Design Pattern

- Motivation, Begriff
- Beschreibung
- Beispiel: Observer
- Beispiel: State

Partnerdiskussion

Design Pattern:
zur Erinnerung

- Motivation, Begriff
- Beschreibung
- Observer
- State
- ...
- ...
- …
- ...
- ...
- ...
- ...
- ...

Warum sind Design Pattern (Entwurfsmuster) interessant?

Warum sprechen wir darüber?

Dauer: 2 Minuten

Design Pattern:
zur Erinnerung

- Motivati
  Begriff
- Beschr
- Observer
- State
- ...
- ...
- ...
- ...
- ...
- ...
- ...
- ...

Die Idee:
(positive wie negative) Erfahrungen weitergeben

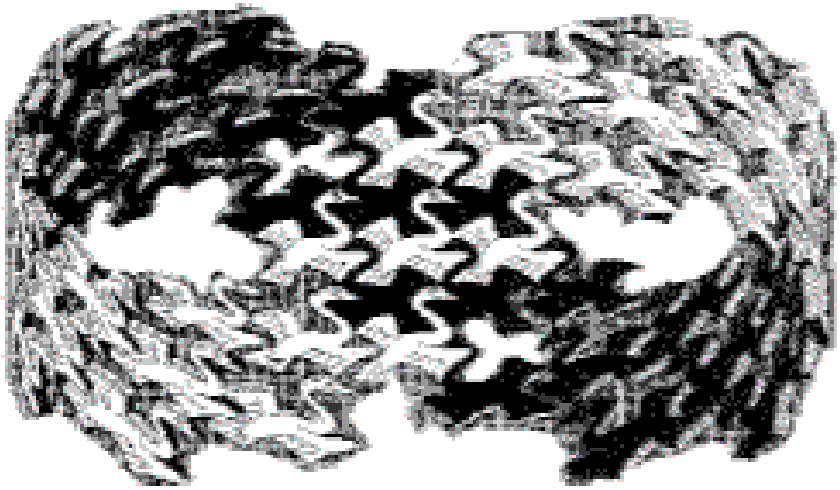
Der Begriff

Design-Pattern (Entwurfsmuster)

- benennen,
- erklären und
- bewerten

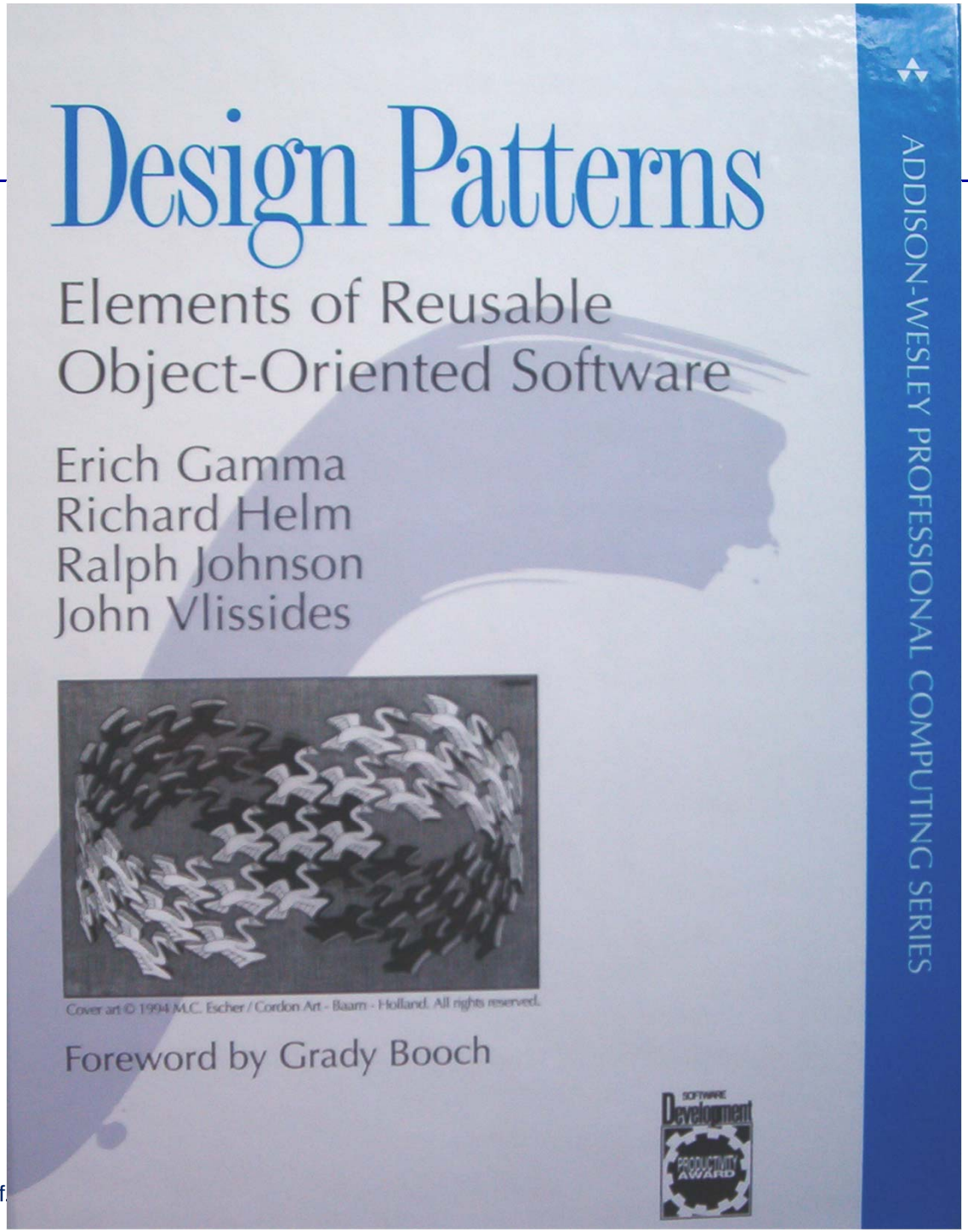
wichtige und wiederholt eingesetzte Design-Elemente objektorientierter Systeme in einer effektiv nutzbaren Form.

Gamma, Helm, Johnson, Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, 1994



Design Pattern:
zur Erinnerung

- Motivation, Begriff
- Beschreibung
- Observer
- State
- ...
- ...
- …
- ...
- ...
- ...
- ...
- ...



Wichtig

Design Pattern:
zur Erinnerung

- Motivation, Begriff
- Beschreibung
- Observer
- State
- ...
- ...
- …
- ...
- ...
- ...
- ...
- ...

Es werden *keine neuen* Design-Konstruktionen *erfunden*, *sondern existierende* (und von erfahrenen Designern eingesetzte) Lösungen für typische Probleme mit ihren Vor- und Nachteilen *dokumentiert*

Frage

Design Pattern:
zur Erinnerung

- Motivation, Begriff
- Beschreibung
- Observer
- State
- ...
- ...
- …
- ...
- ...
- ...
- ...
- ...

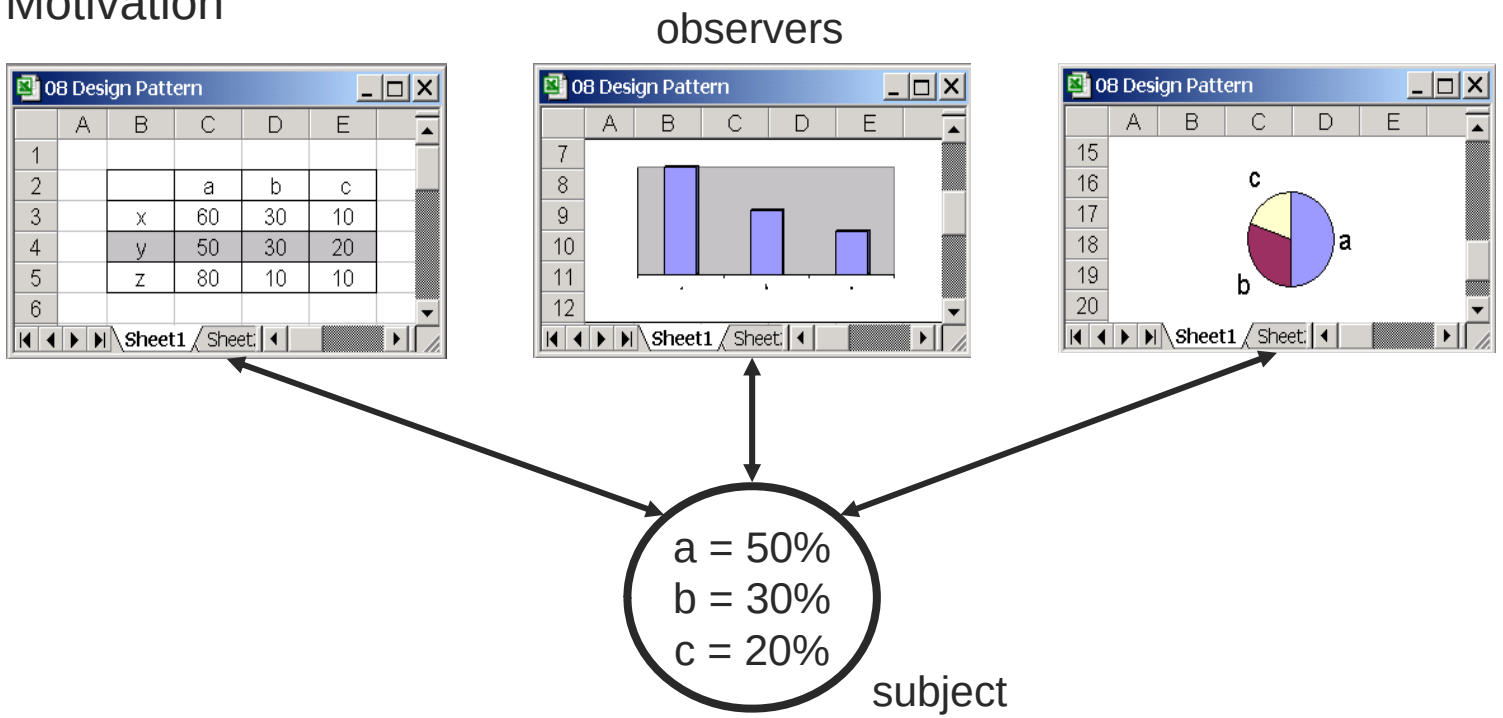
Welche Elemente gehören zur Beschreibung eines Design Patterns?
Was müssen Sie wissen, um ein Pattern verwenden zu können?

(Mindest-)Dokumentation eines Patterns

1. Name
   - Identifikation
   - Teil des "Design-Vokabulars"
2. Problembeschreibung
   - Kontext für die Pattern-Anwendung
   - Beispiele
     - → Konkrete Algorithmen und ihre Repräsentation mittels Objekten
     - → Vorbedingungen für den Einsatz
3. Lösungsbeschreibung
   - Klassen, Beziehungen, Verantwortlichkeiten, Zusammenarbeit
   - Normalerweise kein konkretes Design!
4. Konsequenzen
   - Beispiele
     - → Laufzeit- oder Speicherplatzbedarf
     - → Anzahl / Größe von Klassen / Objekten

Beispiel für ein *behavioral* Pattern: Observer

- Aufgabe
  Definition einer 1 : n – Beziehung zwischen Objekten, so dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des Objekts ändert, von dem sie abhängen
- Motivation

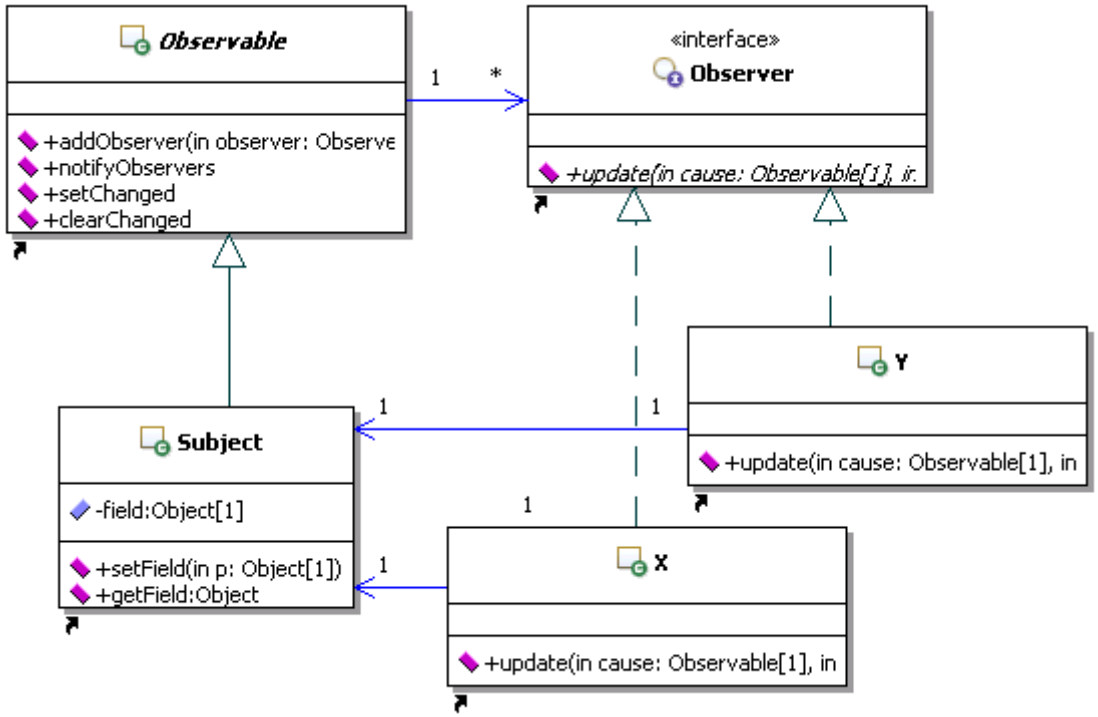


Beispiel für ein *behavioral* Pattern: Observer

- Motivation (Fortsetzung)
  - Daten und die verschiedenen Sichten darauf müssen konsistent gehalten werden
  - Ändern sich die Daten, so müssen die Tabelle und die Graphiken aktualisiert werden
- Anwendbarkeit
  - Wenn ein Modell zwei Aspekte hat, von denen der eine vom anderen abhängt; das Pattern ermöglicht es, diese Aspekte getrennt voneinander zu variieren und zu benutzen
  - Wenn eine Datenänderung eines Objekts Änderungen in anderen Objekten mit unbekannter Zahl nach sich zieht
  - Wenn ein Objekt andere Objekte benachrichtigen muss, ohne diese anderen Objekte zu kennen (geringe Kopplung)

Beispiel für ein *behavioral* Pattern: Observer

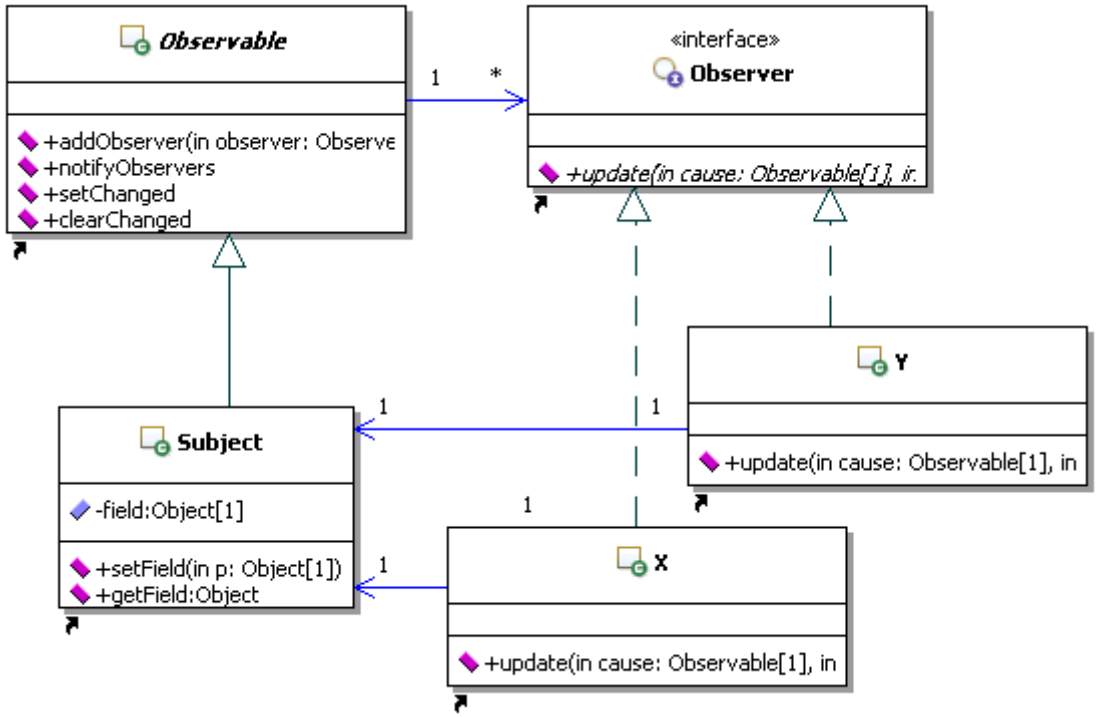
- Struktur 1/2



- Beteiligte
  - Observable
    - → Kennt seine Observer; eine beliebige Anzahl ist möglich
    - → Erlaubt es, Observer-Objekte anzuhängen (*addObserver*) und abzuhängen (*deleteObserver*)
  - Observer
    - → Definiert eine Schnittstelle für Objekte, die über Änderungen informiert werden sollen (*update*)

Beispiel für ein *behavioral* Pattern: Observer

- Struktur 2/2



- Beteiligte
  - Subject
    - → Speichert den Zustand, der für konkrete *Observer*-Objekte (*X*, *Y*) relevant ist
    - → Sendet eine Benachrichtigung (*notifyObservers*), wenn sich sein Zustand ändert
  - Konkrete Observer (*X*, *Y*)
    - → Enthält eine Referenz auf ein konkretes *Subject*-Objekt, um dessen Zustand abfragen zu können
    - → Implementiert die *update*-Methode des *Observers*

Beispiel für ein *behavioral* Pattern: Observer

- Zusammenarbeit der beteiligten Objekte

Beispiel für ein *behavioral* Pattern: Observer

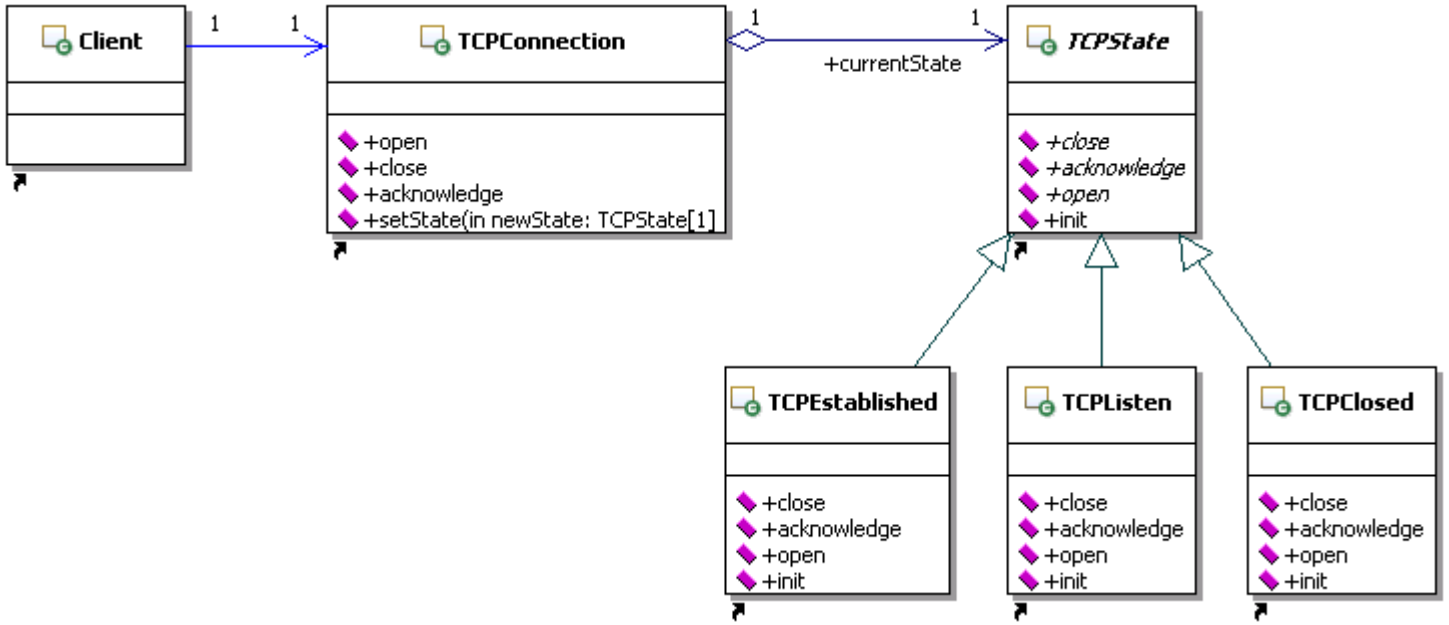
- Konsequenzen
  - Lose Kopplung zwischen *Subject* und *Observer*: Das *Subject* kennt nur die *update*-Methode zu seinen *Observer*n
  - Broadcast-Kommunikation: Die *update*-Information wird an alle Objekte weitergeleitet, die sich zuvor registriert haben; es werden keine Objekte konkret angesprochen (jederzeit Hinzufügen/Wegnehmen möglich)
  - Update-Kaskaden:
    - → Änderungen im *Subject* können zu Kaskaden von *Observer*-Updates führen, kein (einzelnes) Objekt hat die Kontrolle
    - → Die *update*-Information ist relativ unspezifisch

Beispiel für ein *behavioral* Pattern: State

- Aufgabe
  - Erlaubt einem Objekt, sein Verhalten zu ändern, wenn sich sein (interner) Zustand ändert
  - Das Objekt ändert scheinbar seine Klasse (weil es sich in verschiedenen Situationen/Zuständen so unterschiedlich verhält)

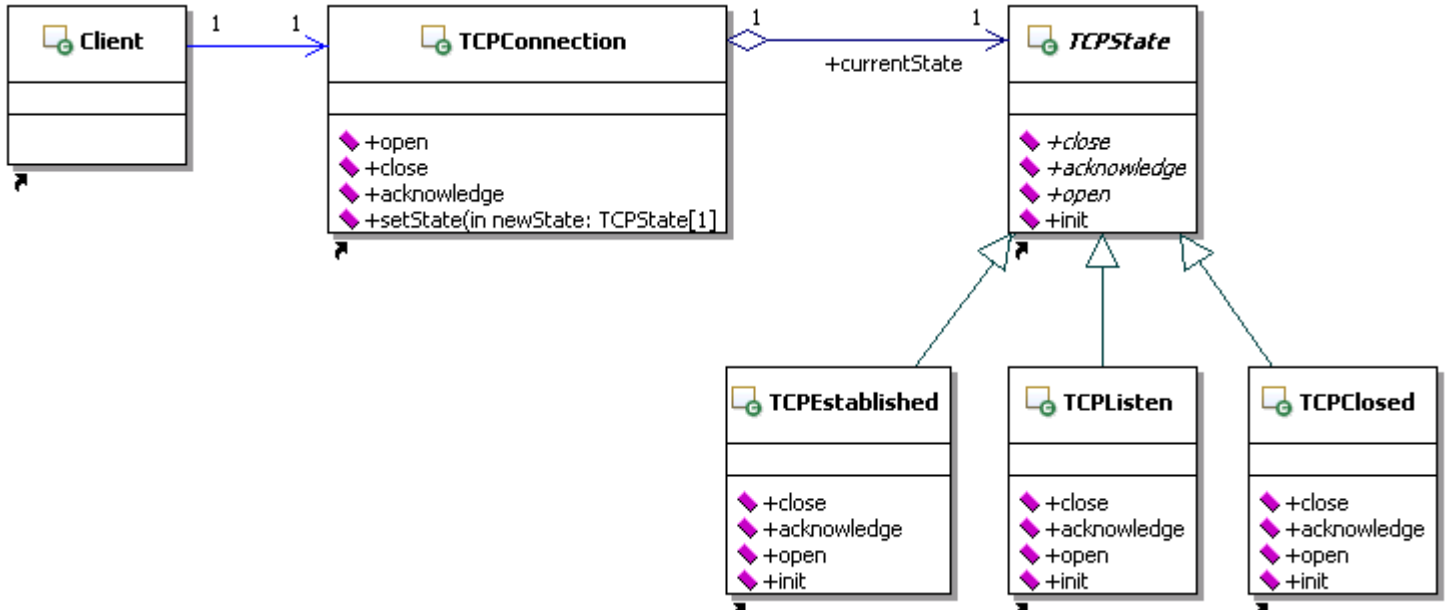
Beispiel für ein *behavioral* Pattern: State

- Motivation
  - Eine TCP-Verbindung kann in 3 Zuständen sein: Established, Listening, Closed
  - In Abhängigkeit vom aktuellen Zustand reagiert die Verbindung unterschiedlich auf gleiche Anfragen, z.B. Open
  - Idee
    - → Eine abstrakte Klasse (oder ein Interface) definiert identische Schnittstellen (und Default-Verhalten) für alle Zustände
    - → Eine konkrete Klasse implementiert das Verhalten in einem konkreten Zustand



Beispiel für ein *behavioral* Pattern: State

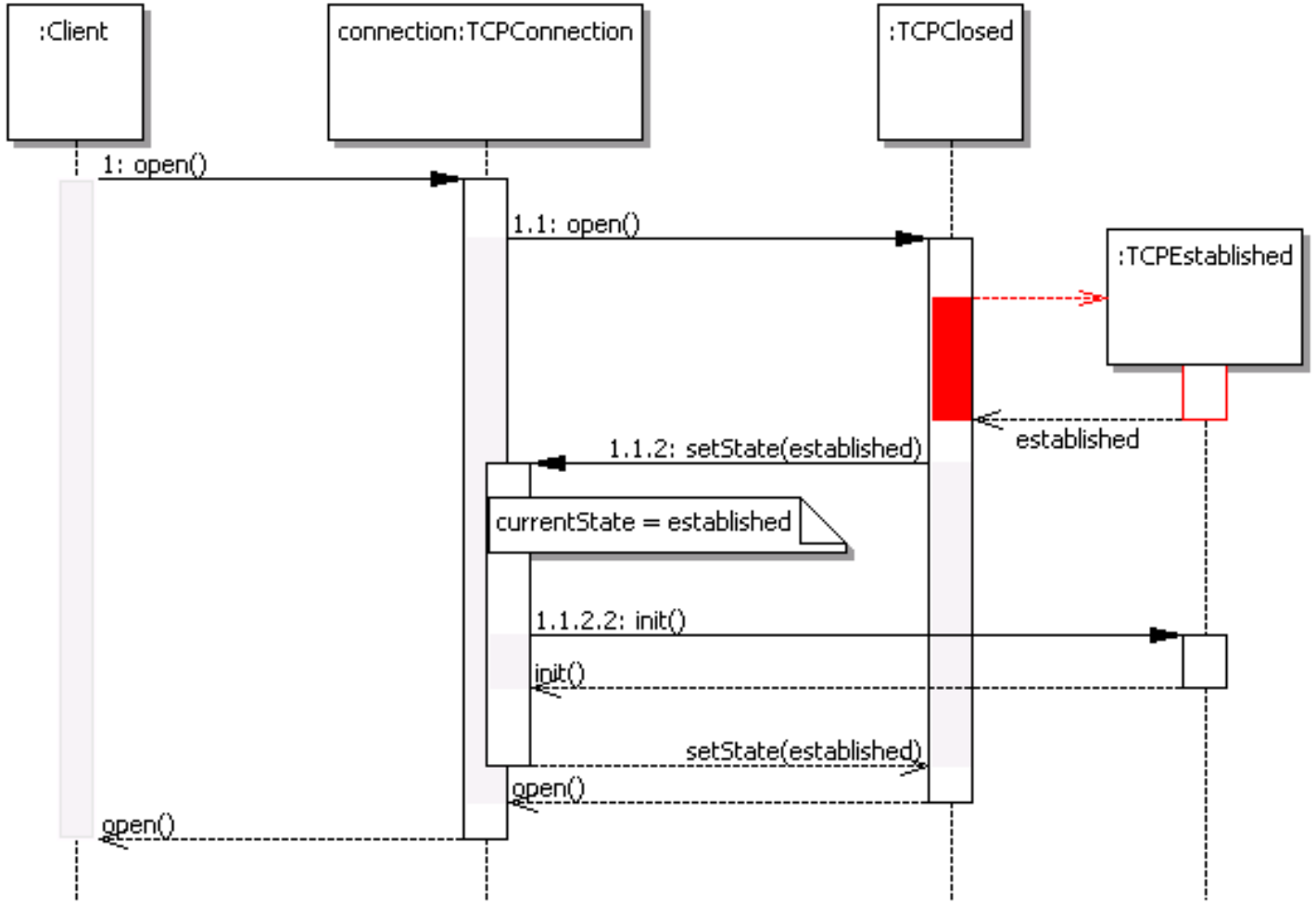
- Anwendbarkeit
  - Wenn das Verhalten eines Objekt von seinem aktuellen Zustand abhängt
  - Wenn Methoden große (zustandsabhängige) Fallunterscheidungen haben; die einzelnen Fälle werden auf Methoden verschiedener Klassen verteilt
- Struktur



  - *TCPConnection*: definiert die Schnittstelle für die Clients; enthält ein internes Zustandsobjekt
  - *TCPState*: definiert die Schnittstelle aller Zustände
  - *TCPEstablished* etc.: implementieren das konkrete Verhalten im jeweiligen Zustand

Beispiel für ein *behavioral* Pattern: State

- Zusammenarbeit der Beteiligten



Beispiel für ein *behavioral* Pattern: State

- Konsequenzen
  -  Zustandsabhängiges Verhalten wird lokalisiert (an einer Stelle behandelt)
  -  Verhalten in unterschiedlichen Zuständen wird auf verschiedene Klassen aufgeteilt
    → Es entstehen mehrere (viele?) Klassen
  -  Zustände und Zustandsübergänge werden explizit gemacht: Übergänge werden in den einzelnen Zuständen oder im umfassenden Objekt ausgelöst (Methode *setState(...)*)