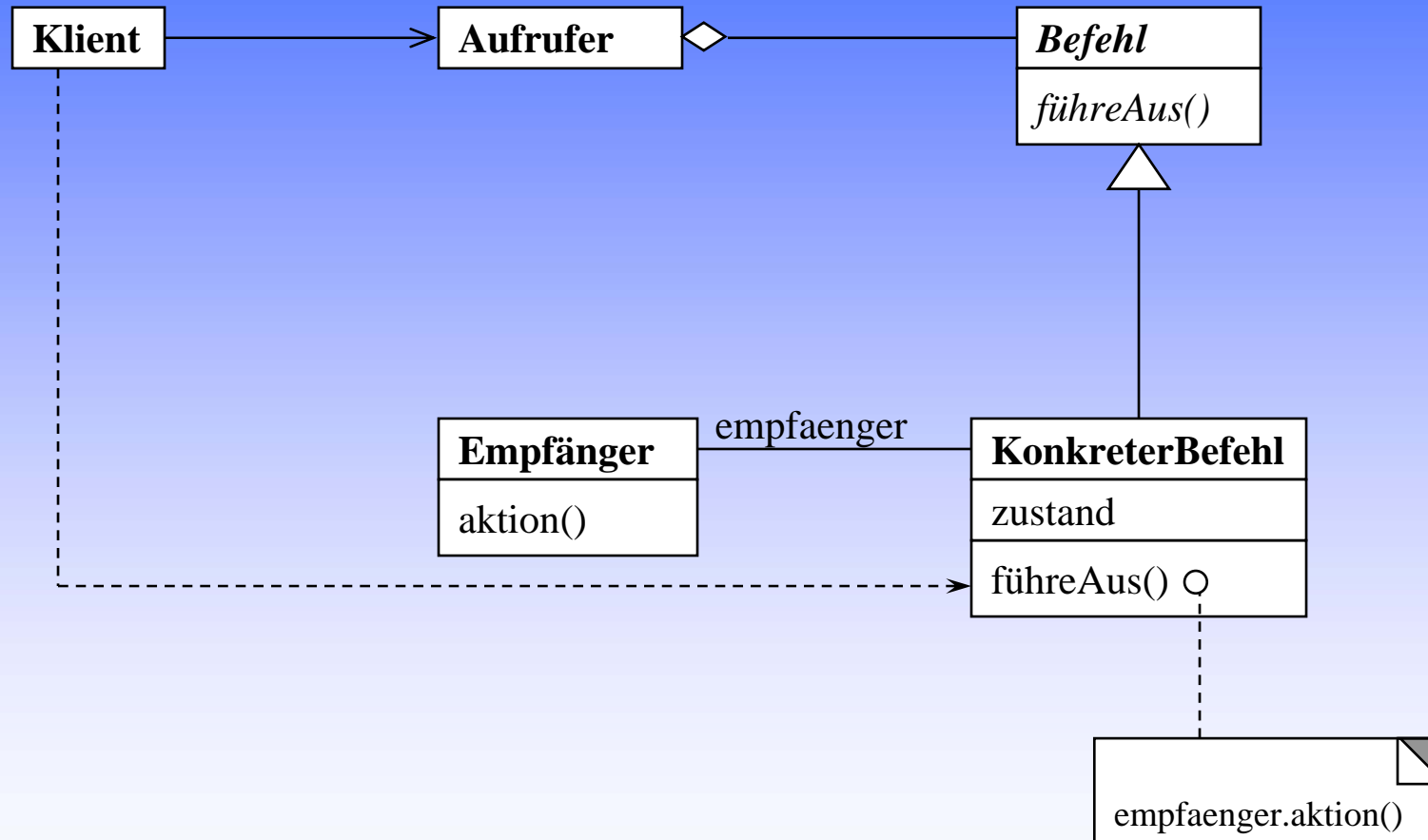
Kapsle einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Warteschlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.

Auch bekannt als

Kommando, Aktion, Transaktion

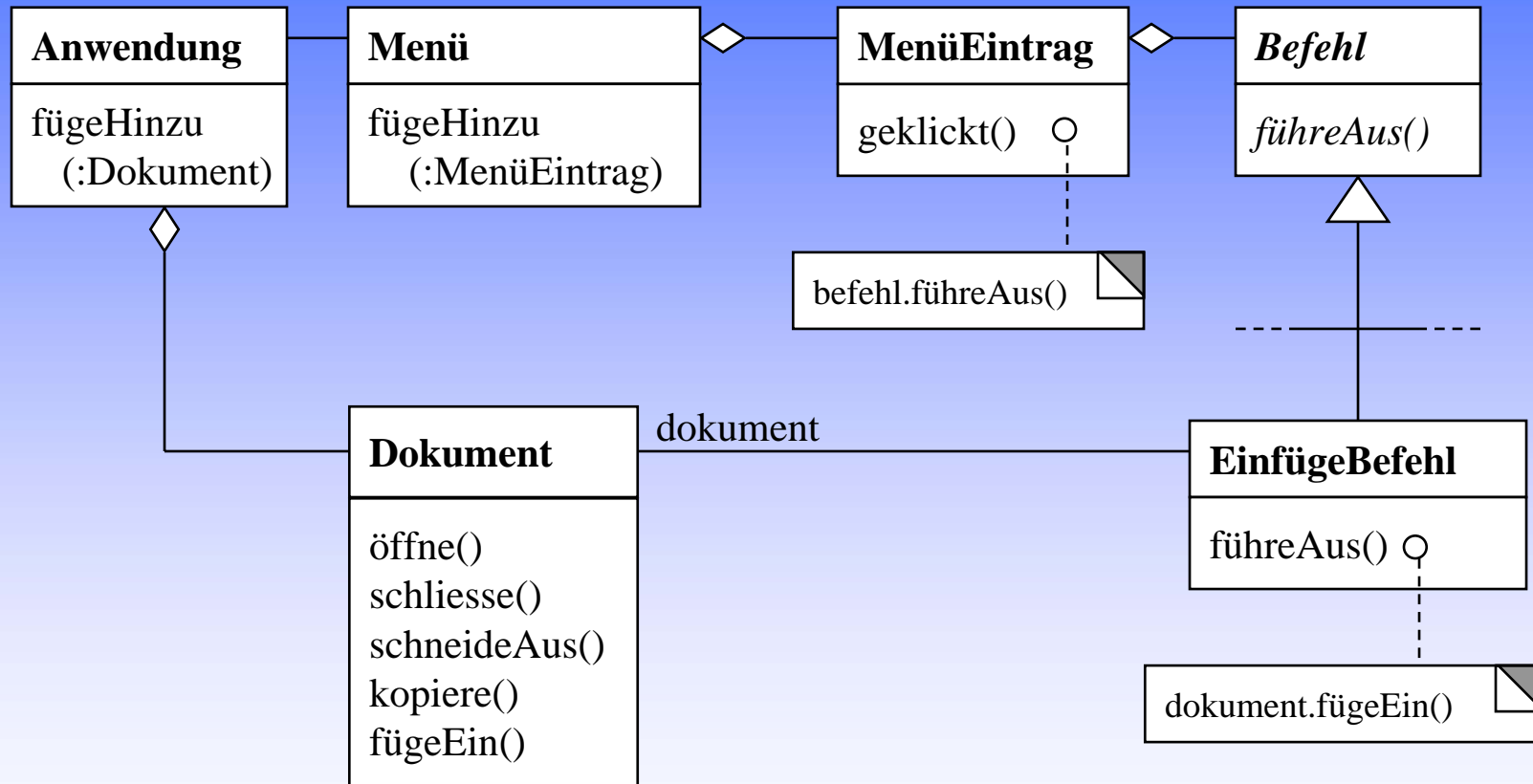


Struktur des Befehls



Beispiel eines Befehls

Menüs bei Benutzerschnittstellen



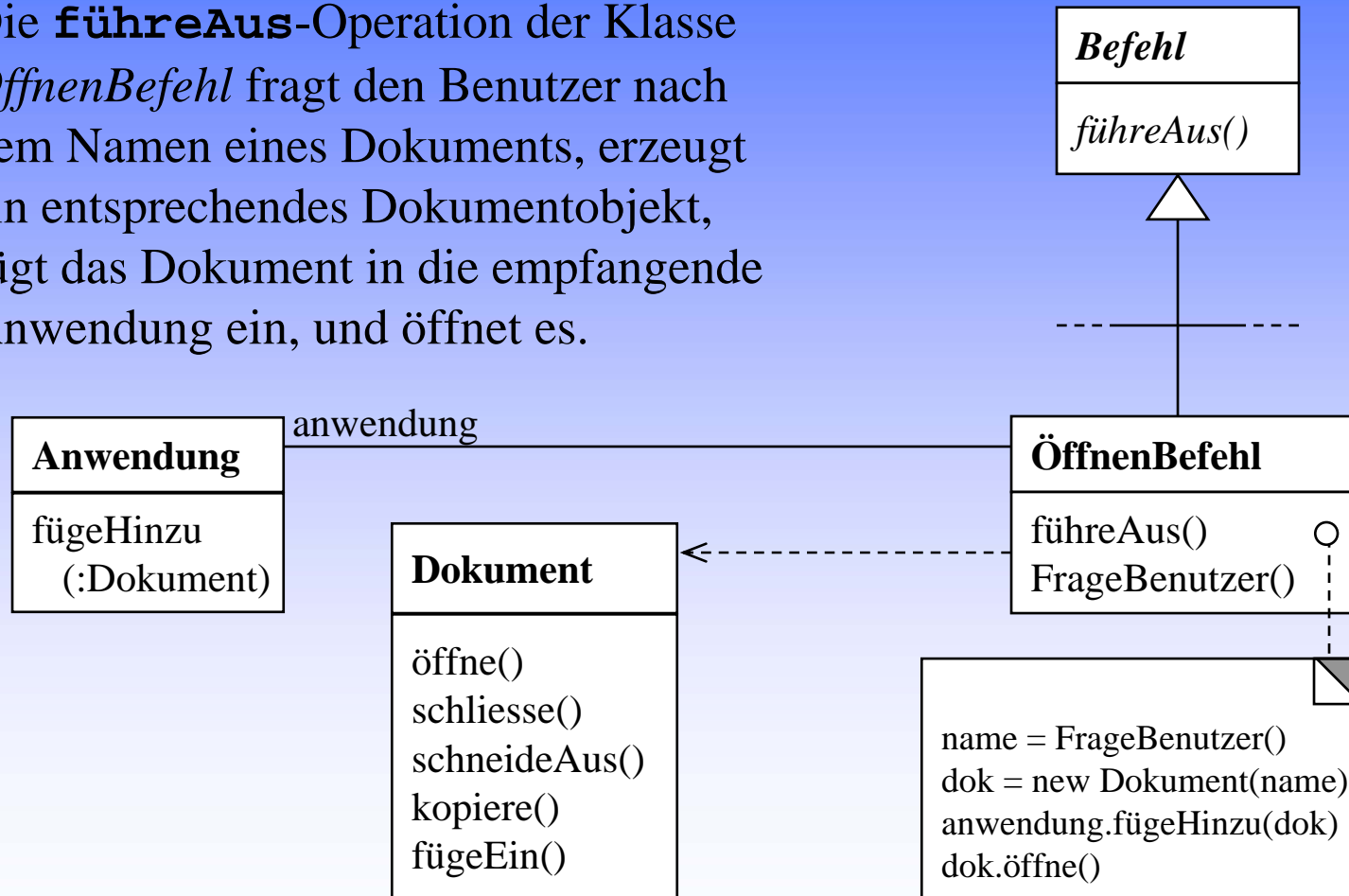
Ein Befehl enthält eine Methode **führeAus** und speichert das Objekt, an dem diese Operation durchgeführt werden soll.



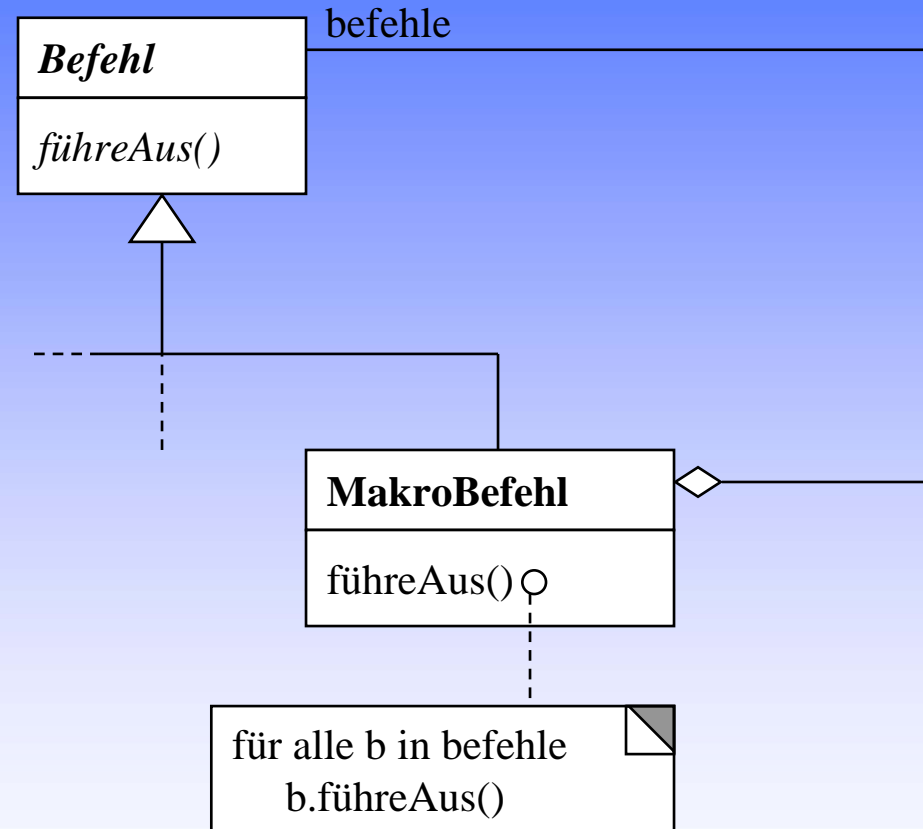
Beispiel eines Befehls

Befehl zum Öffnen eines Dokuments

Die **führeAus**-Operation der Klasse *ÖffnenBefehl* fragt den Benutzer nach dem Namen eines Dokuments, erzeugt ein entsprechendes Dokumentobjekt, fügt das Dokument in die empfangende Anwendung ein, und öffnet es.



Abfolge von Befehlen (Makrobefehl)



Befehl

Anwendbarkeit

- Wenn Objekte mit einer auszuführenden Aktion parametrisiert werden sollen (wie bei den MenüEintrag-Objekten).
- Wenn Anfragen zu unterschiedlichen Zeiten spezifiziert, aufgereiht und ausgeführt werden sollen.
- Wenn ein Rückgängigmachen von Operation (Undo) unterstützt werden soll.
- Wenn das Mitprotokollieren von Änderungen unterstützt werden soll (um System nach Absturz wiederherzustellen).
- Wenn ein System mittels komplexer Operationen strukturiert werden soll, die aus primitiven Operationen aufgebaut werden (Makrobefehle).



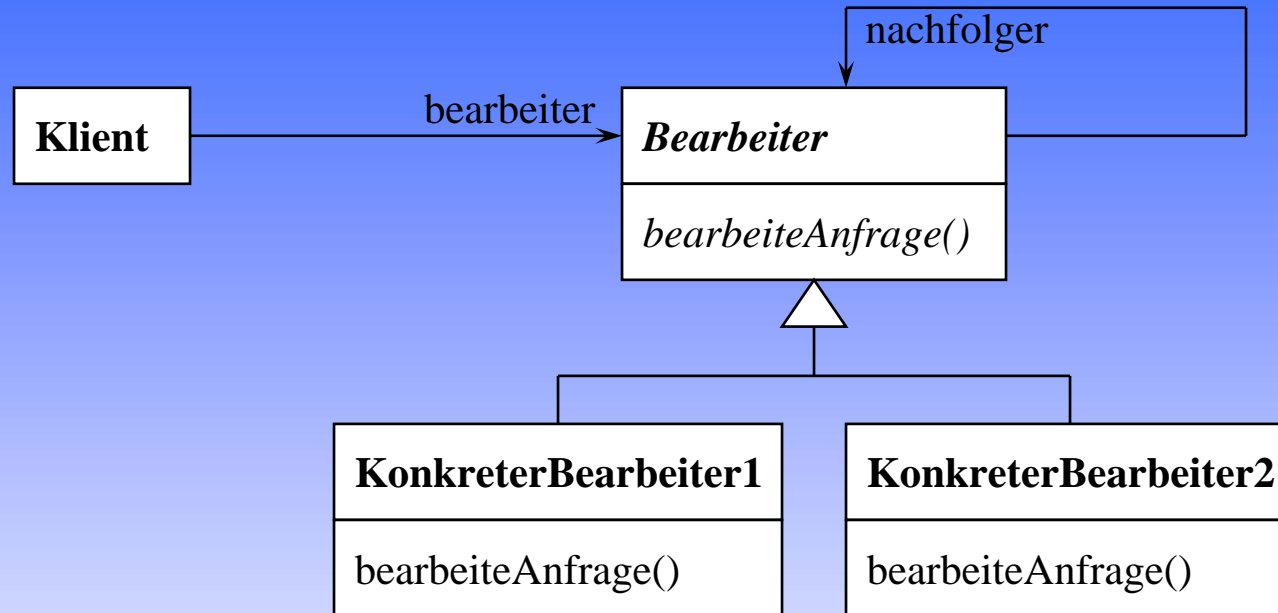
Zuständigkeitskette (Chain of Responsibility)

Zweck

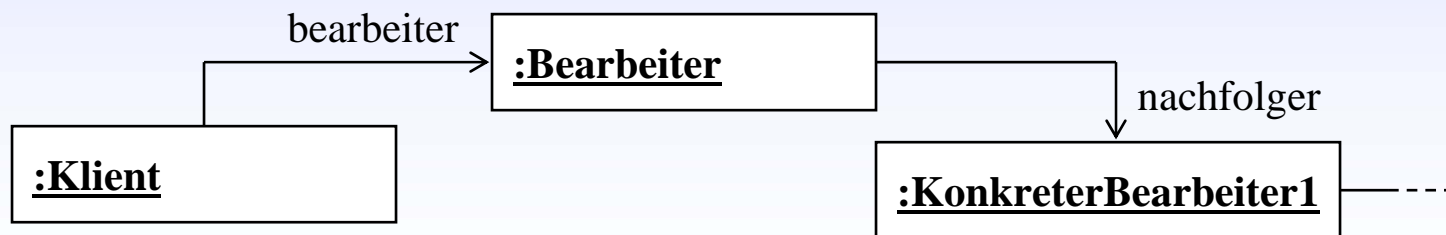
Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Anfrage zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.



Struktur für Zuständigkeitskette

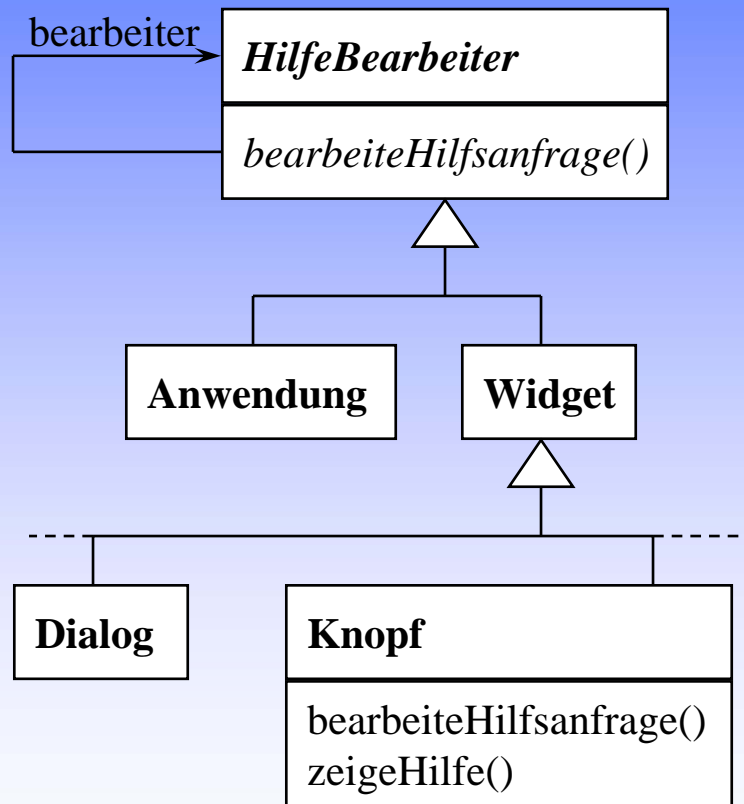


Typische Objektstruktur:

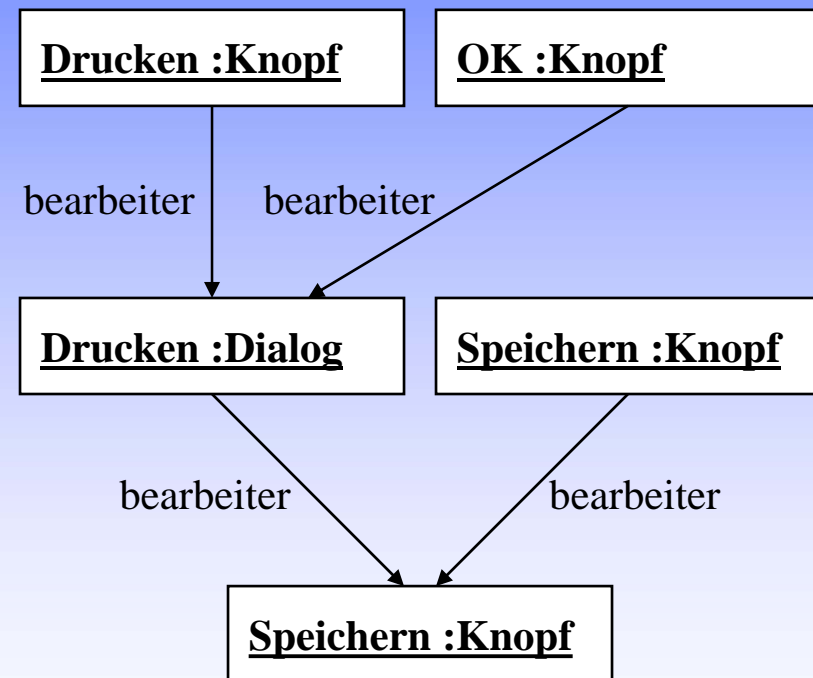


Beispiel Zuständigkeitskette

Hilfesystem für eine grafische Benutzeroberfläche



Objektstruktur:



Zuständigkeitskette

Anwendbarkeit

- Wenn mehr als ein Objekt eine Anfrage bearbeiten können soll und dasjenige Objekt, das dies tut, nicht *von vornherein* bekannt ist. Dieses Objekt muss zur Laufzeit automatisch bestimmt werden.
- Wenn eine Anfrage an eines von mehreren Objekten gerichtet werden soll, ohne den Empfänger explizit anzugeben.
- Wenn eine Menge Objekte, welche eine Anfrage bearbeiten sollen, dynamisch festgelegt wird.



Master/Slave

Zweck

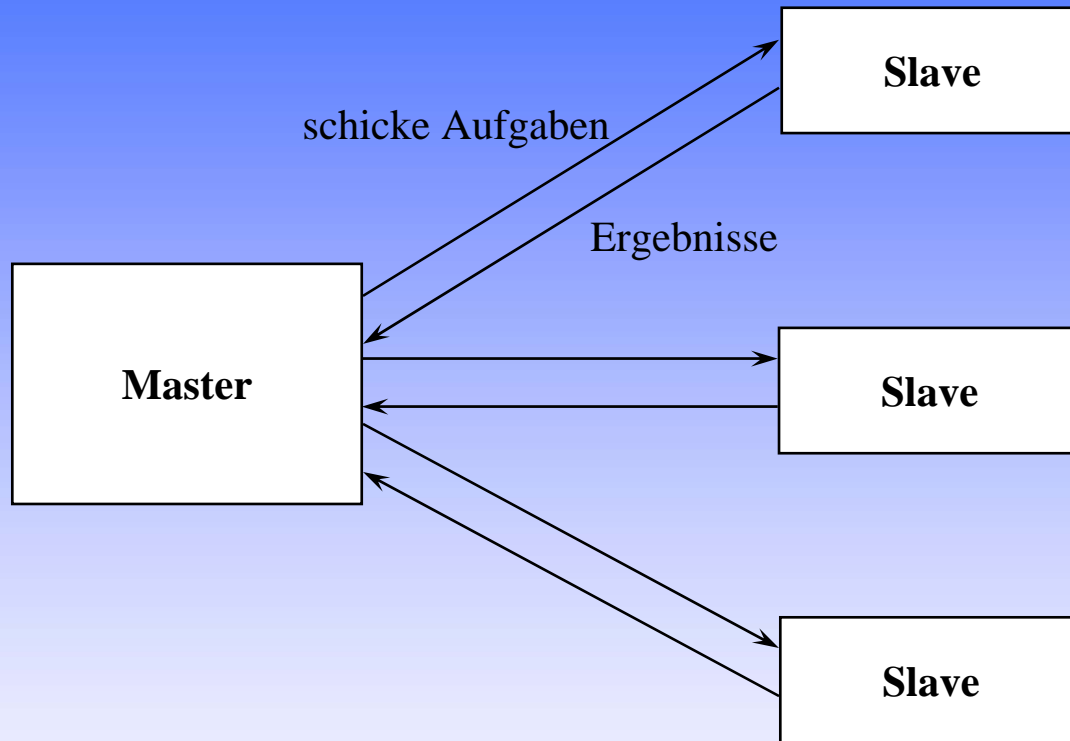
Das Master/Slave-Muster unterstützt Fehlertoleranz und parallele Berechnung. Eine Master-Komponente (Auftraggeber) verteilt die Arbeit an identische Slave-Komponenten (Arbeiter, Auftragnehmer) und berechnet das Endergebnis aus den Teilergebnissen, welche die Arbeiter zurückliefern.

Auch bekannt als

Master/Workers, Auftraggeber/Auftragnehmer

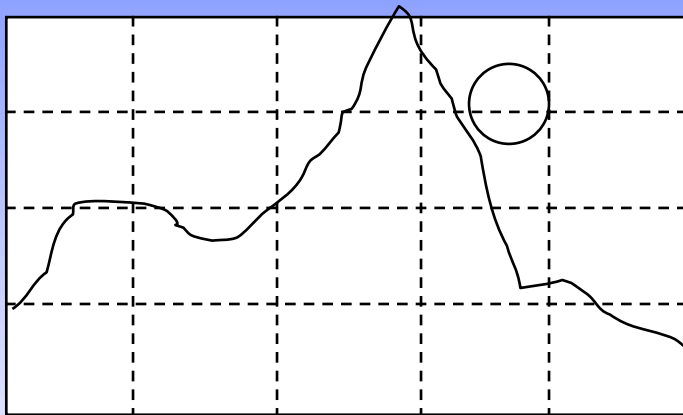


Struktur von Master/Slave



Beispiel von Master/Slave

Parallele Berechnung eines 3D Bildes



Der Master gibt rechteckige Teile des Bildes zur Berechnung an seine Arbeiter weiter und setzt das gesamte Bild aus den einzelnen Teilen zusammen.



Master/Slave

Anwendbarkeit

- Wenn es mehrere Aufgaben gibt, die unabhängig voneinander bearbeitet werden können.
- Wenn mehrere Prozessoren zur parallelen Verarbeitung zur Verfügung stehen.
- Wenn die Belastung der Arbeiter ausgeglichen werden soll.



Prozess-Steuerung (process control)

Zweck

Regulierung eines physikalischen (kontinuierlichen) Prozesses.

- System ohne Rückkoppelung
- System mit Rückkoppelung
 - Regelung mit Rückführung
 - Regelung mit Störgrößenaufschaltung



Prozess-Steuerung: Vokabular

Prozessvariablen: Eigenschaften des Prozesses, die gemessen werden können; folgende spezielle Arten werden häufig unterschieden.

Regelgröße: Prozessvariablen, deren Wert das System kontrollieren möchte.

Eingangsrößen: Prozessvariablen, die als Eingabewerte für den Prozess dienen.

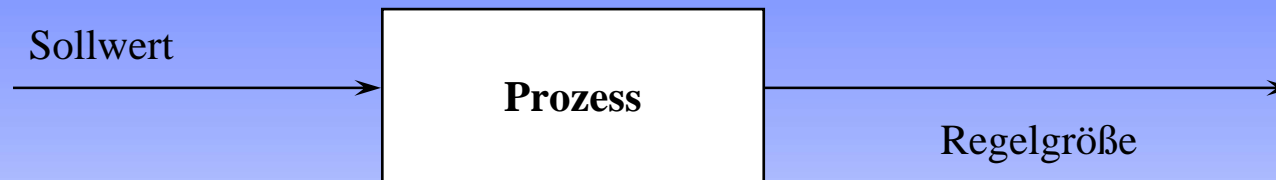
Stellgrößen: Prozessvariablen, deren Wert von der Steuerung verändert werden kann.

Sollwert: Der angestrebte Wert einer Regelgröße.



System ohne Rückkoppelung

Ein System in dem Informationen über Prozessvariablen nicht zur Steuerung des Systems verwendet werden.



Beispiel Heizung:

- gleichbleibender Brenner
- ein Zeittakt wird verwendet, um den Brenner in festgelegten Abständen aus- und anzuschalten.



System mit Rückkoppelung

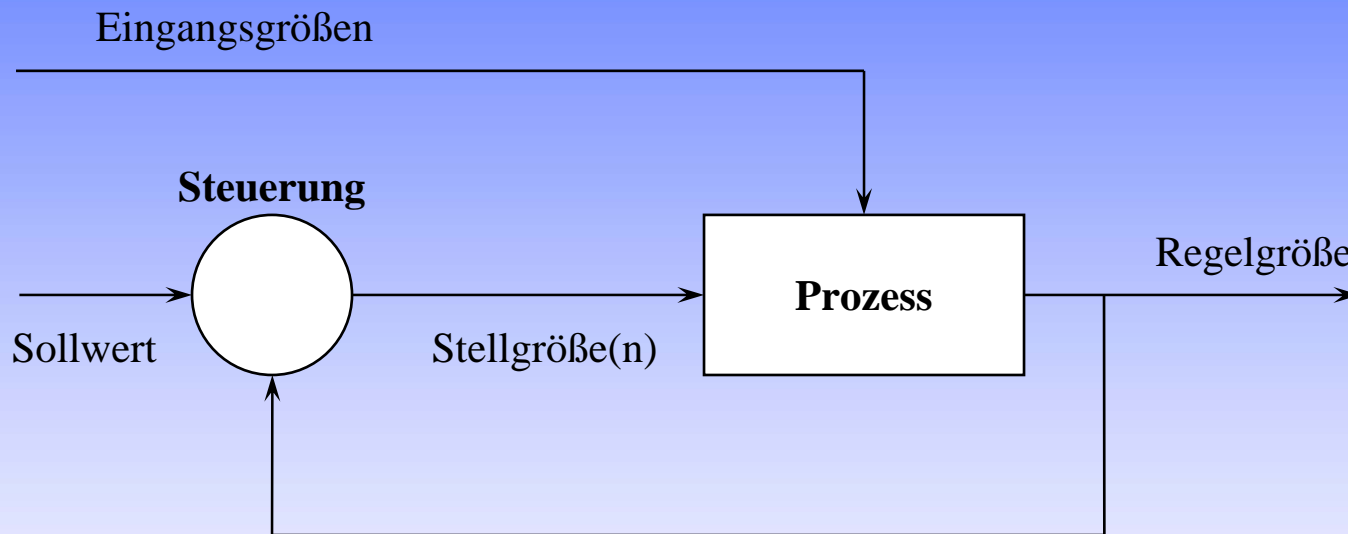
Ein System, in dem Prozessvariablen verwendet werden, um den Prozess zu steuern.

- Regelung mit Rückführung
- Regelung mit Störgrößenaufschaltung



Regelung mit Rückführung

Die Regelgröße wird gemessen und ihr Wert wird verwendet um eine oder mehrere Stellgrößen zu verändern.



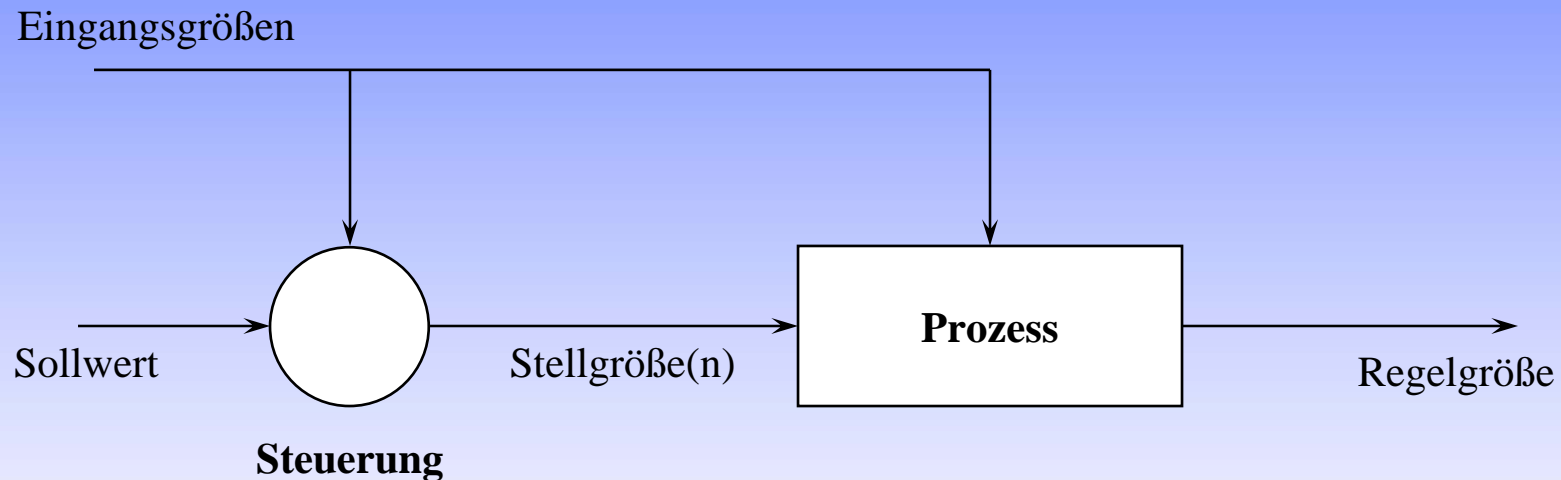
Beispiel Heizung:

Benutze ein Thermostat und schalte den Brenner an und aus wenn es notwendig ist, um eine bestimmte Temperatur zu erreichen (den Sollwert).



Regelung mit Störgrößenaufschaltung

Einige der Prozessvariablen werden gemessen und erwartete Störungen werden kompensiert, ohne auf eine Veränderung der Regelgröße zu warten.



Beispiel Heizung:

Messung der Außentemperatur und Einstellung des Brenner-Zeittaktes in Abhängigkeit von dieser Temperatur.



5. Virtuelle Maschinen

- Interpretierer
- Regelbasierter Interpretierer



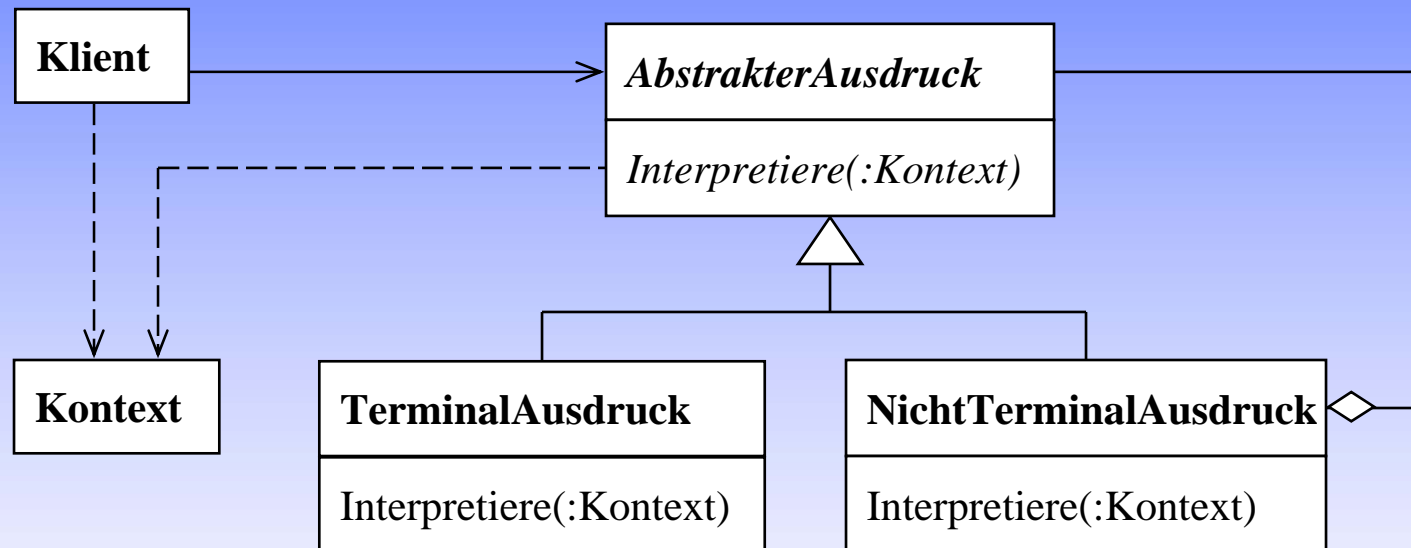
Interpretierer (Interpreter)

Zweck

Definiere für eine gegebene Sprache eine Repräsentation der Grammatik sowie einen Interpretierer, der die Repräsentation nutzt, um Sätze in der Sprache zu interpretieren.



Struktur des Interpretierers



Interpretierer

Anwendbarkeit

Wenn eine Sprache interpretiert werden muss und Ausdrücke der Sprache als abstrakte Syntaxbäume darstellbar sind. Das Interpretierermuster funktioniert am besten, wenn

- die Grammatik einfach ist. Bei komplexen Grammatiken wird die Klassenhierarchie zu groß und nicht mehr handhabbar. In diesem Falle stellen Werkzeuge wie Parsergeneratoren eine bessere Alternative dar.
- die Effizienz unproblematisch ist. Effiziente Interpretierer werden üblicherweise nicht durch Interpretation von Parsebäumen implementiert; sie transformieren die Bäume stattdessen in eine andere Form.



Interpreter vs. Kompositum vs. Besucher

- Der Interpreter und das Kompositum haben die gleiche Struktur. Man spricht von einem Interpreter, wenn Sätze einer Sprache repräsentiert und ausgewertet werden. Der Interpreter kann als Spezialfall des Kompositums gesehen werden.
- Ein Besucher kann dazu verwendet werden, das Verhalten eines jeden Knotens im abstrakten Syntaxbaum in einer einzigen Klasse zu kapseln.



Regelbasierter Interpretierer (Rule-Based System)

Zweck

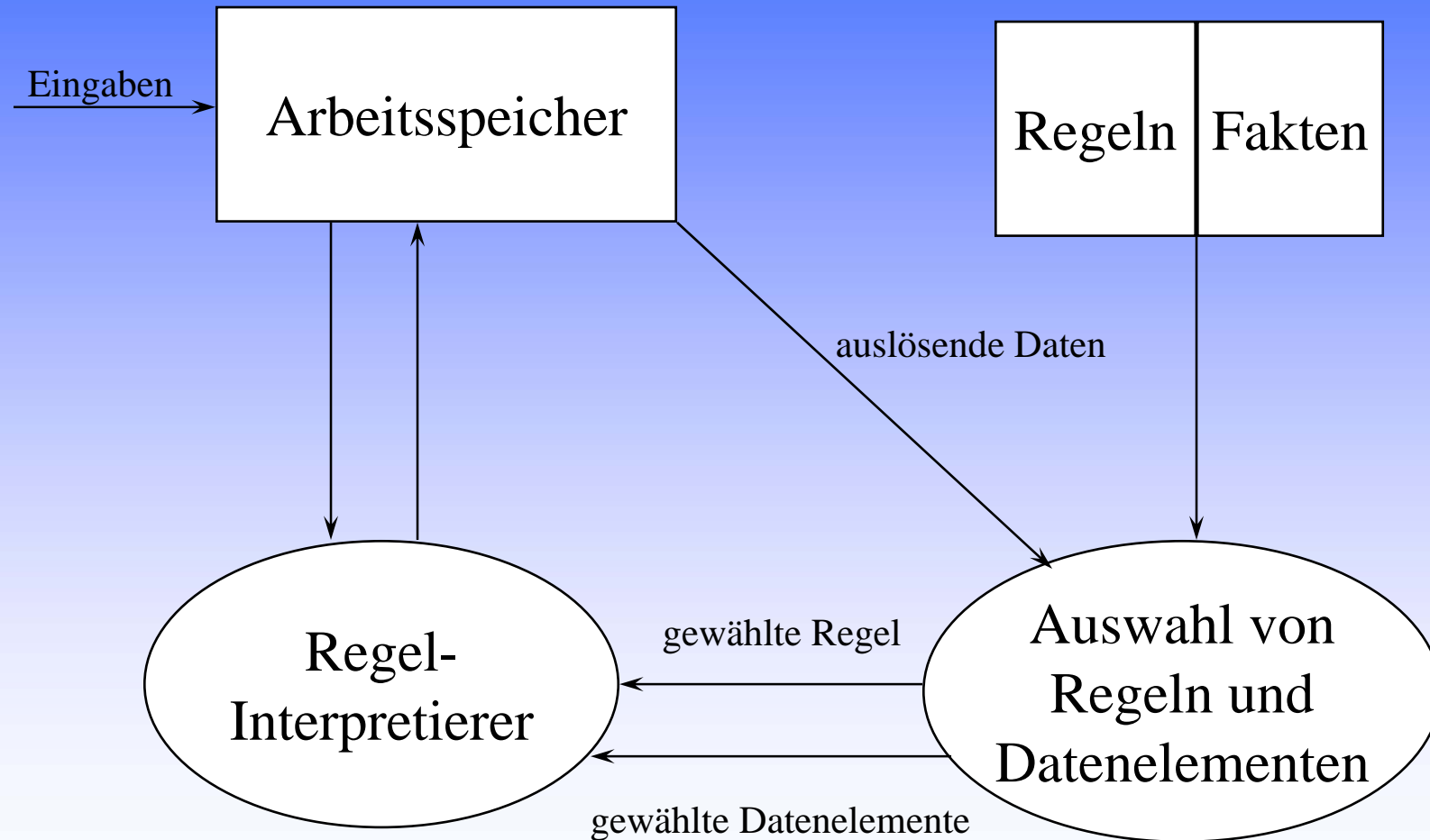
Löse ein Problem durch Anwendung einer Menge von Regeln. Eine Regel besteht aus einem Bedingungsteil und einem Aktionsteil. Falls der Bedingungsteil, angewandt auf eine Menge von Datenelementen im Arbeitsspeicher, wahr ergibt, dann kann der Aktionsteil ausgeführt werden. Der Aktionsteil ändert, ersetzt oder löscht Datenelemente, die im Bedingungsteil ausgewählt wurden, oder fügt neue Datenelemente hinzu.

Auch bekannt als

Regelbasiertes Expertensystem



Regelbasierter Interpretierer



Regelauswahl

Wenn mehr als eine Regel anwendbar ist, wird eine Auswahl nach folgenden Schritten getroffen:

1. Eine gegebene Regel darf auf ein Datenelement im Arbeitsspeicher höchstens einmal angewandt werden.
2. Wenn mehrere Regel-Datenelement-Kombinationen anwendbar sind, dann werden die Regeln ausgewählt, die mit jüngsten Datenelementen operieren.
3. Wähle die Regeln mit der höchsten „Spezifität“, d.h. diejenigen, dessen Bedingungen die meisten Elemente im Arbeitsspeicher benötigen.
4. Wähle die Regel a) nach ihrer Ordnung oder b) zufällig.



Regelbasierter Interpretierer

Anwendbarkeit

- Wenn eine Problemlösung am besten als eine Menge von Bedingungs-Aktions-Regeln formuliert werden kann, z.B. bei Diagnose- oder Konfigurierungsaufgaben.
- Wenn der Aktionsteil nur einfache Operationen an den Datenelementen erfordert (keine Schleifen, keine Rekursion, keine Prozeduraufrufe, um Arbeitsspeicher zu modifizieren).



6. Bequemlichkeits-Muster

- Bequemlichkeits-Methode
- Bequemlichkeits-Klasse
- Fassade
- Null-Objekt



Bequemlichkeits-Methode

Zweck

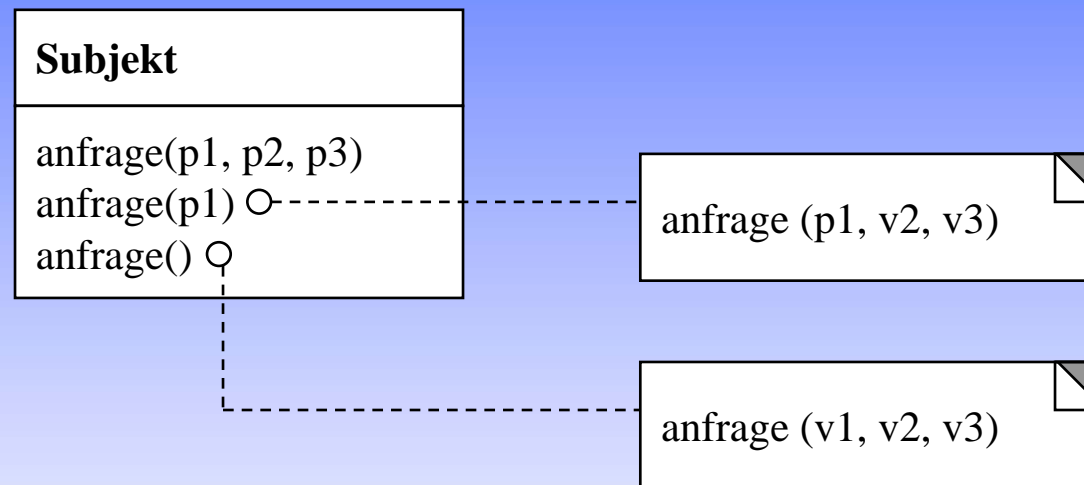
Vereinfachen des Methodenaufrufs durch die Bereitstellung häufig genutzter Parameterkombinationen durch zusätzliche Methoden (Überladen).

Anwendbarkeit

Wenn Methodenaufrufe häufig mit den gleichen Parametern auftreten.



Bequemlichkeits-Methode



Bequemlichkeits-Klasse

Zweck

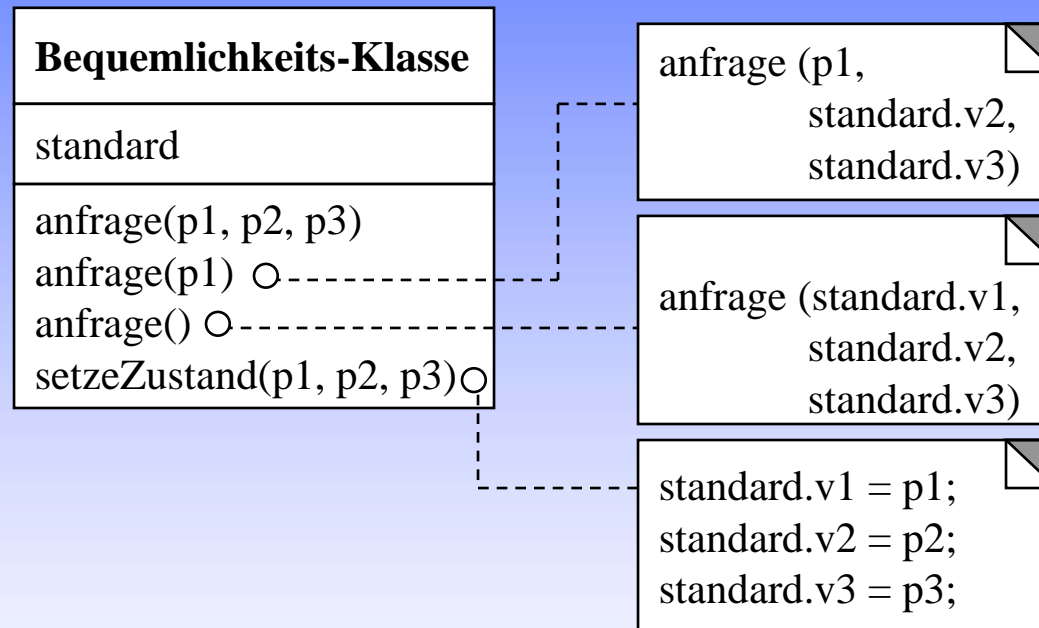
Vereinfache den Methodenaufruf durch Bereithaltung der Parameter in einer speziellen Klasse.

Anwendbarkeit

Wenn Methoden häufig mit den gleichen Parametern aufgerufen werden, die sich nur selten ändern.



Bequemlichkeits-Klasse



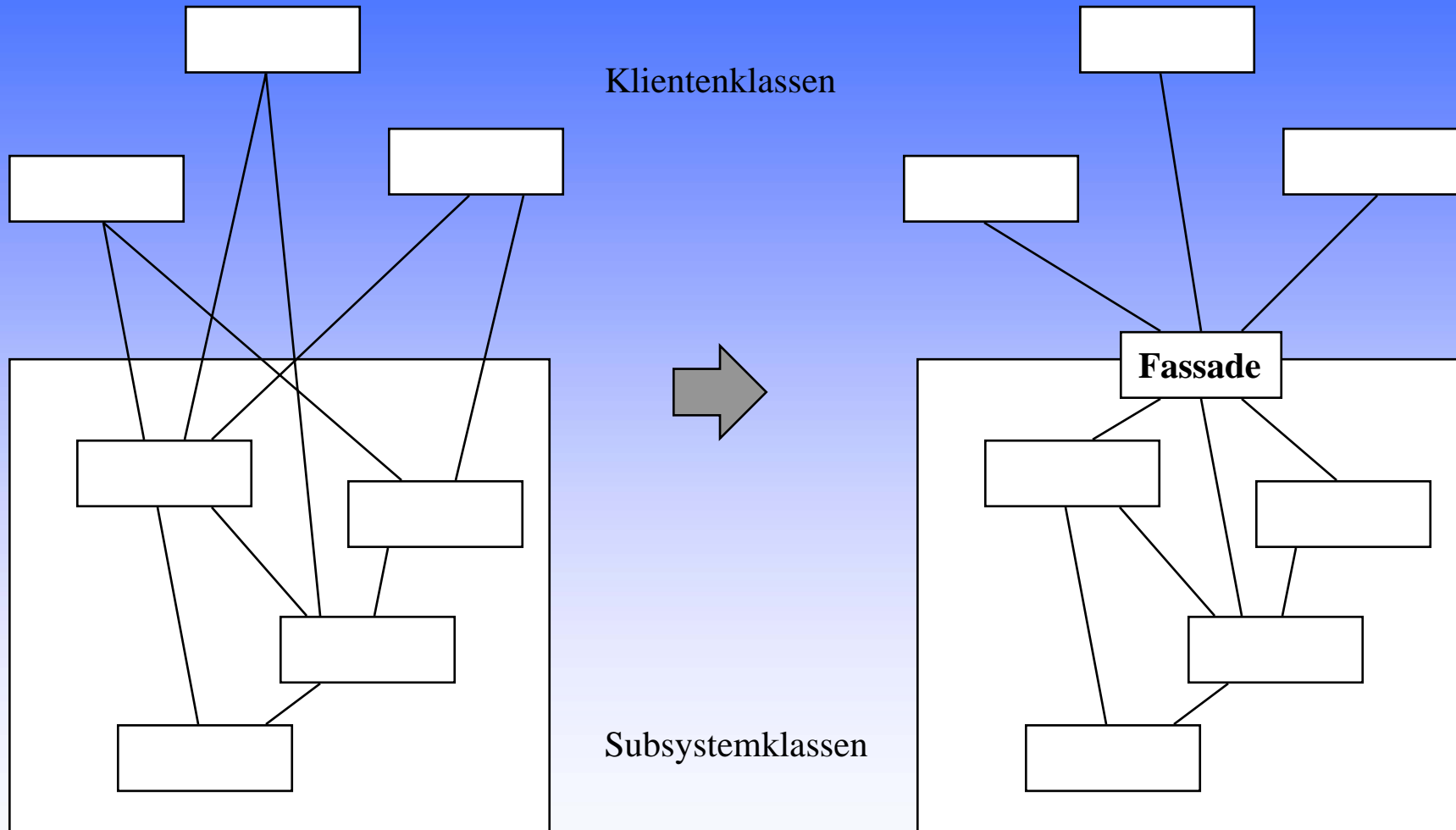
Fassade (Facade)

Zweck

Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.



Fassade



Fassade

Anwendbarkeit

- Wenn eine einfache Schnittstelle zu einem komplexen Subsystem angeboten werden soll. Eine Fassade kann eine einfache voreingestellte Sicht auf das Subsystem bieten, die den meisten Klienten genügt.
- Wenn es viele Abhängigkeiten zwischen den Klienten und den Implementierungsklassen einer Abstraktion gibt. Die Einführung einer Fassade entkoppelt die Subsysteme von Klienten und anderen Subsystemen.
- Wenn Subsysteme in Schichten aufgeteilt werden sollen. Man verwendet eine Fassade, um einen Eintrittspunkt zu jeder Subsystemschicht zu definieren.



Null-Objekt

Zweck

Stelle einen Stellvertreter zur Verfügung, der die gleiche Schnittstelle bietet, aber nichts tut. Das Null-Objekt kapselt die Implementierungs-Entscheidungen (wie genau es „nichts tut“) und versteckt diese Details vor seinen Mitarbeitern.

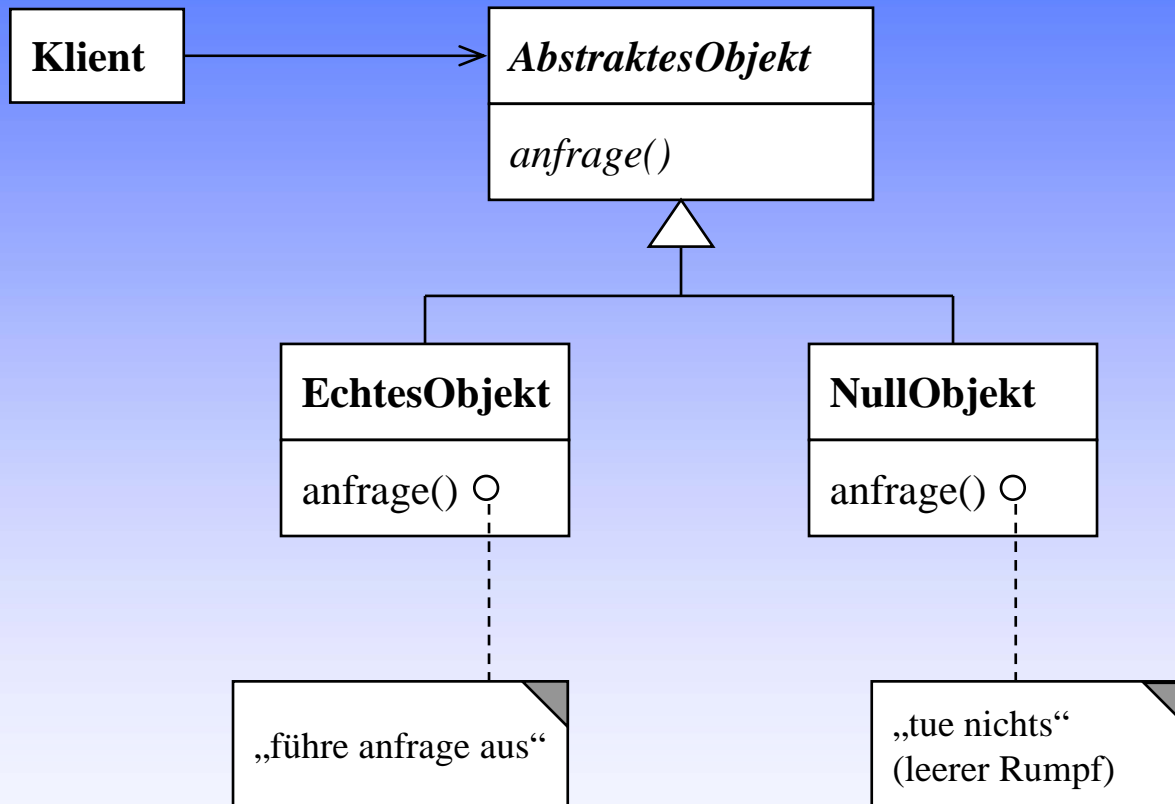
Motivation

Es wird verhindert, dass der Code mit Tests gegen Null-Werte verschmutzt wird, wie:

```
if (thisCall.callingParty != NIL)
    thisCall.callingParty.action()
```



Struktur des Null-Objektes



Null-Objekt

Anwendbarkeit

- Wenn ein Objekt Mitarbeiter benötigt und einer oder mehrere von ihnen nichts tun sollen.
- Wenn Klienten sich nicht um den Unterschied zwischen einem echten Mitarbeiter und einem der nichts tut kümmern sollen.
- Wenn das „tue nichts“-Verhalten von verschiedenen Klienten wiederverwendet werden soll.

