DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

ENTWURFSMUSTER

Lars Briem

(briem.lars@googlemail.com)

Duale Hochschule Baden Württemberg - Standort Karlsruhe

Muster

- Warum Muster
  - Muster beschreiben wiederkehrende Probleme in unserer Umwelt
    - Inklusive des Kerns einer Lösung dafür
    - Lösungen können beliebig oft auf unterschiedliche Art ausgeführt werden
  - Vergleichbare Muster existieren in der Architektur und Natur

  $\Rightarrow$ Muster helfen Probleme effizient zu lösen

Entwurfsmuster

- Elemente wiederverwendbarer objekt-orientierter Software
  - Helfen Probleme zu lösen
  - Liefern ein erprobtes Konzept
  - Basieren auf realen Entwicklungen
  - Offenbaren Beziehungen tiefergehender Strukturen und Mechanismen

$\Rightarrow$ IKEA-Baukastensystem für OOP

Nutzen von Entwurfsmustern

- Vermittlung von Wissen auf abstraktem Niveau
  - Räder werden nicht immer wieder neu erfunden
- Ausprägung einer höherwertigen Sprache in OOP
  - Vereinfachen und beschleunigen die Kommunikation zwischen Entwicklern
- Helfen komplexer werdende Softwaresysteme zu beherrschen
  - Größere Bausteine helfen, den Überblick zu behalten
  - Siehe auch integrierte Schaltkreise in der Elektronik

Gliederung von Entwurfsmustern

- Zweck bzw. Verwendung
  - Erzeugungsmuster
  - Strukturmuster
  - Verhaltensmuster
- Geltungsbereich
  - Auf Klassenebene
    - Statisch
    - Wird beim Kompilieren festgelegt
  - Auf Objektebene
    - Dynamisch
    - Wird zur Laufzeit festgelegt

Erzeugungsmuster

- Trennen die Erstellung der Objekte von deren Verwendung
- Konkrete Instanzen werden einfacher ersetzbar für anderes Verhalten
- System wird unabhängig von der Zusammensetzung bzw. Implementierung der Objekte
- Kapseln Wissen über
  - Konkret verwendete Klasse bzw. Implementierung
  - Erstellung und Kombination der Objekte

Strukturmuster

- Kombinieren Klassen und Objekte, um größere Strukturen zu schaffen
- Kombination von mehreren Interfaces
- Übersetzung von einem zum anderen Interface
- Kombination von Funktionalität zur Laufzeit
- Sparen von Ressourcen bzw. Laufzeit

Verhaltensmuster

- Zuweisung von Verantwortlichkeiten an Objekte
- Austausch von Algorithmen bzw. Verhalten
- Kommunikation zwischen Objekten
- Steuerung des Kontrollflusses einer Anwendung zur Laufzeit (auch komplexer)
  - Verbindung zwischen Elementen der Software

Übersicht der Entwurfsmuster

Ein Auszug der "Gang of Four" Muster

	Erzeugungsmuster	Strukturmuster	Verhaltensmuster
Klasse	Fabrikmethode	Adapter(Klasse)	Interpreter Schablonenmethode
Objekt	Abstrakte Fabrik Einzelstück Erbauer Prototyp	Adapter(Objekt) Brücke Dekorierer Fassade Fliegengewicht Kompositum Stellvertreter	Beobachter Besucher Iterator Kommando Memento Strategie Vermittler Zustand Zuständigkeitskette

Integration in die Entwicklung

- Während dem kompletten Entwicklungsprozess einsetzbar
  - Entwurf, Implementierung, Refactoring
- Hauptsächlich als Kommunikationsmittel einsetzen
  - Implementierungsdetails können stark variieren
- Vorsicht für "zu viel" Entwurfsmustern
  - Bei unnötiger Verwendung von Entwurfsmustern wird der Code unnötig komplex
  - Nur einsetzen, wenn notwendig

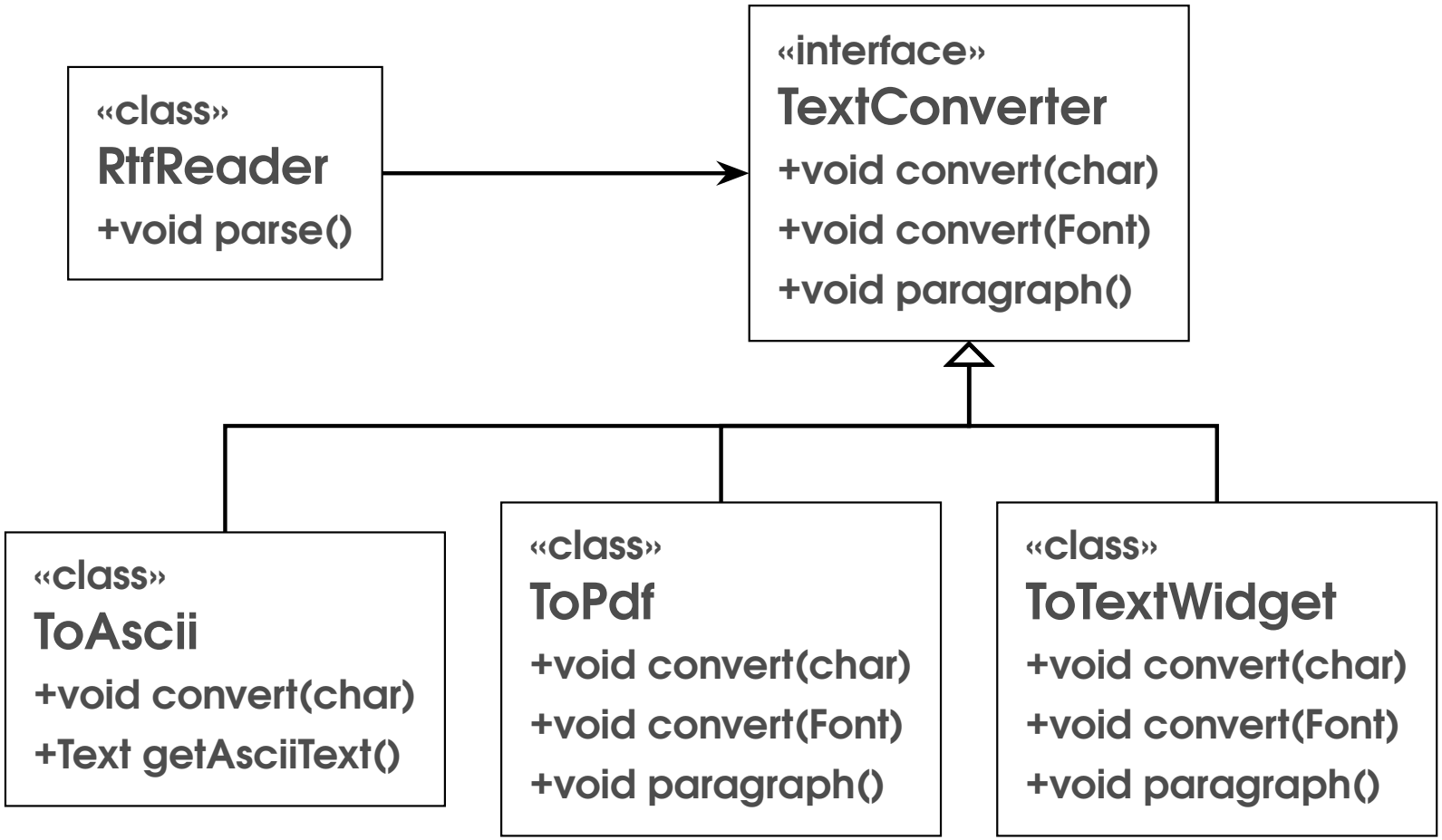
Erbauer

- Trennung der Erstellung von komplexen Objekten von ihrer Repräsentation
- Gleicher Erstellungsprozess bzw. Konstruktionsprozess kann unterschiedliche Repräsentation erzeugen
- Klassifikation
  - Objektbasiertes Erzeugungsmuster
  - Eher kurzlebig

Erbauer – Motivation

- Wiederverwendung einer komplexen Logik zur Umwandlung von Objekten
- Erzeugungslogik für verschiedene Formate von Konvertierungs- bzw. Konstruktionslogik trennen
- Schrittweise Erzeugung von komplexen Produkten
- Wiederverwendung der Erzeugungs- bzw. Konstruktionslogik unabhängig voneinander

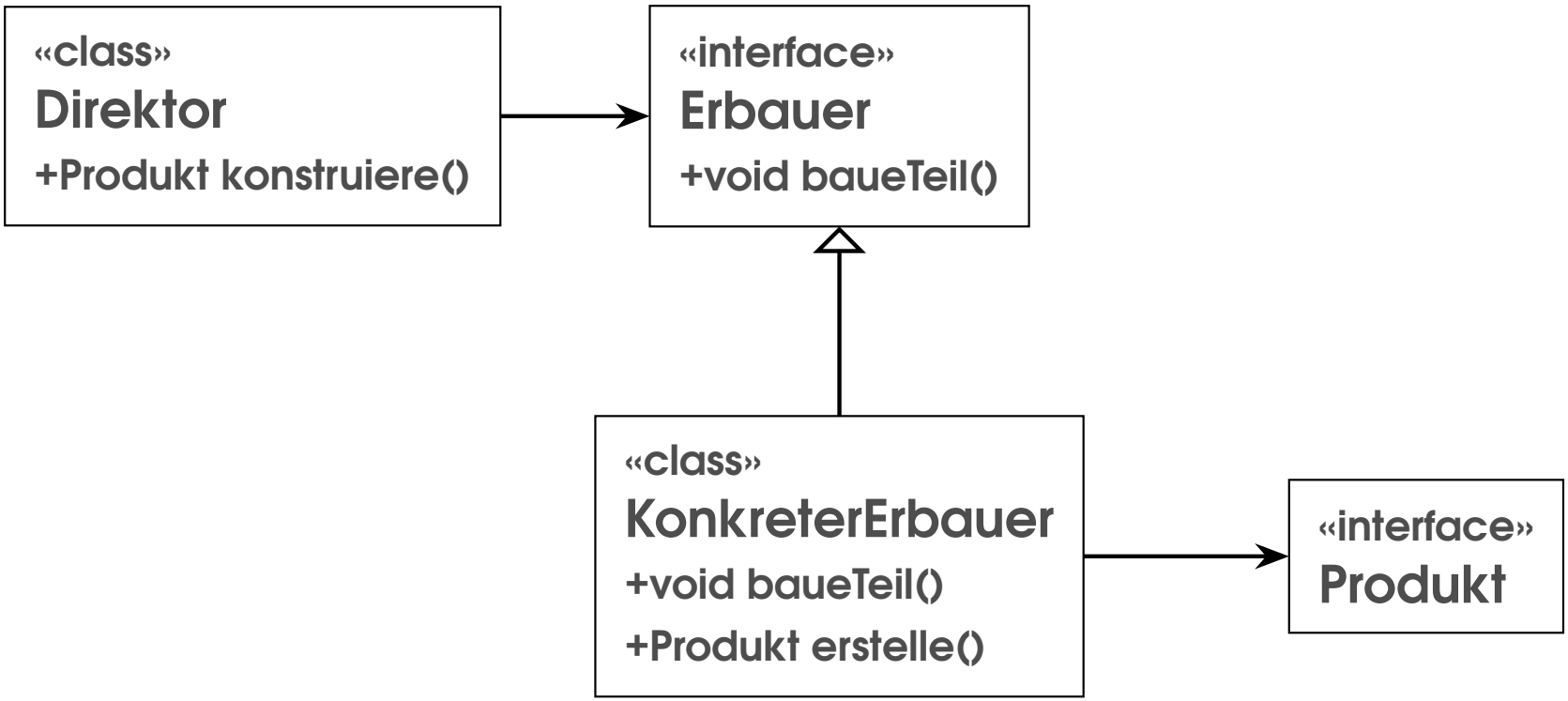
Erbauer – Beispiel



Erbauer – Anwendung,. . .

- wenn der Algorithmus zur Erzeugung komplexer Objekte unabhängig von den Teilen bzw. der Zusammensetzung dieser sein soll
- wenn die Erstellung der Objekte verschiedene Repräsentationen zur Folge hat

Erbauer – Struktur



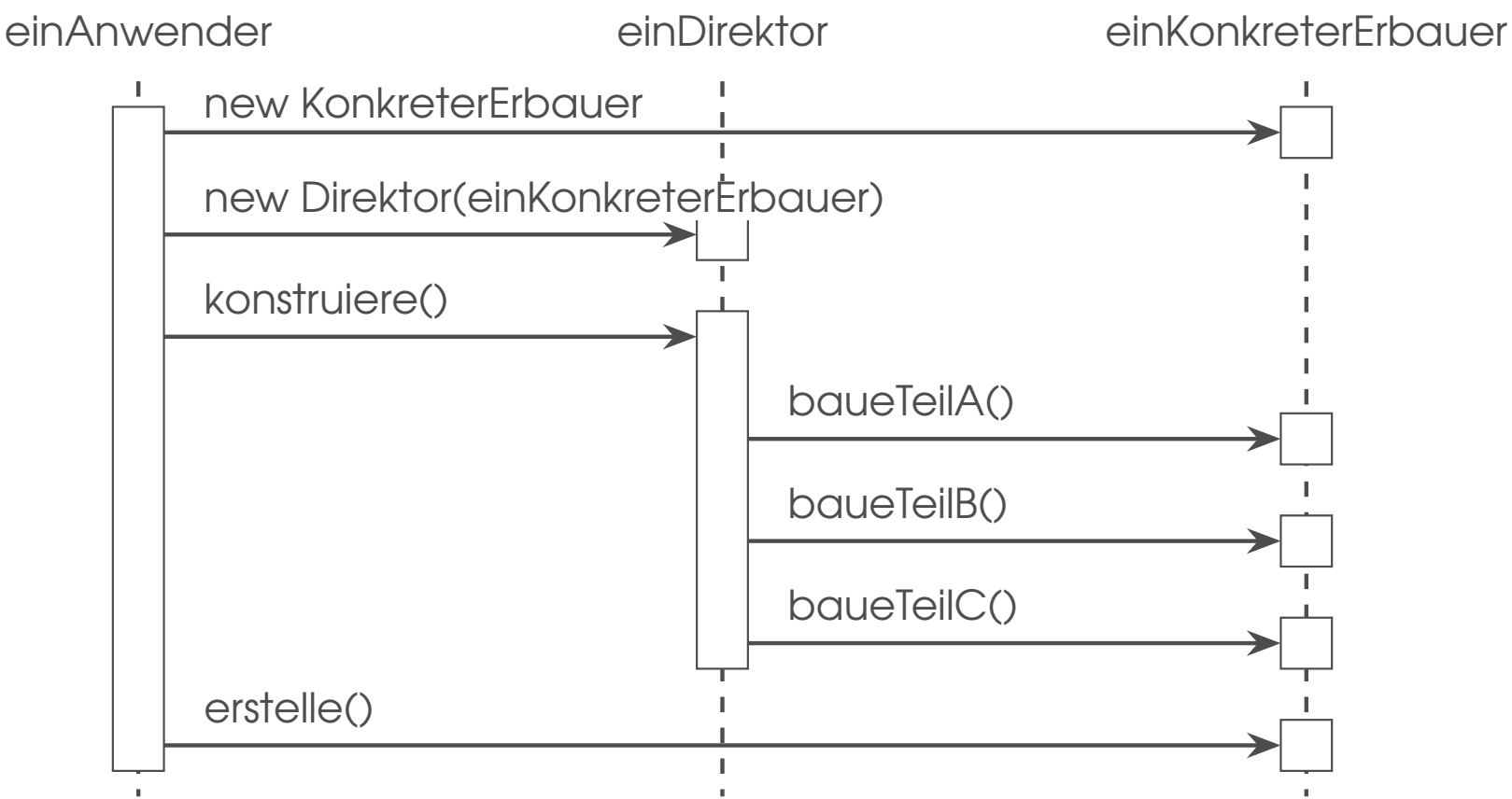
Erbauer – Akteure

- Erbauer
  - Definiert die Schnittstelle zur Erstellung der Teile eines Produkts
- Konkreter Erbauer
  - Erzeugt, konstruiert und setzt verschiedene Teile des Produkts zusammen
  - Implementiert Erbauer Schnittstelle
  - Definiert und verwaltet erstellte Teile
  - Stellt Möglichkeit zur Verfügung das Produkt zu erzeugen

Erbauer – Akteure

- Direktor
  - Konstruiert ein Produkt mit Hilfe des Erbauers
- Produkt
  - Repräsentiert komplex erzeugtes Objekt
  - Konkreter Erbauer erzeugt die interne Repräsentation und definiert den Prozess zum Zusammenfügen der Teile zu einem Ganzen
  - Schließt Klassen mit ein, die die internen Teile beschreiben, enthält Schnittstellen zum Zusammenfügen der Teile

Erbauer – Interaktion der Akteure



Erbauer – Auswirkungen

- Interne Repräsentation des Produkts kann variieren
  - Verstecken der internen Repräsentation und Zusammensetzung
- Genaue Kontrolle über den Konstruktionsprozess
  - Schritt für Schritt Erstellung der Produkte
  - Direktor gibt erst zurück, wenn fertig konstrukiert

Erbauer – Auswirkungen

- Trennung von Code zur Erstellung und Repräsentation
  - Erhöhte Modularität
  - Konkreter Erbauer enthält Code zur Erzeugung einzelner Teile des Produkts
  - Trennung der Verantwortung von Konvertierung bzw. Konstruktion (Direktor) und konkreter Erzeugung (Konkreter Erbauer)

Erbauer – Beispiel

```
public class User {
  private final String firstname;       // required
  private final String lastname;        // required
  private final Role role;              // required
  private final String emailAddress;    // optional
  private final String telephoneNumber; // optional
  private final String roomNumber;      // optional

  public User(String firstname, String lastname, Role role,
    String emailAddress, String telephoneNumber, String roomNumber) {
    super();
    this.firstname = firstname;
    this.lastname = lastname;
    this.role = role;
    this.emailAddress = emailAddress;
    this.telephoneNumber = telephoneNumber;
    this.roomNumber = roomNumber;
  }
}
```

Erbauer – Beispiel

```
public class User {
  private final String firstname;       // required
  private final String lastname;        // required
  private final Role role;              // required
  private final String emailAddress;    // optional
  private final String telephoneNumber; // optional
  private final String roomNumber;      // optional

  public User(String firstname, String lastname, Role role,
    String emailAddress, String telephoneNumber, String roomNumber) {
    super();
    this.firstname = firstname;
    this.lastname = lastname;
    this.role = role;
    this.emailAddress = emailAddress;
    this.telephoneNumber = telephoneNumber;
    this.roomNumber = roomNumber;
  }
}

User teacher = new User("Lars", "Briem", Role.teacher, null, null, null);
```

Erbauer – Beispiel

```
public class User {
  ...
  public User(String firstname, String lastname, Role role,
    String emailAddress, String telephoneNumber, String roomNumber) {
    ...
  }
  public User(String firstname, String lastname, Role role) {
    this(String firstname, String lastname, Role role, null, null, null);
  }
  public User(String firstname, String lastname, Role role, String eMailAddress) {
    this(String firstname, String lastname, Role role, eMailAddress, null, null);
  }
}

User teacher = new User("Lars", "Briem", Role.teacher);
User student = new User("Kurs", "Sprecher", Role.student, "student@dhbw.de");
```

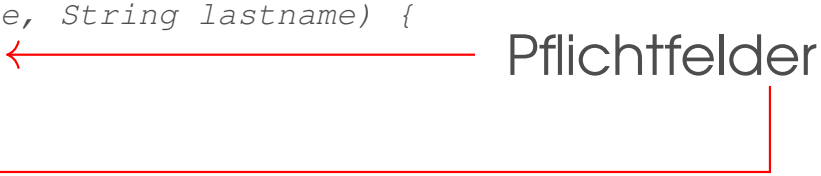
$\Rightarrow$ Beliebige Kombination der Parameter bei gleichem Typ nicht möglich

Erbauer – Beispiel

```
public final class CreateUser {
  private String firstname;
  private String lastname;
  private Role role;
  private String emailAddress;
  private String telephoneNumber;
  private String roomNumber;
  private CreateUser(String firstname, String lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
  public static CreateUser named(String firstname, String lastname) {
    return new CreateUser(firstname, lastname);
  }
  public CreateableUser as(Role role) {
    this.role = role;
    return new CreateableUser();
  }
  public class CreateableUser {
    ...
  }
  private User build() {
    return new User(this.firstname, this.lastname, this.role,
      this.emailAddress, this.telephoneNumber, this.roomNumber);
  }
}
```

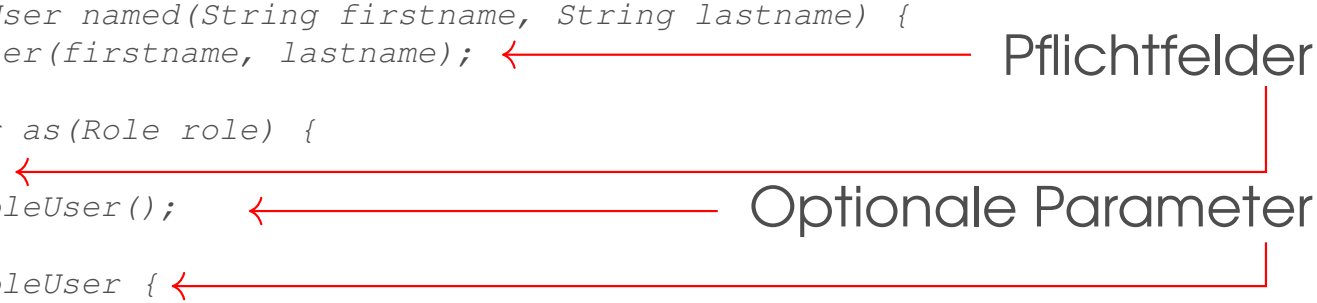
Erbauer – Beispiel

```
public final class CreateUser {
  private String firstname;
  private String lastname;
  private Role role;
  private String emailAddress;
  private String telephoneNumber;
  private String roomNumber;
  private CreateUser(String firstname, String lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
  public static CreateUser named(String firstname, String lastname) {
    return new CreateUser(firstname, lastname);
  }
  public CreateableUser as(Role role) {
    this.role = role;
    return new CreateableUser();
  }
  public class CreateableUser {
    ...
  }
  private User build() {
    return new User(this.firstname, this.lastname, this.role,
      this.emailAddress, this.telephoneNumber, this.roomNumber);
  }
}
```



Erbauer – Beispiel

```
public final class CreateUser {
  private String firstname;
  private String lastname;
  private Role role;
  private String emailAddress;
  private String telephoneNumber;
  private String roomNumber;
  private CreateUser(String firstname, String lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
  public static CreateUser named(String firstname, String lastname) {
    return new CreateUser(firstname, lastname);
  }
  public CreateableUser as(Role role) {
    this.role = role;
    return new CreateableUser();
  }
  public class CreateableUser {
    ...
  }
  private User build() {
    return new User(this.firstname, this.lastname, this.role,
      this.emailAddress, this.telephoneNumber, this.roomNumber);
  }
}
```



Erbauer – Beispiel

```
public final class CreateUser {
        ...
        public class CreateableUser {
                private CreateableUser() {}
                public CreateableUser withTelephoneNumber(String telephoneNumber) {
                        CreateUser.this.telephoneNumber = telephoneNumber;
                        return this;
                }
                public CreateableUser withEmailAddress(String emailAddress) {
                        CreateUser.this.emailAddress = emailAddress;
                        return this;
                }
                public CreateableUser withRoomNumber(String roomNumber) {
                        CreateUser.this.roomNumber = roomNumber;
                        return this;
                }
                public User build() {
                        return CreateUser.this.build();
                }
        }
        ...
}
User teacher = CreateUser.named("Lars", "Briem").as(Role.teacher).build();
User student = CreateUser.named("Kurs", "Sprecher").as(Role.student)
  .withEmailAddress("student@dhbw.de").build();
```

Erbauer – Weitere Hinweise

- Erbauer Schnittstelle so generell wie möglich halten
  - alle konkreten Erbauer abdecken
  - ohne an Details der Erbauer gebunden zu sein
- Kein abstrakte Oberklasse für Produkte
  - Interna der Produkte sind zu unterschiedlich
  - Schablonenmethoden reichen nicht aus

Erbauer – Weitere Hinweise

- Leere Methoden anstelle abstrakter Methoden in Erbauer
  - Konkrete Erbauer müssen nur notwendiges implementieren
- Fungiert als "named Parameter" zur Erzeugung großer Objekte, die nicht selbst in der Hand sind
  - In diesem Fall "Konkreter Erbauer" meist ausreichend

Erbauer – Verwandte Muster

- Abstrakte Fabrik
  - Hauptunterschied zur Fabrik ist die schrittweise Erzeugung
  - Eine Fabrik erzeugt Objekte sofort
- Kompositum
- Ein Kompositum wird häufig durch einen Erbauer erzeugt

Erbauer – Zusammenfassung

- Entkoppelt den Algorithmus zur Erzeugung von Objekten von den Details
- Baut Objekte schrittweise zusammen
- Ermöglicht eine getrennte Wiederverwendung beider Teile
- Vereinfacht die Erweiterbarkeit
  - Unterstützt das Open Closed Principle

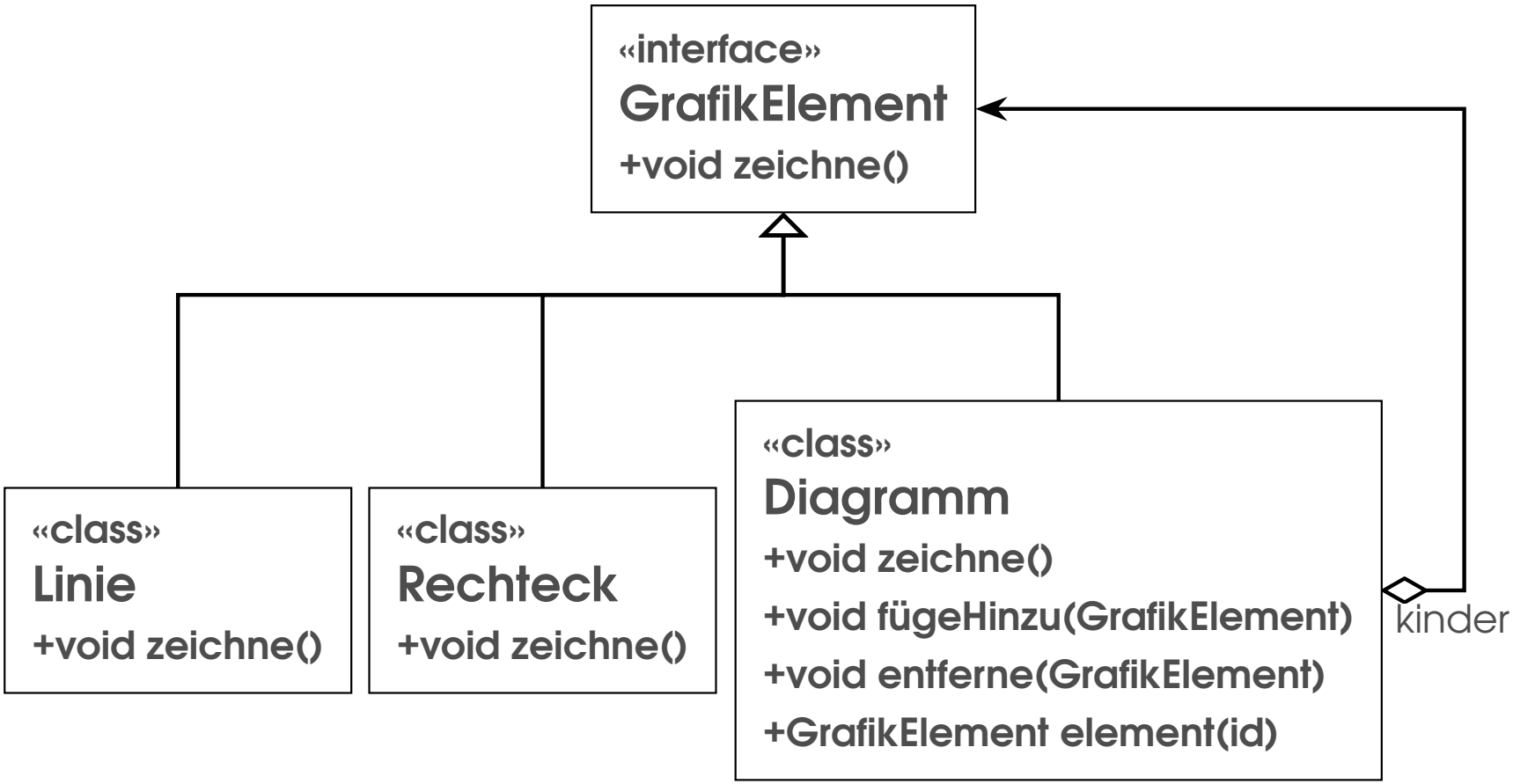
Kompositum

- Setze Objekte zu Baum-Strukturen zusammen, um Teil-Ganzes Hierarchien zu bilden
- Anwender behandeln einzelne Elemente und Komposita gleich
- Klassifikation
  - Objektbasiertes Strukturmuster
  - Datenorientiert
  - Unendliche Rekursion mit Objekten

Kompositum – Motivation

- Kombination einfacher Elemente zur Erzeugung komplexer Strukturen
- Gleichbehandlung von
  - Elementen, die etwas ausführen
  - Containern, die Elemente aufnehmen
- Anwender soll Elemente nicht unterscheiden müssen
  - Implementierungsdetails verbergen

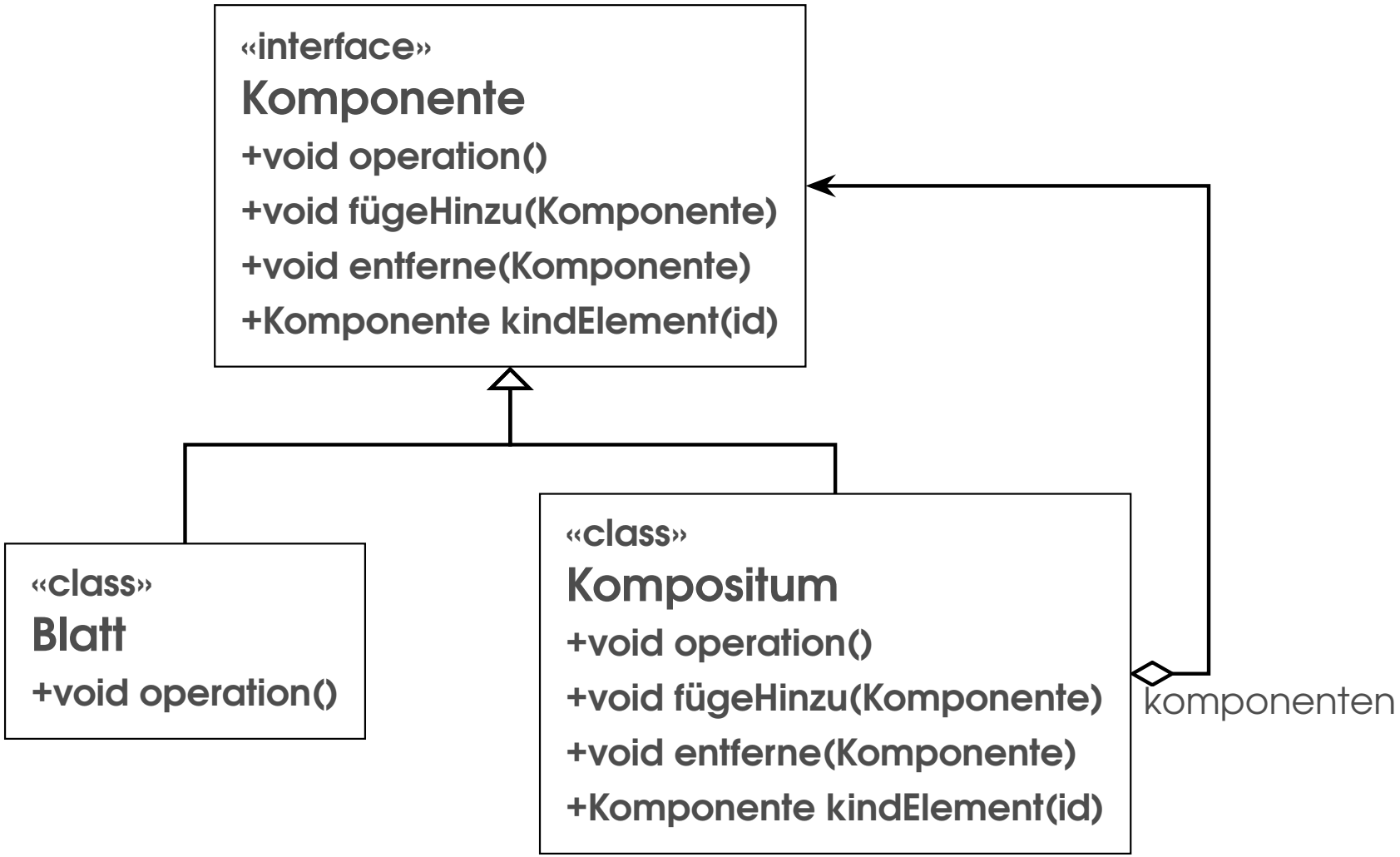
Kompositum – Beispiel



Kompositum – Anwendung

- Teil-Ganzes Hierarchien sollen repräsentiert werden
- Für Anwender soll es egal sein, ob einzelne oder mehrere Elemente vorhanden sind
- Anwender behandeln alle Elemente der Struktur gleich

Kompositum – Struktur



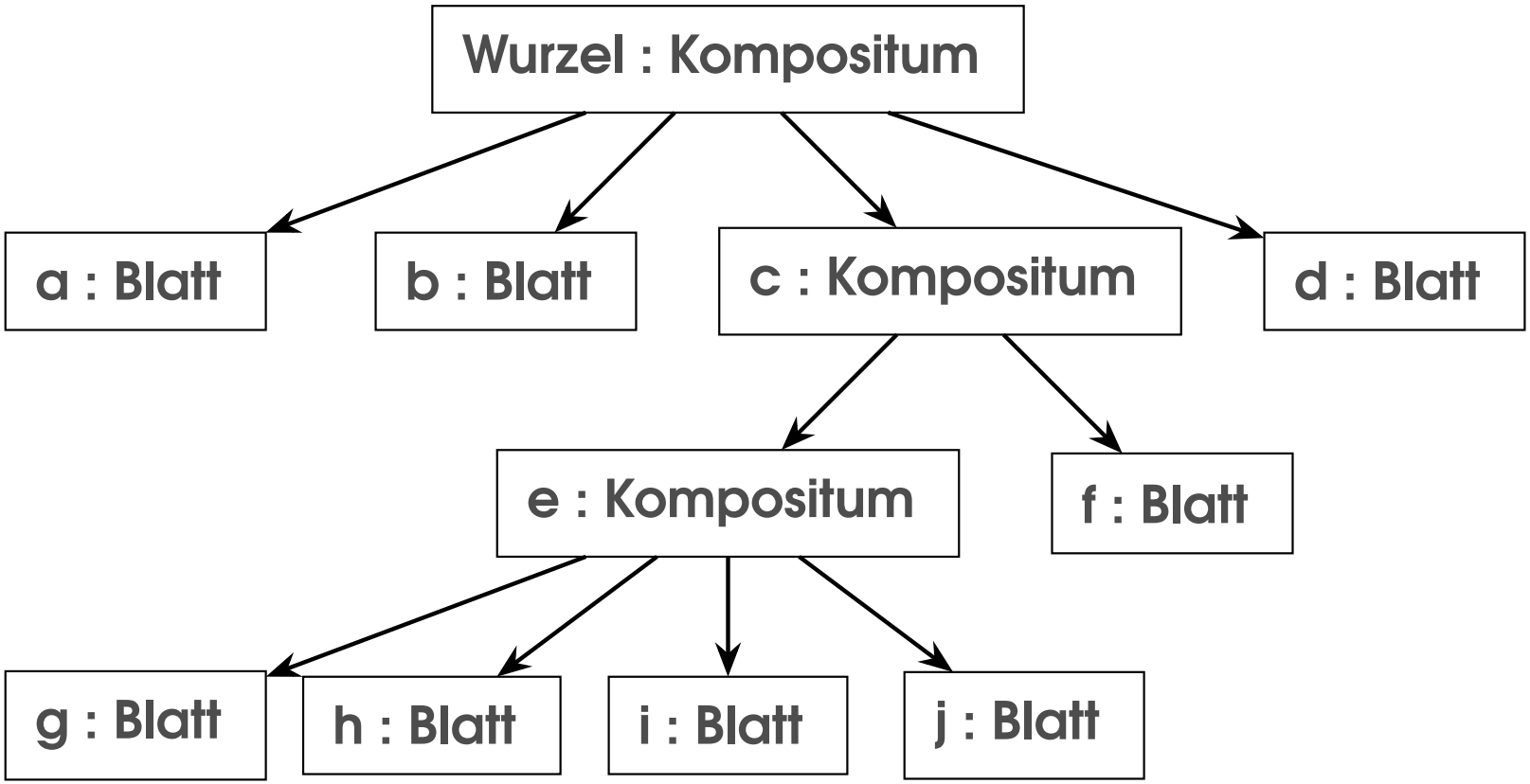
Kompositum – Akteure

- Anwender
  - Manipuliert Objekte im Kompositum nur über Interface
- Komponente
  - Definiert das Interface der Objekte im Kompositum
  - Implementiert Standardverhalten für alle Fälle
  - Definiert Interface zur Verwaltung der Kinder
  - Optional: Definiert und implementiert Schnittstelle für Zugriff auf Eltern, wenn notwendig

Kompositum – Akteure

- Blatt
  - Repräsentiert Blatt-Objekt
  - Hat keine Kinder
  - Definiert Verhalten einfacher Objekte
- Kompositum
  - Definiert Verhalten für Objekte mit Kindern
  - Verwaltet Kinder
  - Implementiert Verhalten bezogen auf Kinder

Kompositum – Interaktion der Akteure



Kompositum – Auswirkungen

- Einfache Elemente können beliebig zusammengebaut werden
  - Rekursive Verschachtelung notwendig
- Vereinfacht die Logik beim Anwender
  - Behandelt alle Objekte gleich
- Neue Komponenten können einfach definiert werden
  - Bestehende arbeiten problemlos mit neuen zusammen

Kompositum – Auswirkungen

- Design zu generell
  - Schwieriger Komponenten einzuschränken
  - Wenn nur bestimmte Elemente erwünscht sind, kann das Typsystem nicht helfen, Überprüfung nur zur Laufzeit

- Referenz zu Eltern kann bei Verarbeitung hilfreich sein
  - Komponente enthält die Logik zur Verwaltung der Elternbeziehung
  - Sorgt für die Einhaltung der Invarianzen

Transparenz vs. Typsicherheit

- Wo soll die Verwaltung der Kinder definiert und implementiert werden?
- Definition in der Wurzel (Komponente)
  - Maximale Transparenz (alle gleich behandeln)
  - Geringere Typsicherheit, weil Anwender sinnlose Aufrufe machen kann
  - Bereitstellung von Standard Add/Remove (Leere Methoden)
    - Anwender erhält keinen Fehler in Blatt, obwohl nichts passiert
    - Anstatt leer, lieber Exception schmeißen bei Add/Remove –> Laufzeitfehler statt Typsicherheit

Transparenz vs. Typsicherheit

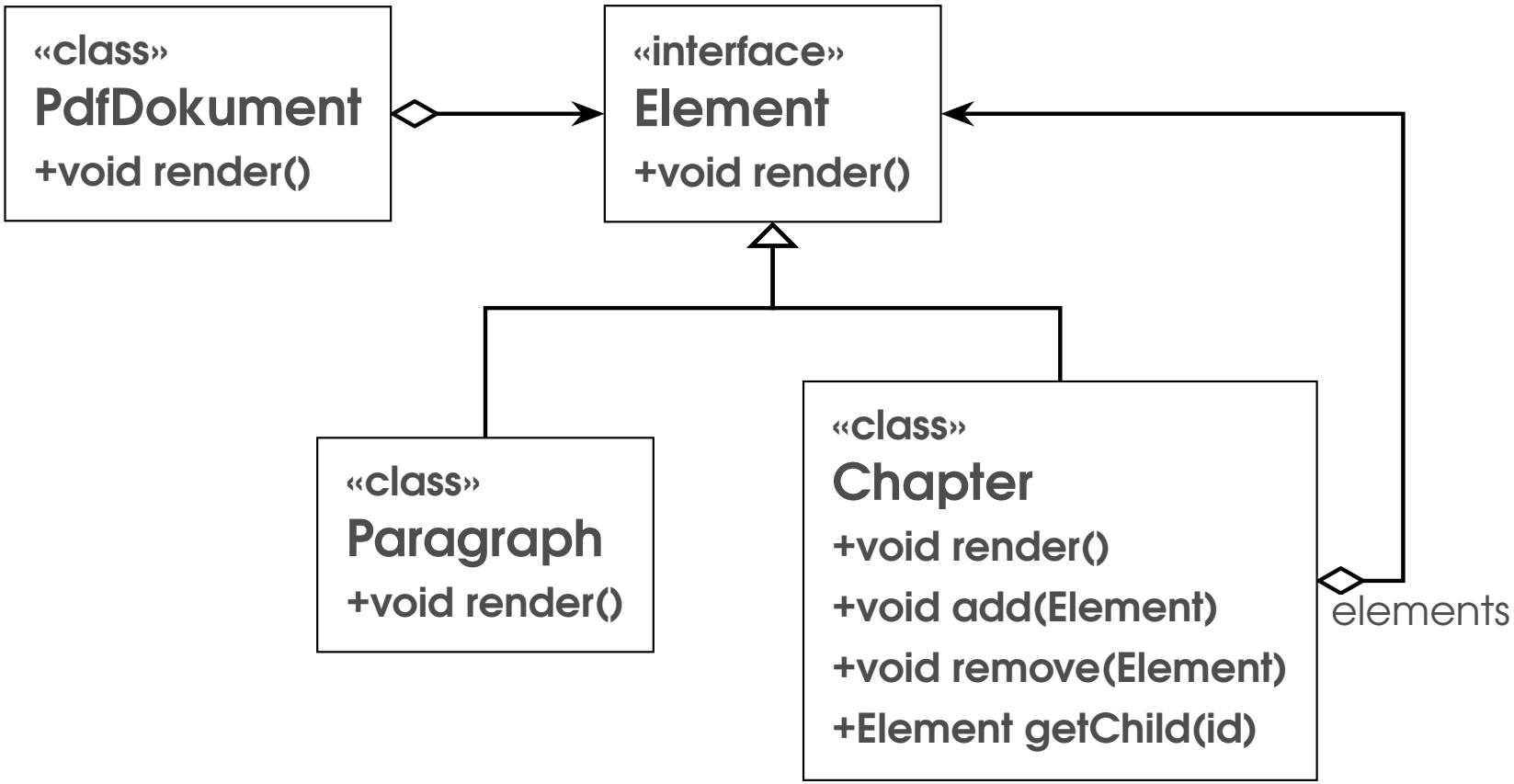
- Definition am Kompositum
  - Maximale Typsicherheit
  - Elemente können nur zu "sinnvollen" Klassen hinzugefügt werden
  - Geringere Transparenz, da unterschiedliche Interfaces
  - – Konvertierung zu Kompositum bei Bedarf notwendig (cast)
  - – Verhalten des Anwenders koppelt sich an Kompositum anstatt Komponente

Transparenz vs. Typsicherheit

- Implementierung von Add/Remove in Komponente inklusive Liste der Kinder
  - Overhead für Blatt Elemente
- Alternative
  - Definition einer "getComposite" Methode
  - Bei Kompositum liefert sie dieses zurück
  - Bei Blatt liefert sie `null`

  $\Rightarrow$ `null`-Überprüfung notwendig

Kompositum – Beispiel - PDF Dokument



Kompositum – Zusammenfassung

- Versteckt einfache und komplexe Objekt-Hierarchien hinter einer Schnittstelle
- Kombiniert einfache Elemente zu komplexen Strukturen
- Anwender kann alle Elemente gleich behandeln

Dekorierer

- Dynamische Zuweisung einer weiteren Verantwortung bzw. Zuständigkeit zu einem Objekt
- Flexible Alternative für Objekthierarchien

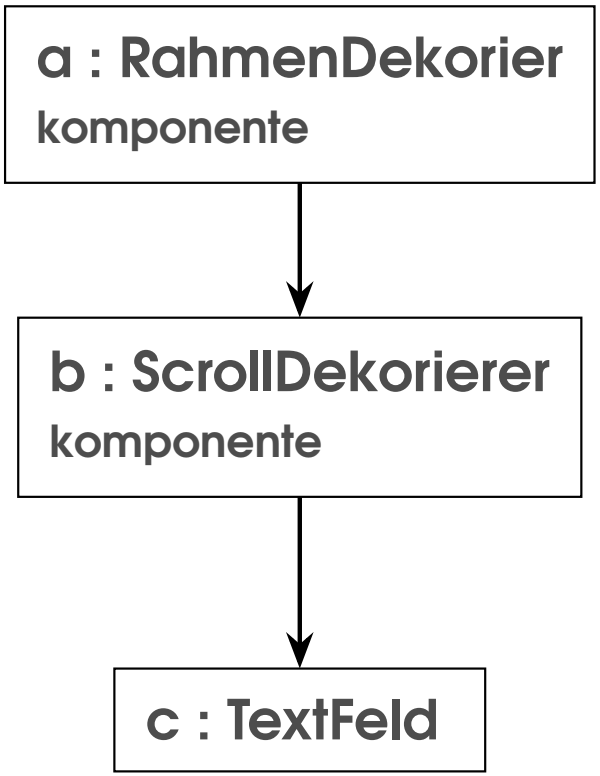
Dekorierer – Einordnung

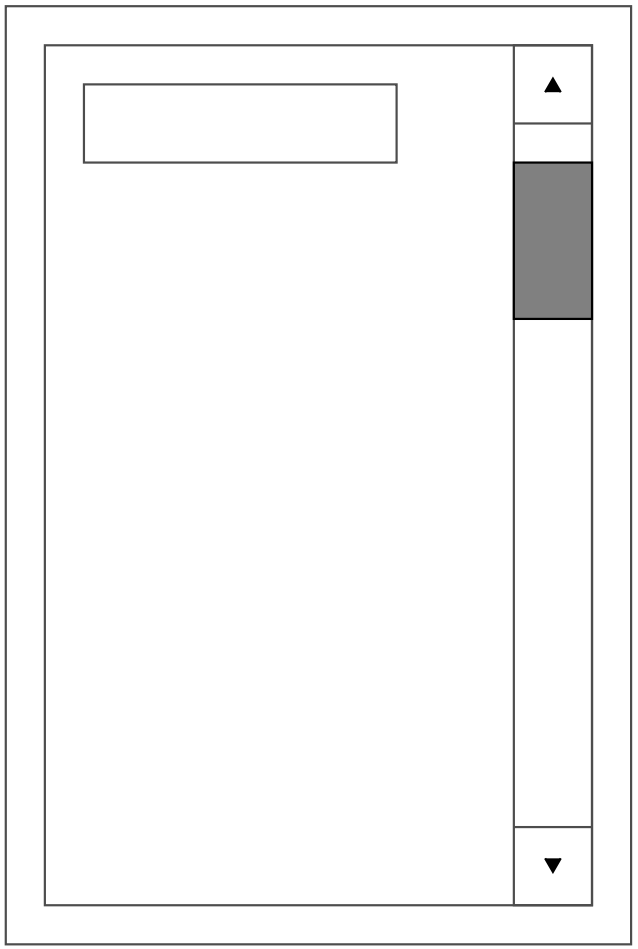
- Objektbasiertes Strukturmuster
- Leichtgewichtig
- Instanzenreich
- Auch bekann als
  - Decorator
  - Wrapper

Dekorierer – Motivation

- Das Hinzufügen von Zuständigkeiten zu einer Klasse mittels Ableitung ist sehr starr
  - Anwender hat keine Entscheidungsgewalt
- Verschachtelung von Objekten zum Hinzufügen von Funktionalität liefert mehr Freiheiten bzw Kombinationsmöglichkeiten
- Zusatzfunktionalität soll transparent dazwischen liegen

Dekorierer – Beispiel

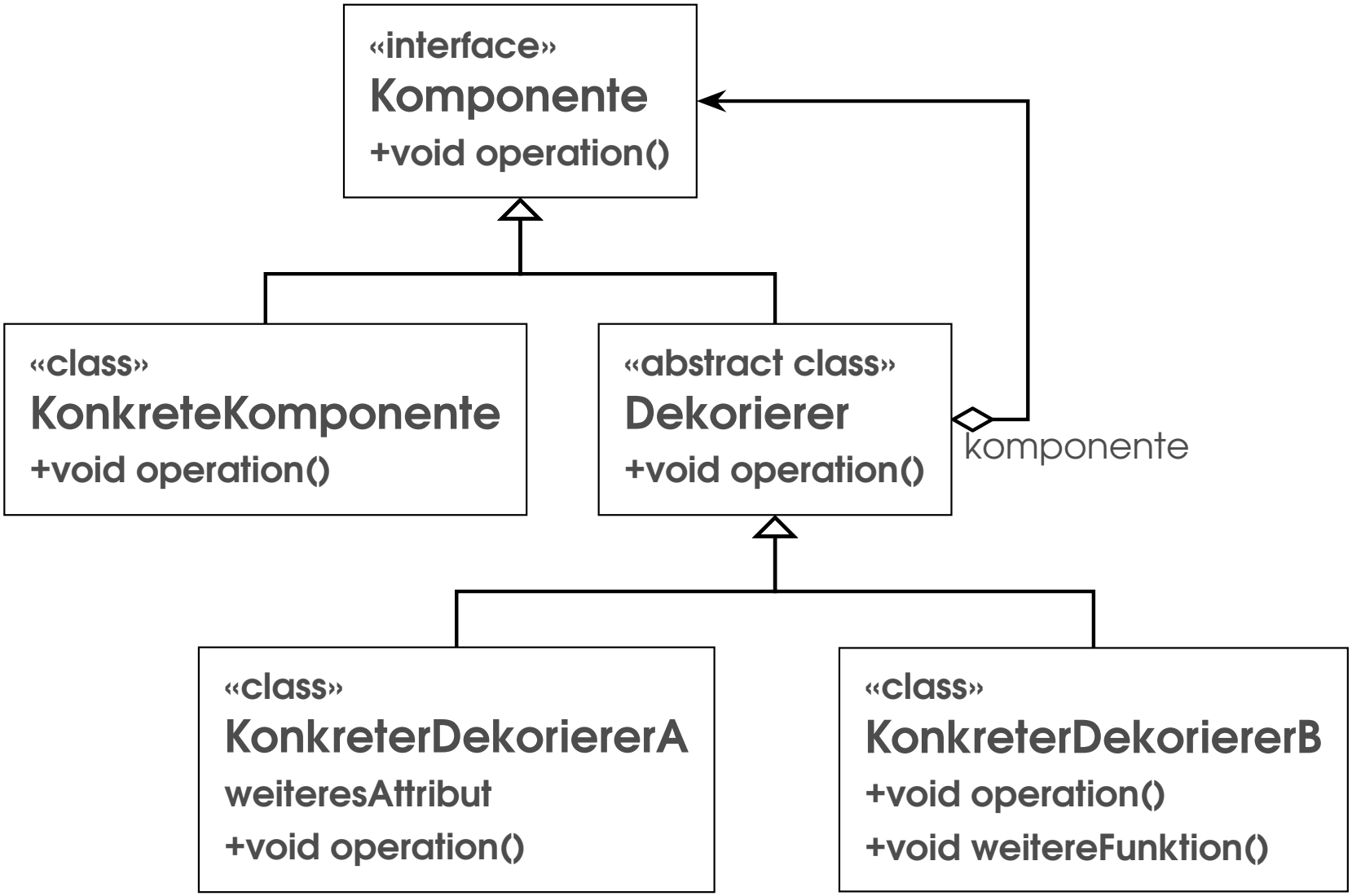




Dekorierer – Anwendung

- Zuweisung von Zuständigkeiten zu einzelnen Objekten dynamisch und transparent, ohne andere zu beeinflussen
- Für entfernbare Zuständigkeiten
- Wenn Ableitungen einer bestehenden Klasse zu komplex ist bzw. die Objekthierarchie extrem aufgebläht wird

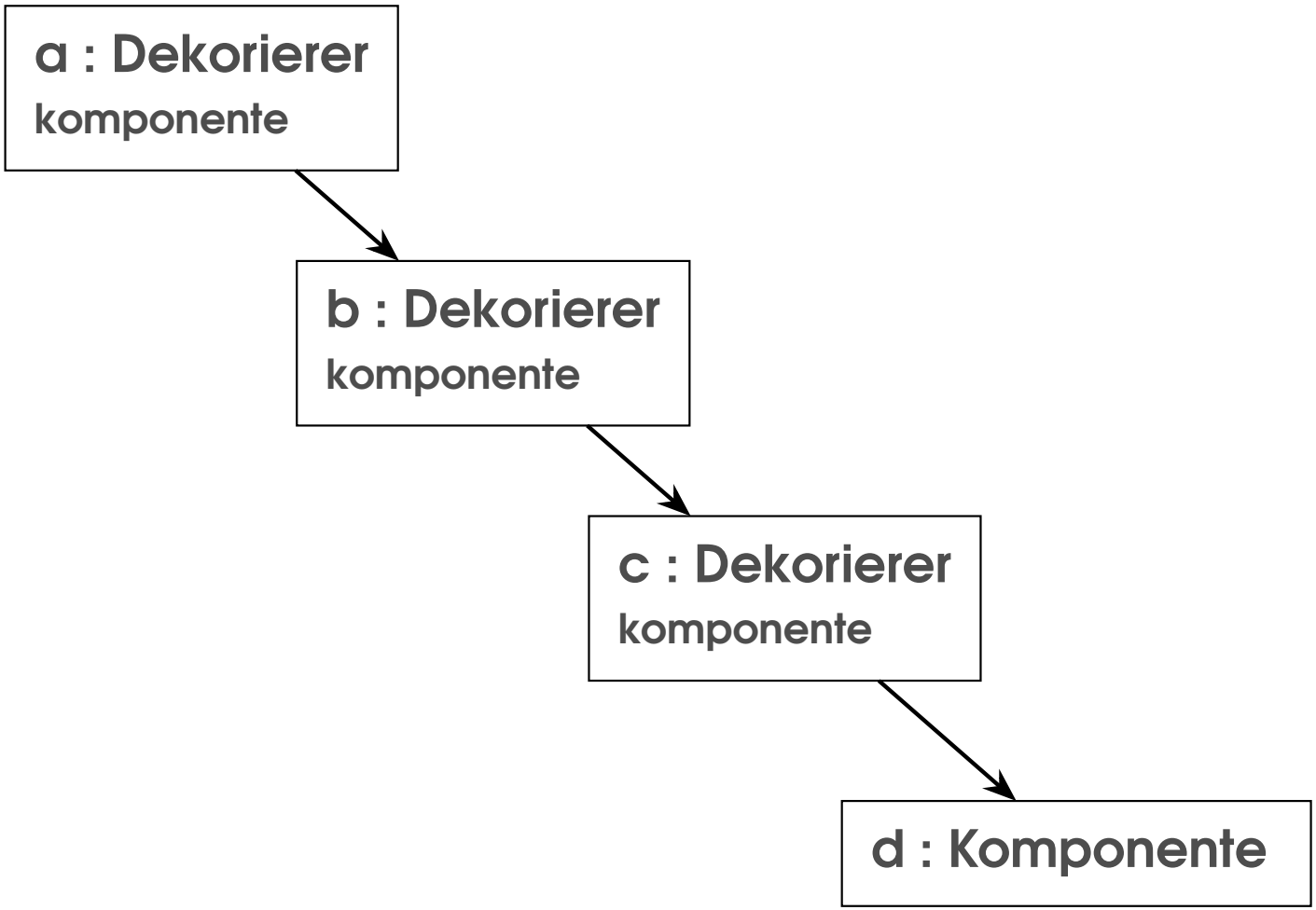
Dekorierer – Struktur



Dekorierer – Akteure

- Komponente
  - Definiert das Interface, das dynamisch erweitert werden soll
- Konkrete Komponente
  - Definiert Komponente, die dynamisch erweitert werden kann
- Dekorierer
  - Hält eine Referenz auf eine Komponente
  - Implementiert das Interface der Komponente
- Konkreter Dekorierer
  - Fügt weitere Zuständigkeit zur Komponente hinzu

Dekorierer – Interaktion der Akteure



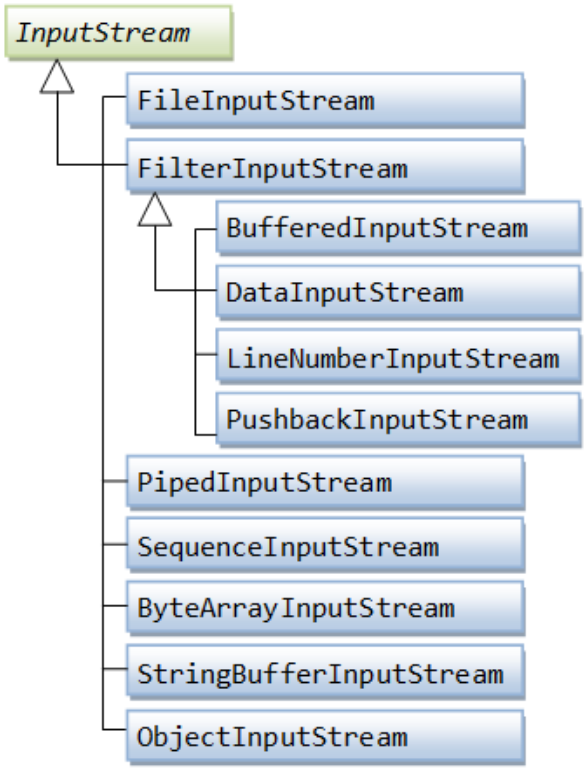
Dekorierer – Auswirkungen

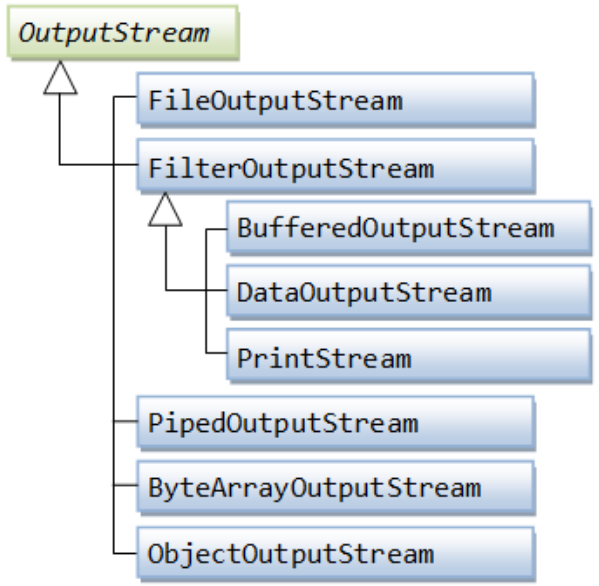
- \+ Flexiblere Alternative zur Ableitung von Objekten
  - \+ Zuständigkeit kann dynamisch hinzugefügt bzw. entfernt werden
  - \+ Beliebige Kombination von Zuständigkeiten, auch mehrfach
- \+ Führt zu einfachen, zusammensteckbaren Klassen
  - \+ Unterstützt das Open Closed Principle
- \+ Vermeidet große konfigurierbare Klassen

Dekorierer – Auswirkungen

- Identität des Dekorierers und der Komponente unterschiedlich
  - `equals` und `hashCode` liefern für Dekorierer und Komponente `false`
- Viele kleine Objekte erschweren das Debuggen bzw. Lernen des Systems für ungeübte

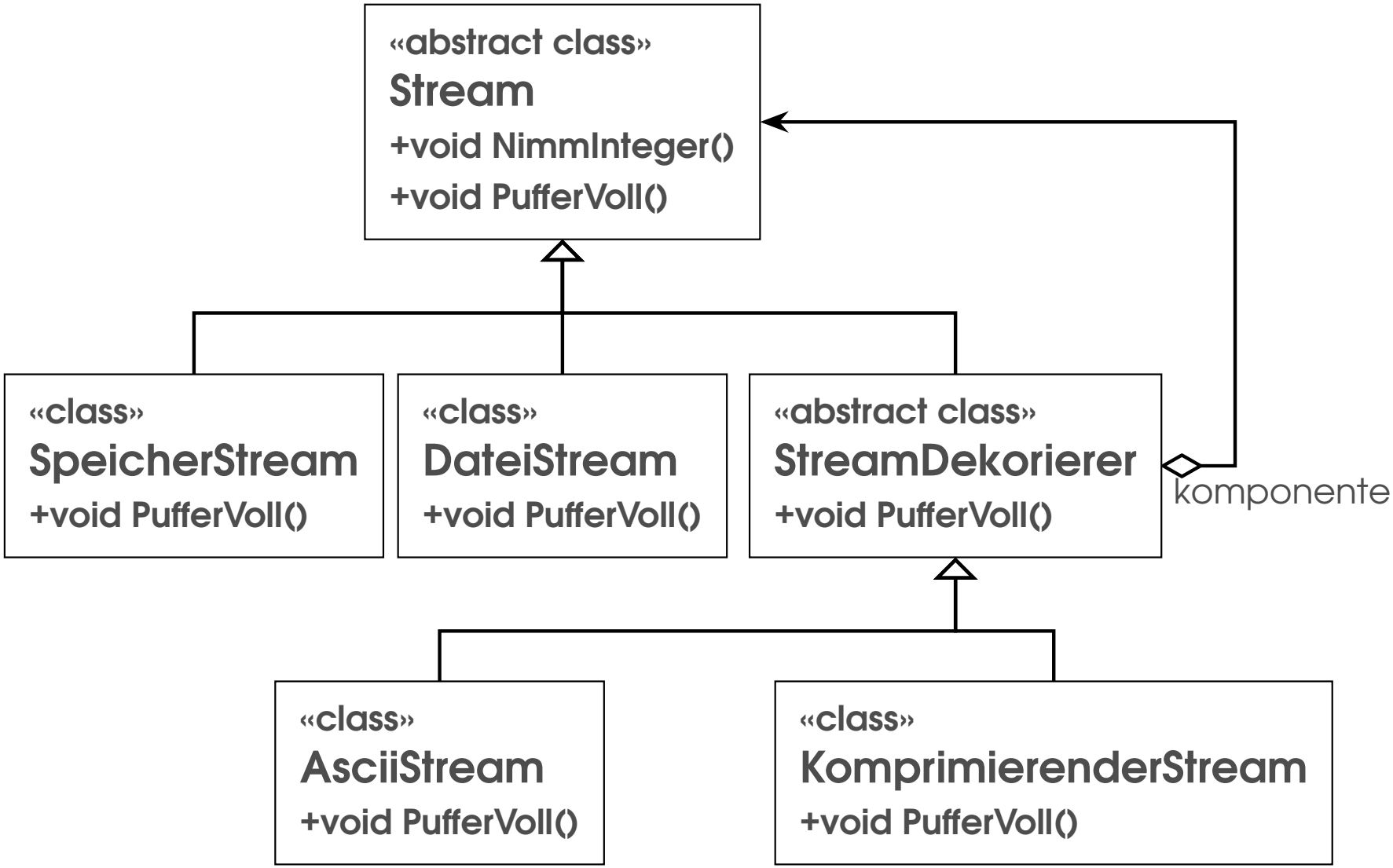
Dekorierer – Beispiel - InputStream






```
InputStream input = new AudioInputStream(format,
                    new DecryptInputStream(secret,
                    new UncompressInputStream(zipType,
                    new BufferedInputStream(
                    new FileInputStream(someFile)
                    ))));
```

Dekorierer – Beispiel - Streamstruktur



Dekorierer – Zusammenfassung

- Erweitern eines Objekts mit zusätzlicher Funktionalität
- Einhaltung einer flachen Objekt-Hierarchie
- Zusätzliche Funktionalität bleibt transparent

Beobachter

- Definiere 1-zu-viele Beziehung zwischen Objekten
- Benachrichtige und Aktualisiere alle abhängigen Objekte automatisch, wenn 1 Objekt den Zustand ändert

Beobachter – Einordnung

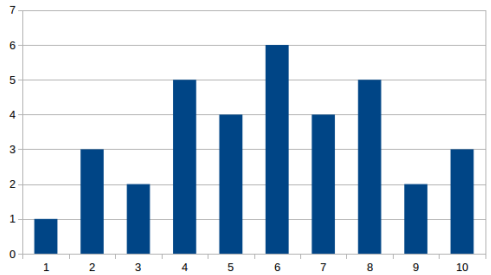
- Objektbasiertes Verhaltensmuster
- Langlebig
- Auch bekannt als
  - Dependents
  - Observer
  - Publish-Subscribe
  - Signal-Slot

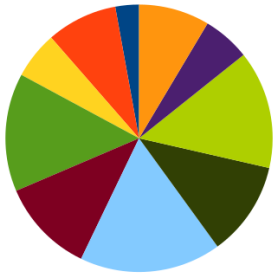
Beobachter – Motivation

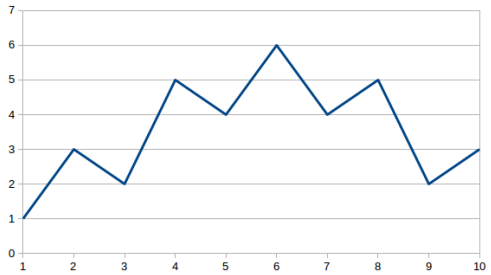
- Sicherstellung bzw. Erhaltung der Konsistenz in modularen Systemen
- Lose Kopplung der Komponenten bei Erhaltung der Konsistenz
- Bei Änderung der Daten sollen alle möglichst schnell über Änderungen informiert werden

Beobachter – Beispiel

- Verschiedene UI für gleiche Daten
  - Liniendiagramm
  - Balkendiagramm
  - Punktwolke
  - ...
- Bei Änderung der Daten sollen sich die Diagramme automatisch aktualisieren



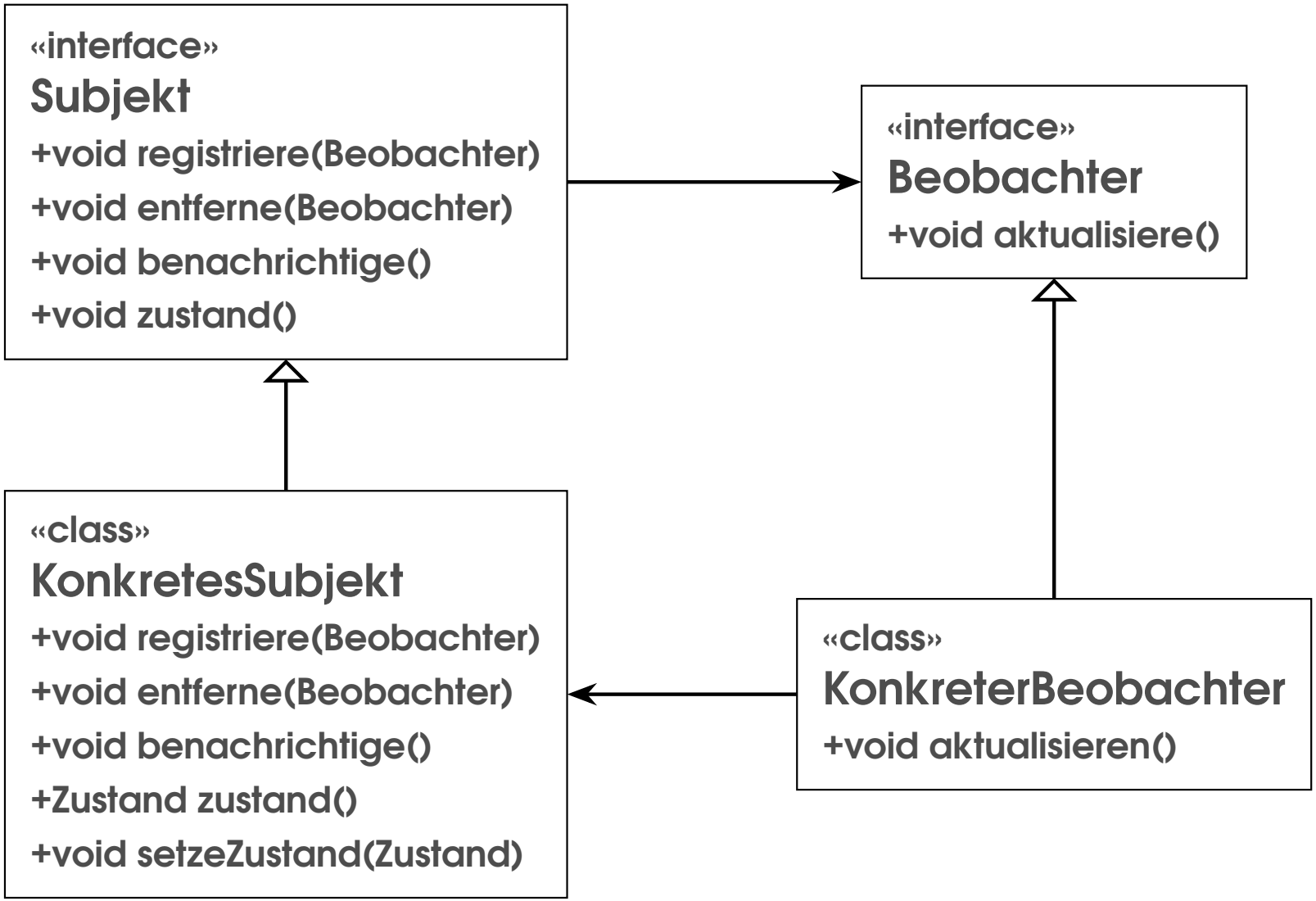




Beobachter – Anwendung

- Wenn die Änderung eines Objekts die Änderung eines anderen Objekts nach sich zieht und nicht bekannt ist, wie viele Objekte sich ändern
- Wenn ein Objekt andere benachrichtigen soll, ohne den konkreten Typ der Objekte zu kennen
  - Führt zu loser Kopplung
- Wenn eine Abstraktion mehrere Aspekte hat, die von einem anderen Aspekt derselben Abstraktion abhängen

Beobachter – Struktur



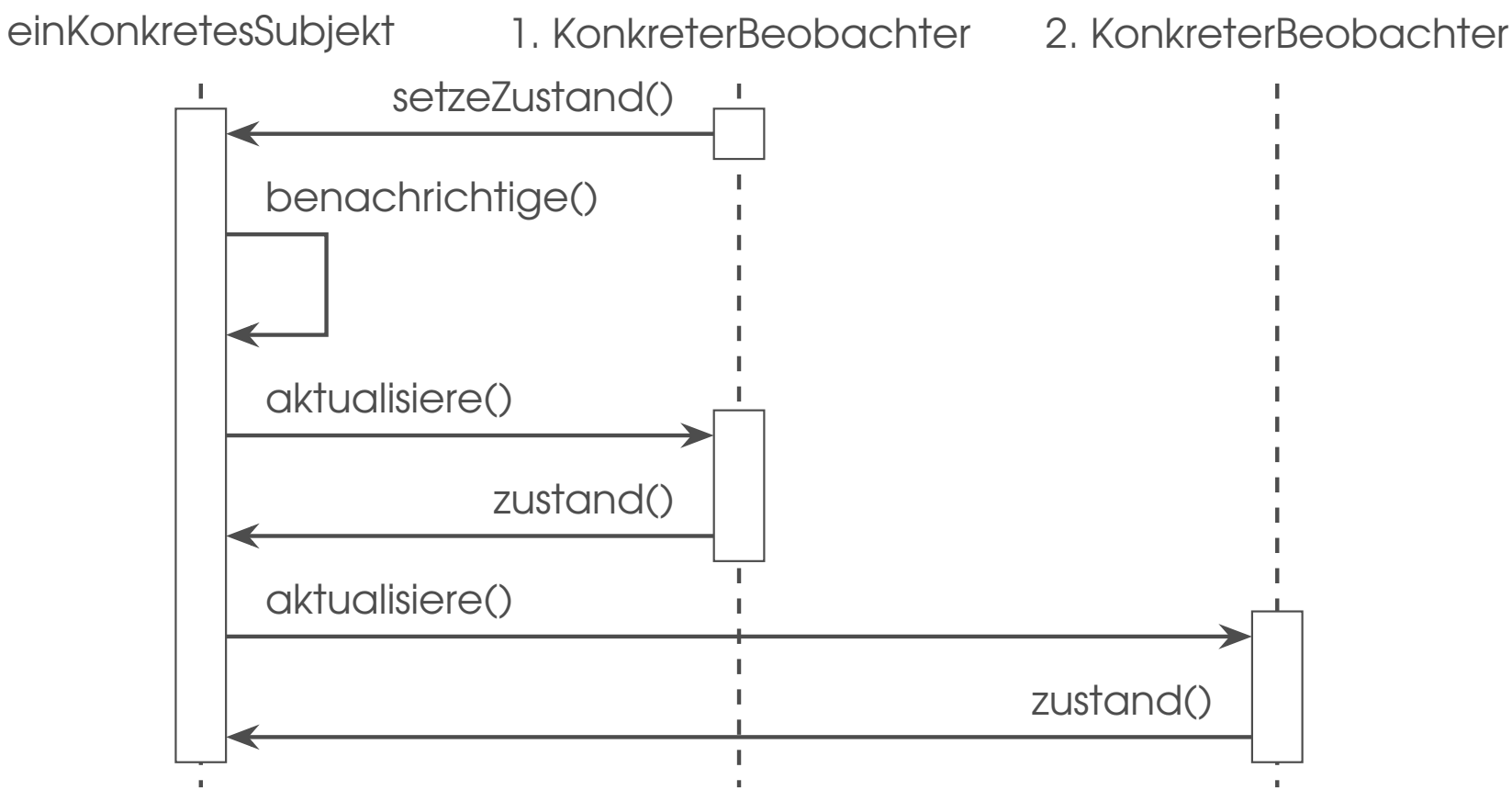
Beobachter – Akteure

- Subjekt
  - Kennt beliebig viele Beobachter
  - Stellt Interface zur Registrierung und Abmeldung von Beobachtern bereit
  - Stellt Interface zum Abrufen des aktuellen Zustands bereit
- Konkretes Subjekt
  - Speichert für Beobachter interessanten Zustand
  - Benachrichtigt Beobachter über Zustandsänderung

Beobachter – Akteure

- Beobachter
  - Definiert Schnittstelle zur Benachrichtigung bzw. Aktualisierung der Objekte
- Konkreter Beobachter
  - Hält Referenz auf konkretes Subjekt
  - Speichert Zustand, der konsistent mit Subjekt sein soll
  - Implementiert Beobachter Interface zur Aktualisierung des Zustands

Beobachter – Interaktion der Akteure



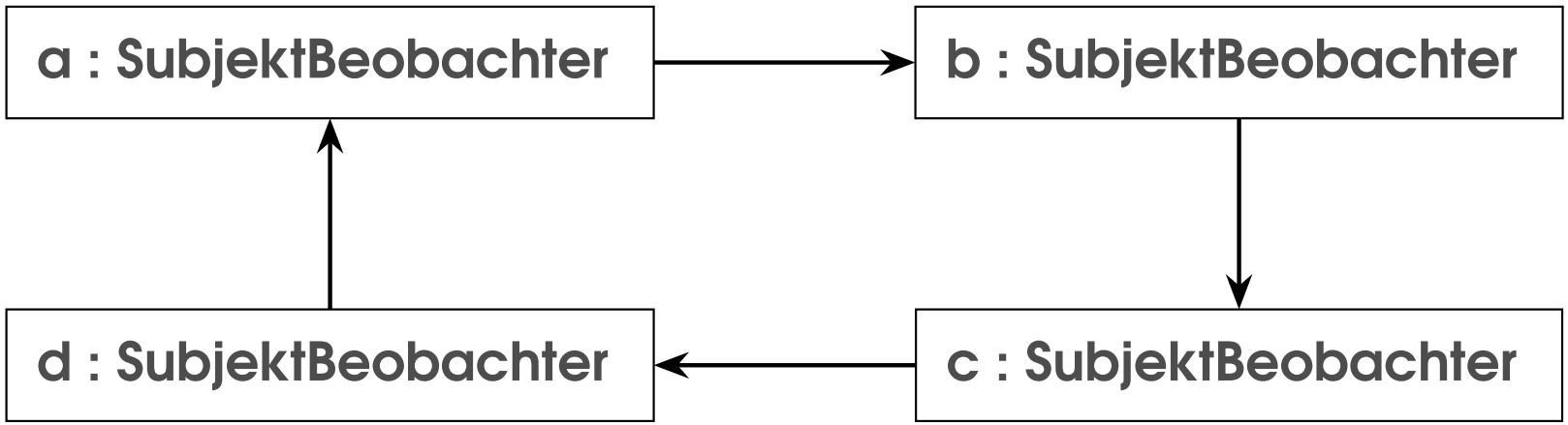
Beobachter – Auswirkungen

- Lediglich eine abstrakte Kopplung zwischen Subjekt und Beobachter über einfaches Interface
  - Subjekt und Beobachter können in unterschiedlichen Schichten liegen
  - Beide getrennt wiederverwendbar
- Automatische Broad-/Multicast Kommunikation an interessierte Objekte
  - Dynamische Menge von Beobachtern
  - Jederzeit änderbar

Beobachter – Auswirkungen

- Unerwartete Aktualisierung
  - Beobachter kennen sich nicht und wissen nicht, was eine Veränderung des Zustands bewirkt
  - Beobachtungszyklen können entstehen

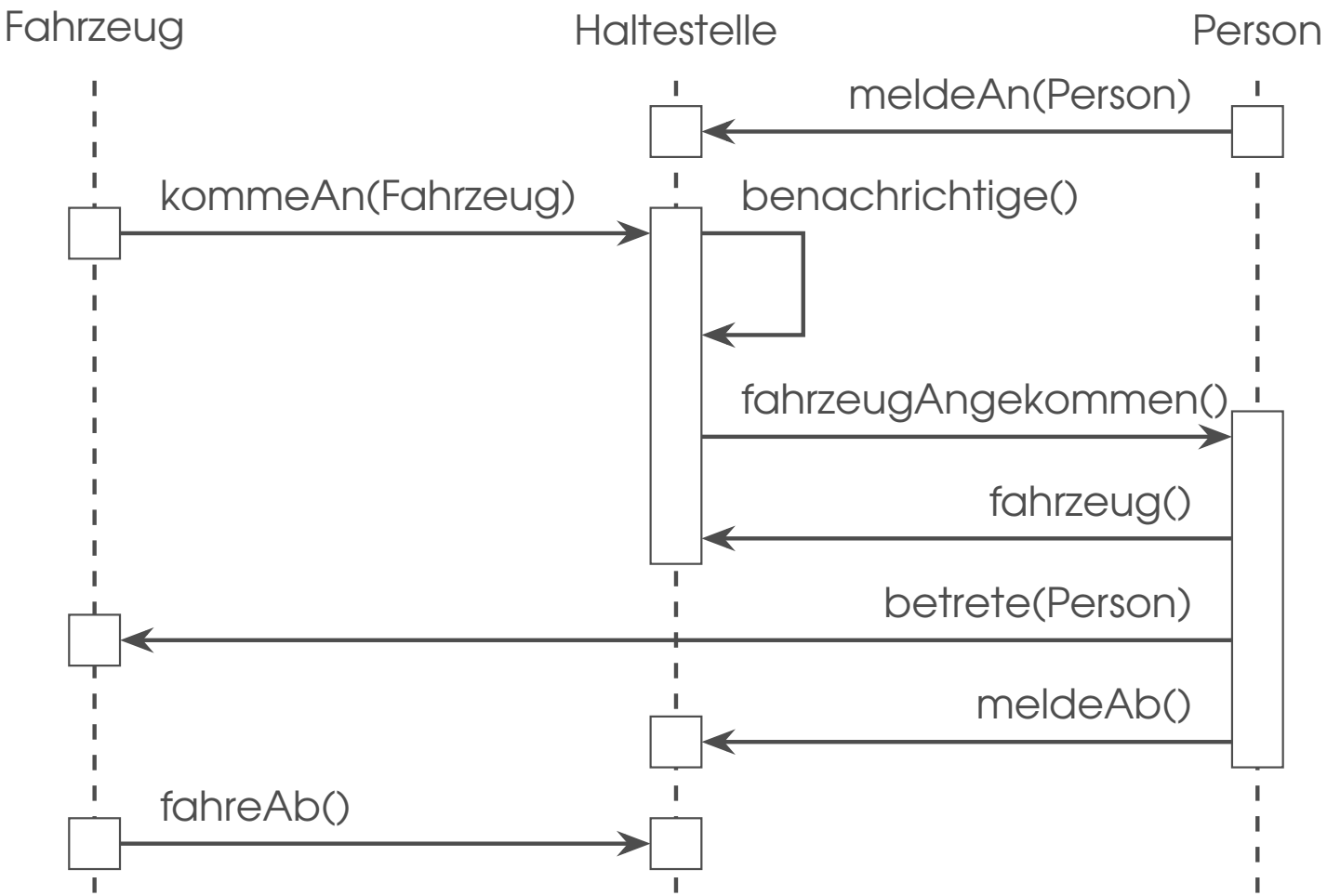
  $\Rightarrow$ Nur echte Aktualisierung weitergeben



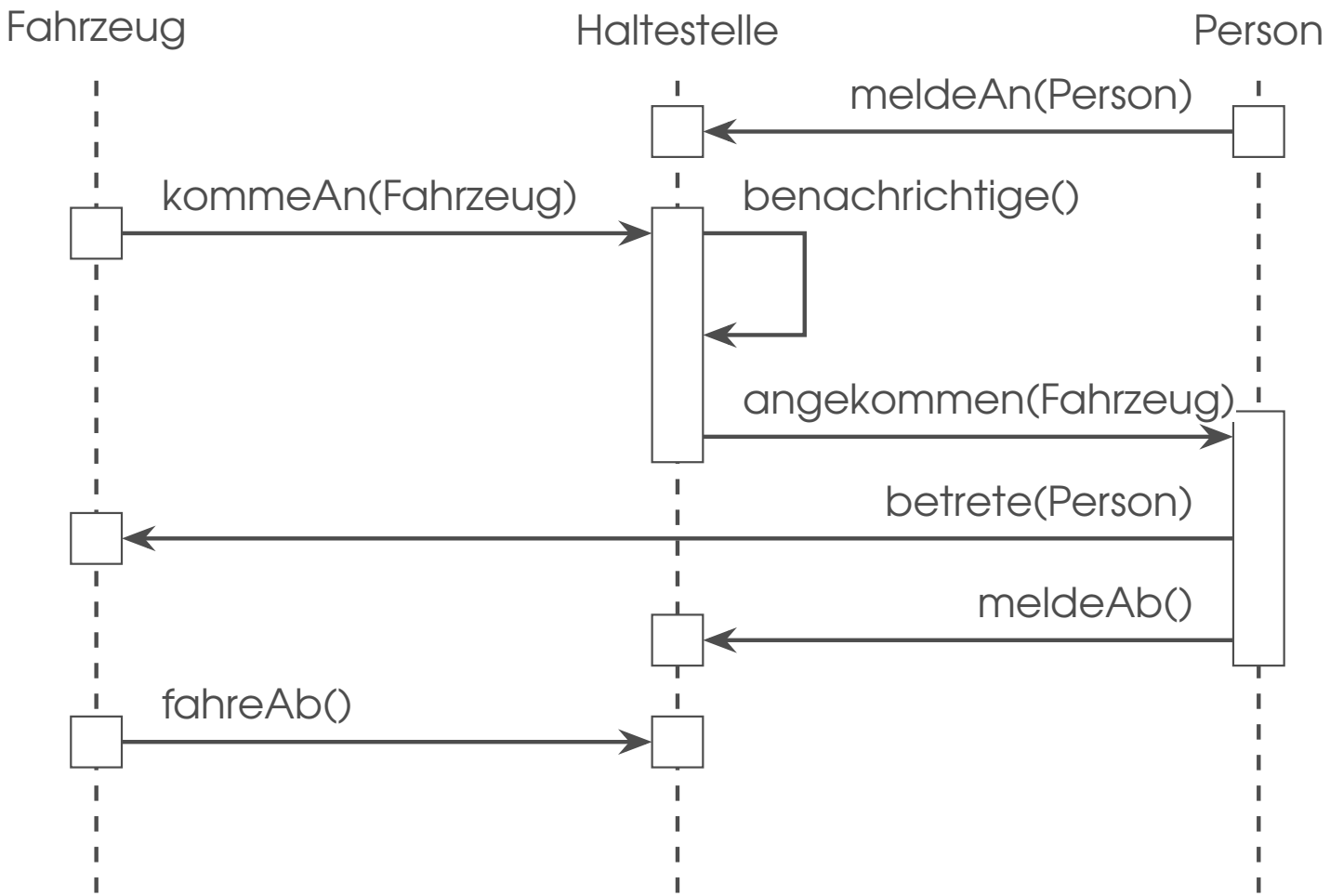
Beobachter – Arten

- Push-Modell
  - Subjekt benachrichtigt Beobachter
  - Inklusive Informationen über Änderung
  - Stärkere Kopplung des Subjekts an Beobachter, da es Annahmen trifft, was Beobachter interessiert
- Pull-Modell
  - Subjekt benachrichtigt Beobachter über Änderung (minimale Nachricht)
  - Beobachter muss sich Informationen selbst holen
  - Beobachter müssen Änderungen selbst herausfinden

Beobachter – Beispiel - Pull



Beobachter – Beispiel - Push



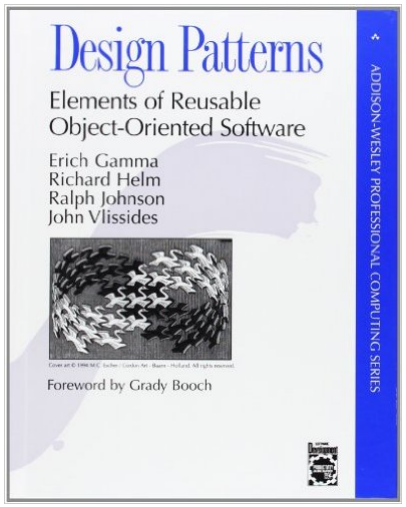
Beobachter – Zusammenfassung

- Automatische Konsistenz von abhängigen Zuständen
- Koppelt die Elemente nur lose aneinander
- Sofortige Benachrichtigung bei Änderung des Zustands
- 2 Arten von Benachrichtigungen

Zusammenfassung

- Liefern Lösungen für wiederkehrende Probleme
- Sind in mehrere Kategorien eingeteilt
- Bauen auf allgemeinen Programmier Prinzipien auf
- Es existieren auch Anti-Entwurfsmuster
  - Big Ball of Mud
  - Spaghetti Code
  - ...

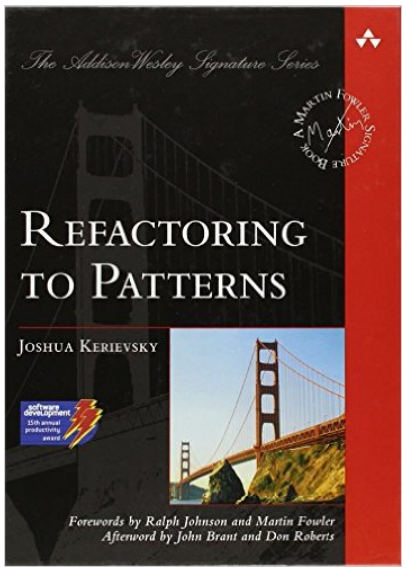
Literatur



- Design Patterns
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Addison-Wesley
  - ISBN: 978-0201633610

Weitere Infos

- Entwurfsmuster auf YouTube
  - John Lindquist erklärt Entwurfsmuster mit StarCraftII
  - `https://www.youtube.com/playlist?list=PL8B19C3040F6381A2`



- Refactoring to Patterns
  - Joshua Kerievsky
  - Addison-Weslay
  - ISBN: 978-0321213358