Das Fliegengewicht-Muster und das Dekorierer-Muster

Vortrag von
Yvonne Gabriel

Das Fliegengewicht-Muster

Gliederung

1. Zweck
2. auch bekannt als
3. Motivation
4. Anwendbarkeit
5. Struktur
6. Teilnehmer
7. Interaktionen
8. Konsequenzen
9. Implementierung
10. bekannte Verwendungen
11. verwandte Muster

1. Zweck

- kleine Objekte gemeinsam verwenden
- → effiziente Verwendung von großen Mengen
- → Speicherplatz sparen

2. auch bekannt als

- Flyweight

3. Motivation

- Verwendung vieler kleiner Objekte manchmal sinnvoll
- naive Implementierung benötigt viel Speicher
- Fliegengewicht-Muster:
  gemeinsame Nutzung von kleinen Objekten
  → annehmbare Kosten

Beispiel: Dokumenteditor

- Objekte: Tabellen und Grafiken
- selten einzelne Zeichen
- Grund: hoher Speicherverbrauch (und hohe Laufzeit)

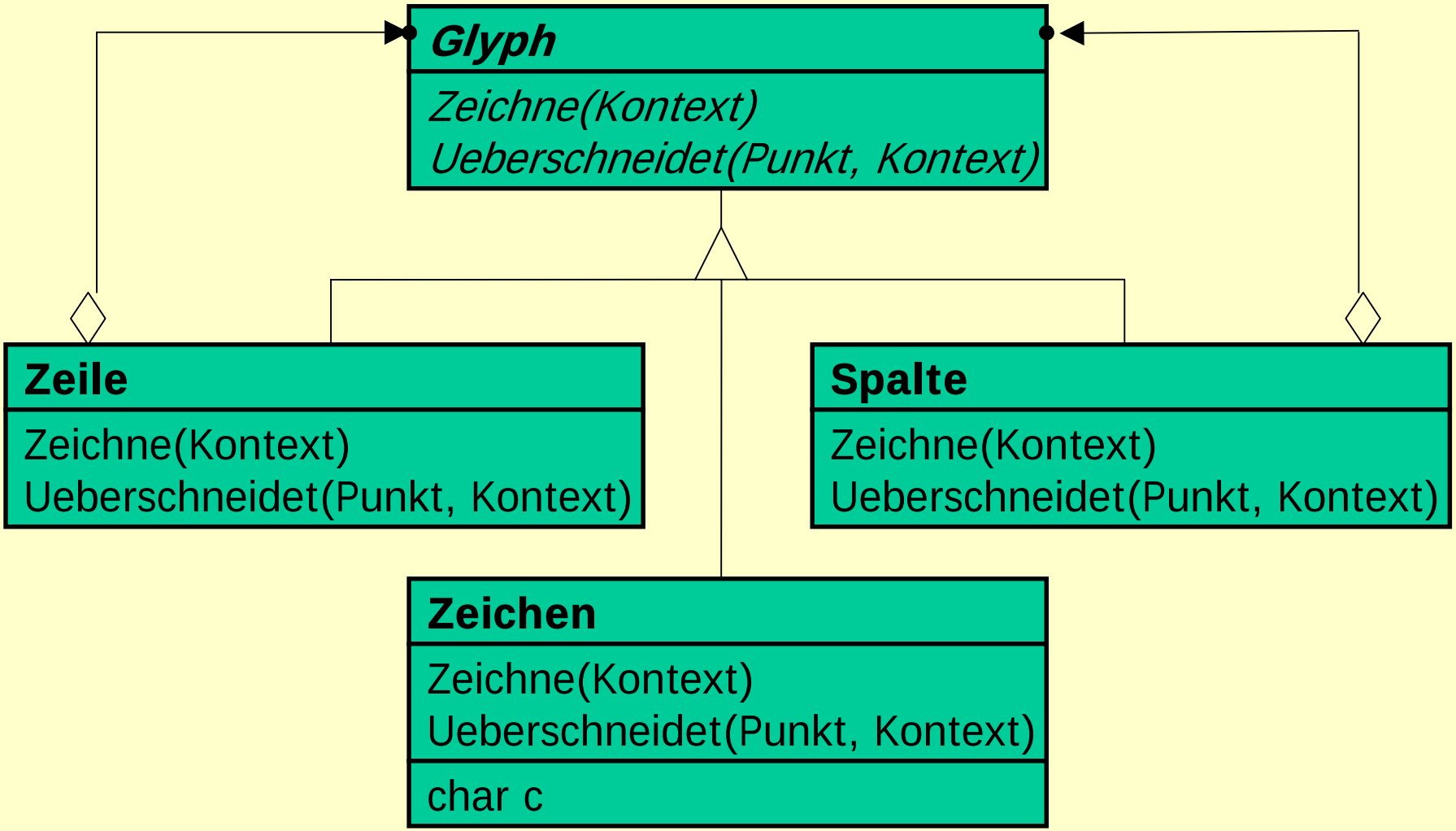
3. Motivation

- Fliegengewicht:
  - Objekt, welches in mehreren Kontexten genutzt werden kann
  - agiert in jedem Kontext unabhängig
  - macht keine Annahmen über Kontext
  - modelliert gehäuft auftretendes Konzept
- Unterscheidung:
  - intrinsischer Zustand
  - extrinsischer Zustand (als Parameter)

3. Motivation

- im Beispiel:
  - Zeichen ist Fliegengewicht; speichert nur Zeichencode (intrinsisch)
  - nicht: Positionierung, Formatierung, etc. (extrinsisch)
  - Zeilenobjekt kennt Positionen für seine Buchstaben → Parameter

3. Motivation



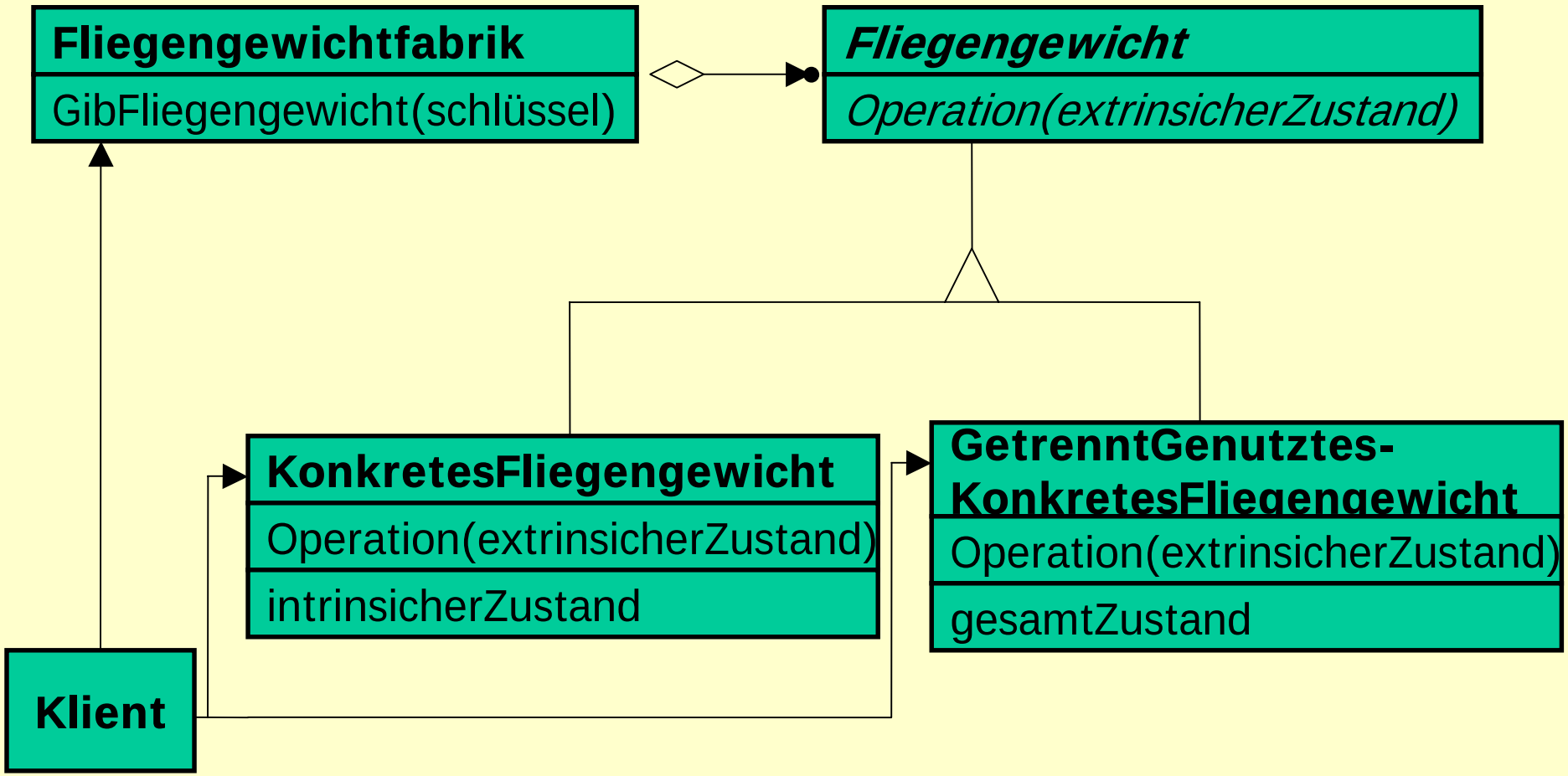
3. Motivation

- Gewinn:
  wesentlich weniger verwendete Objekte
- bei wenigen Kontextwechseln immense Einsparungen

4. Anwendbarkeit

- alle folgenden Bedingungen:
  - große Menge von Objekten
  - nur deshalb hohe Speicherkosten
  - Großteil des Zustands → extrinsisch
  - viele Gruppen von Objekten → wenige gemeinsam genutzte Objekte
  - Anwendung unabhängig von Objektidentitäten

5. Struktur


Fliegengewichtfabrik
GibFliegengewicht(schlüssel)
Fliegengewicht
Operation(extrinsicherZustand)
KonkretesFliegengewicht
Operation(extrinsicherZustand)
intrinsicherZustand
GetrenntGenutztes-
KonkretesFliegengewicht
Operation(extrinsicherZustand)
gesamtZustand
Klient

6. Teilnehmer

- Fliegengewicht:
  - deklariert Schnittstelle
- KonkretesFliegengewicht:
  - implementiert Schnittstelle
- GetrenntGenutztesKonkretesFliegengewicht:
  - nicht gemeinsam genutzt

6. Teilnehmer

- FliegengewichtFabrik
  - erzeugt und verwaltet Fliegengewichte
  - stellt korrekte Benutzung sicher
- Klient:
  - verwaltet Referenz auf Fliegengewichte
  - speichert oder berechnet extrinsischen Zustand der Fliegengewichte

7. Interaktionen

- klare Trennung von intrinsischem und extrinsichem Zustand
  → Klient übergibt extrinsischen an Fliegengewicht
- Erzeugung der Fliegengewichte immer über FliegengewichtFabrik; nie direkt

8. Konsequenzen

- möglicherweise höhere Laufzeitkosten
- Speicherplatzgewinn

- oft gemeinsam mit Kompositionsmuster verwendet

9. Implementierung

- extrinsischen Zustand entfernen:
  - identifizieren und entfernen
  - im Beispiel: typographische Informationen
- gemeinsam genutzte Objekte verwalten:
  - nur über Fabrik erzeugen
  - im Beispiel: Fabrik hat Tabelle von Fliegengewichten, indiziert durch Zeichencode
  - ggf. Referenzzählung oder automatische Speicherbereinigung

10. Bekannte Verwendungen

- InterViews3.0: Dokumenteditor Doc
- ET++

11. Verwandte Muster

- Kompositum
  (oft kombiniert, gemeinsam genutzte Blätter)
- Zustandsmuster und Strategiemuster
  (deren Objekte werden am besten als Fliegengewichte implementiert)

Fragen?

Das Dekorierer-Muster

Gliederung

1. Zweck
2. auch bekannt als
3. Motivation
4. Anwendbarkeit
5. Struktur
6. Teilnehmer
7. Interaktionen
8. Konsequenzen
9. Implementierung
10. bekannte Verwendungen
11. verwandte Muster

1. Zweck

- Objekte dynamisch um Funktionalitäten erweitern

2. auch bekannt als

- gebundener Umwickler (Wrapper)

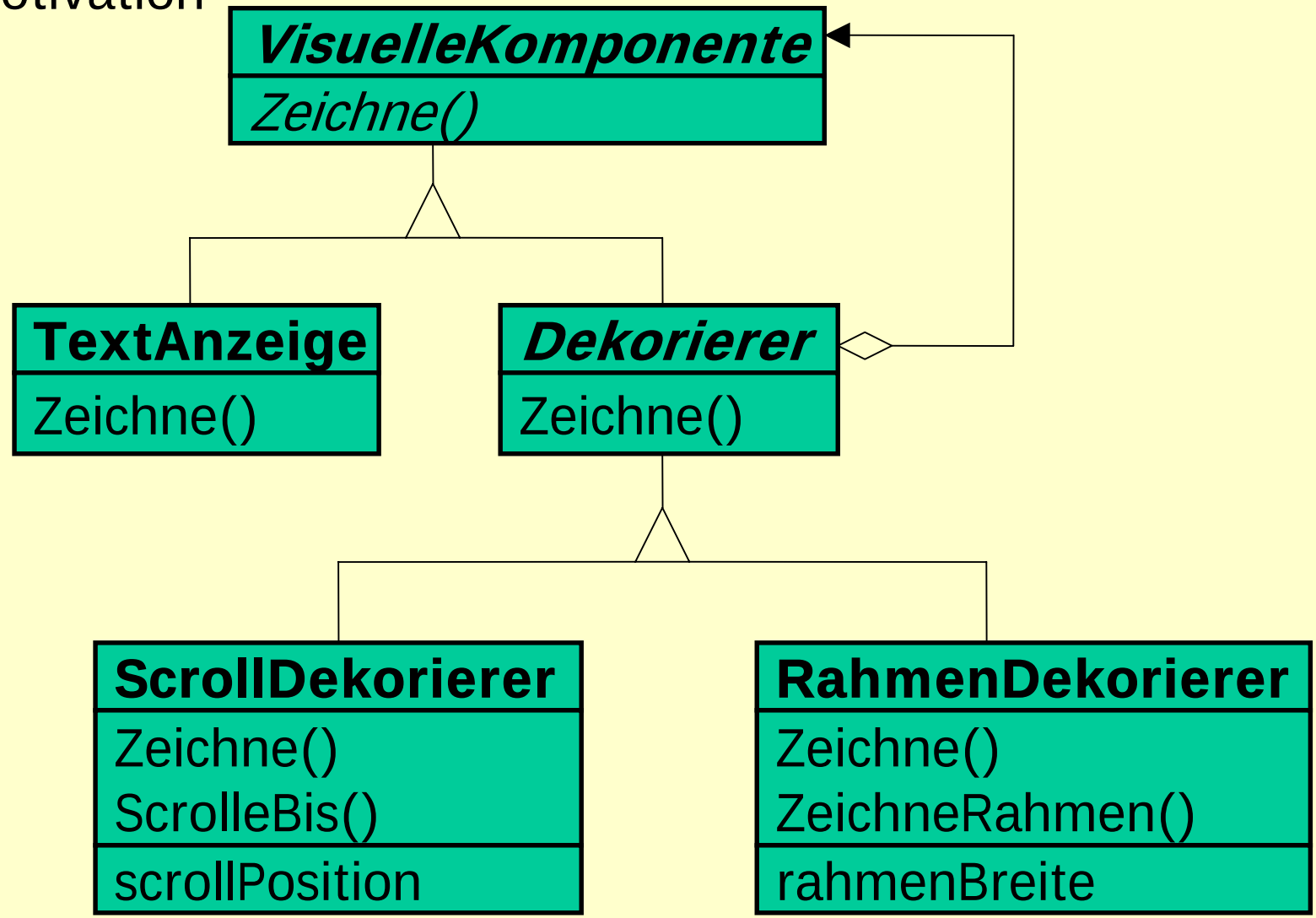
3. Motivation

- Funktionalität einzelner Objekte erweitern, ohne Klasse zu ändern
- Beispiel:
  grafische Benutzungsschnittstelle:
  - Umrahmung, Scrollen für beliebige Komponenten

3. Motivation

- Mögliche Lösung: Vererbung
  Nachteile:
  - unflexibel: Rahmen ist statisch
- flexibler: Dekorierer
  - Komponente wird in Objekt (Dekorierer) eingeschlossen
  - Dekorierer fügt Rahmen hinzu
  - Schnittstelle entspricht der der dekorierten Komponente

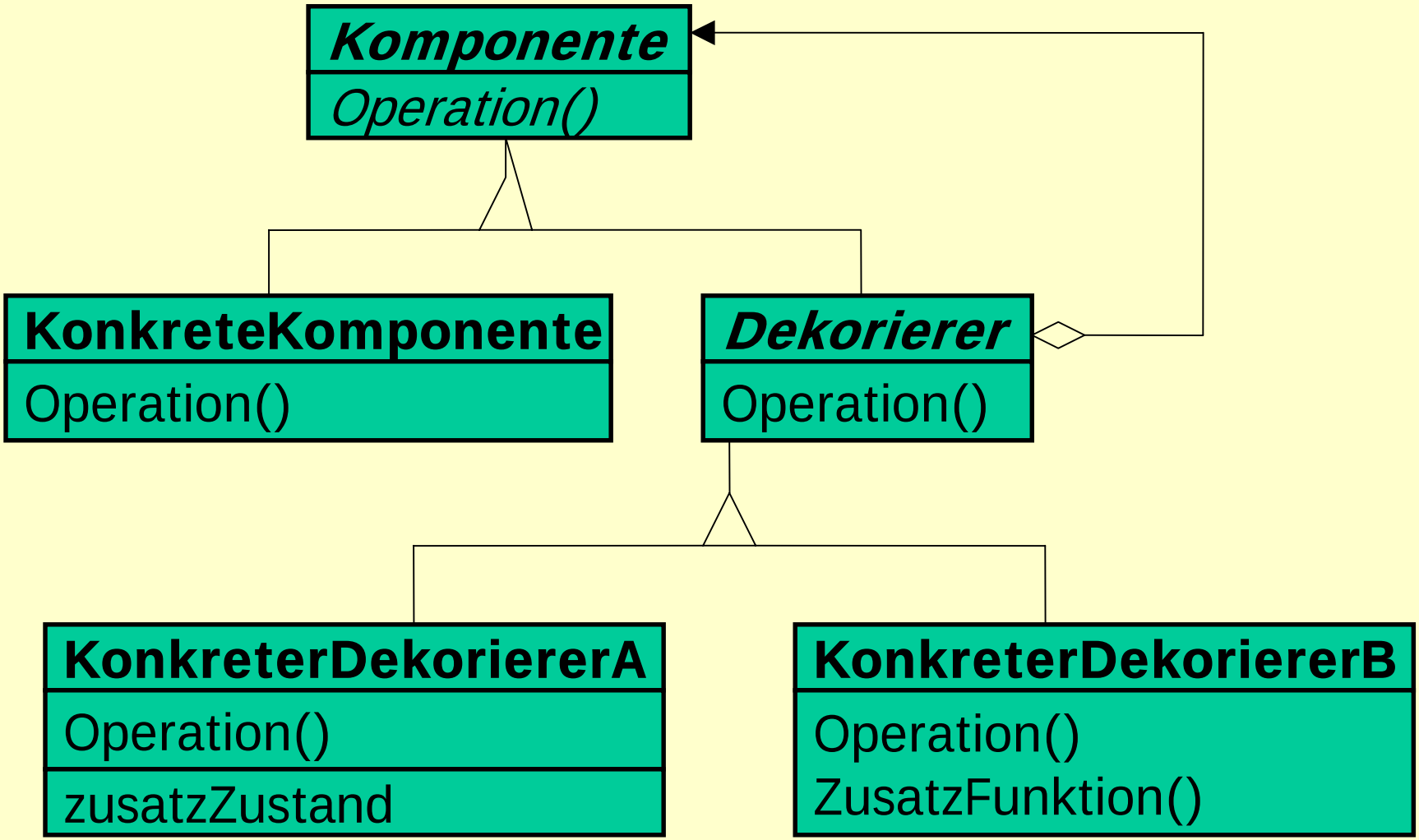
3. Motivation



4. Anwendbarkeit

- einzelnen Objekten dynamische und transparente zusätzliche Funktionalitäten geben
- zugefügte Funktionalitäten wieder entfernen
- Erweiterung durch Vererbung nicht praktisch durchführbar

5. Struktur


Komponente
Operation()
KonkreteKomponente
Operation()
Dekorierer
Operation()
KonkreterDekoriererA
Operation()
zusatzZustand
KonkreterDekoriererB
Operation()
ZusatzFunktion()

6. Teilnehmer

- Komponente:
  - definiert Schnittstelle
- KonkreteKomponente:
  - definiert erweiterbares Objekt

6. Teilnehmer

- Dekorierer:
  - verwaltet Referenz auf Komponentenobjekt
  - definiert entsprechende Schnittstelle
- konkreter Dekorierer:
  - leitet Anfragen weiter
  - Zusatzfunktionalitäten

7. Interaktionen

- Dekorierer leitet Anfragen an Komponente weiter
- kann zusätzliche Operationen durchführen

8. Konsequenzen

- Flexibilität
- kein Überfrachten mit Funktionalität
- leicht erweiterbar
- Dekorierer und seine Komponente nicht identisch!

9. Implementierung

- Schnittstellenkonformanz:
  - Schnittstelle des Dekoriererobjekts $\leftrightarrow$ Schnittstelle der dekorierten Komponente
- Weglassen der abstrakten Dekoriererklasse
  - nur eine Erweiterung

9. Implementierung

- Komponentenklassen leichtgewichtig lassen:
  - Schnittstelle, nicht Daten
  - Zustände erst in Unterklassen

9. Implementierung

- Hüllenänderung vs. Ändern des Inneren eines Objektes:
  - Dekorierer = Hülle, die das Verhalten ändert
  - Alternative: Inneres anpassen

    ➔ **Strategiemuster**

10. Bekannte Verwendungen

- viele objektorientierte Klassenbibliotheken zur Erstellung von Benutzerschnittstellen (InterViews, ET++, ObjectWorks\Smalltalk)
- ET++-Stream-Klassen

11. Verwandte Muster

- Proxy
- Adapter
- Kompositum
- Strategie

Fragen?

Frohe Weihnachten
und einen
guten Rutsch!