

---

# Vererbung versus Delegation

```
class A { public void x() { z(); }  
        protected void z() { /* A-Code */ ... } }  
class B extends A {  
    protected void z() { /* B-Code */ ... }  
    public void y() { delegate.x(); }  
    private A delegate = new A(); } }
```

- Vererbung: `new B().x()` → B-Code  
Delegation: `new B().y()` → A-Code
- in der Regel kein Überschreiben von Methoden wie `z()`
- Delegation sogar häufiger benötigt als Vererbung
- Unterschied wichtig (da Verwechslung gefährlich)

---

# Template Method

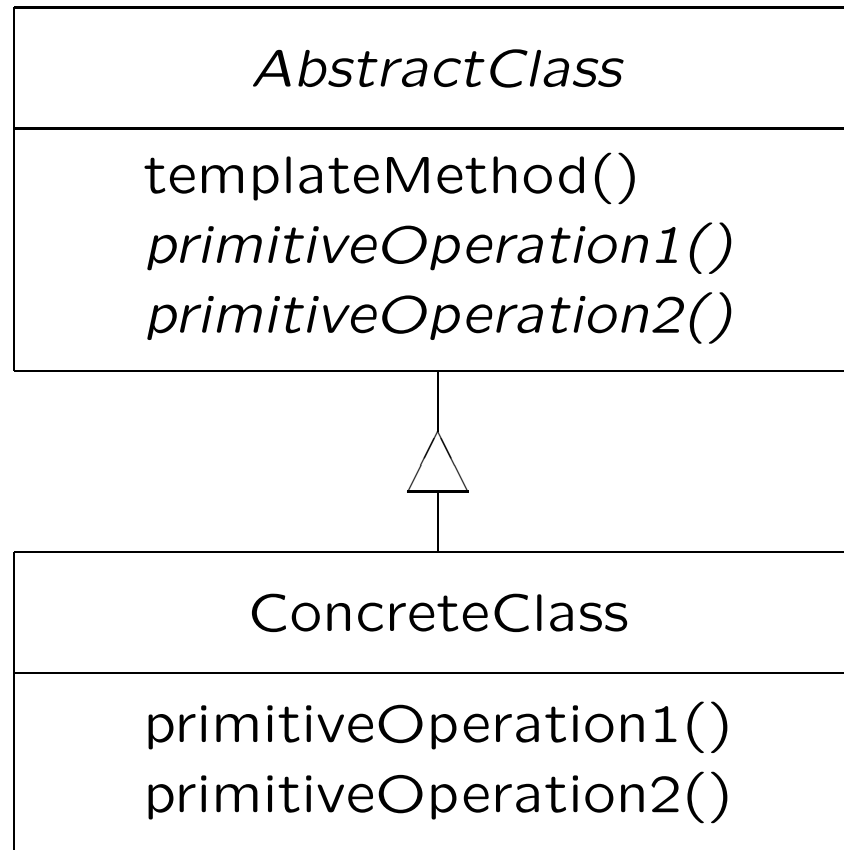
Zweck: definiert Grundgerüst eines Algorithmus,  
Implementierung einzelner Schritte in Unterklasse

Anwendungsgebiete:

- unveränderlicher Teil eines Algorithmus einmal implementiert, veränderliche Teile in Unterklassen
- gemeinsames Verhalten mehrerer Unterklassen lokal zusammengefasst (Refaktorisierung)
- mögliche Erweiterungen durch *hooks* kontrollieren  
hooks in Unterklassen überschreibbar

---

# Template Method: Struktur



---

# Template Method: Eigenschaften

- fundamentale Technik zur direkten Codewiederverwendung
- Oberklasse ruft Methoden der Unterklasse auf (umgekehrte Kontrollstruktur)
- neben konkreten Operationen in „AbstractClass“ meist nur eine von mehreren Arten von Operationen aufgerufen:
  - abstrakte primitive Operationen
  - hooks
  - Factory Methods

---

# Template Method: Implementierung

- möglichst wenige primitive Operationen
- primitive Operationen sind `protected`
- primitive Operationen sind `abstract`, wenn sie überschrieben werden müssen
- Template Method selbst kann `final` sein

---

# Design Pattern: State

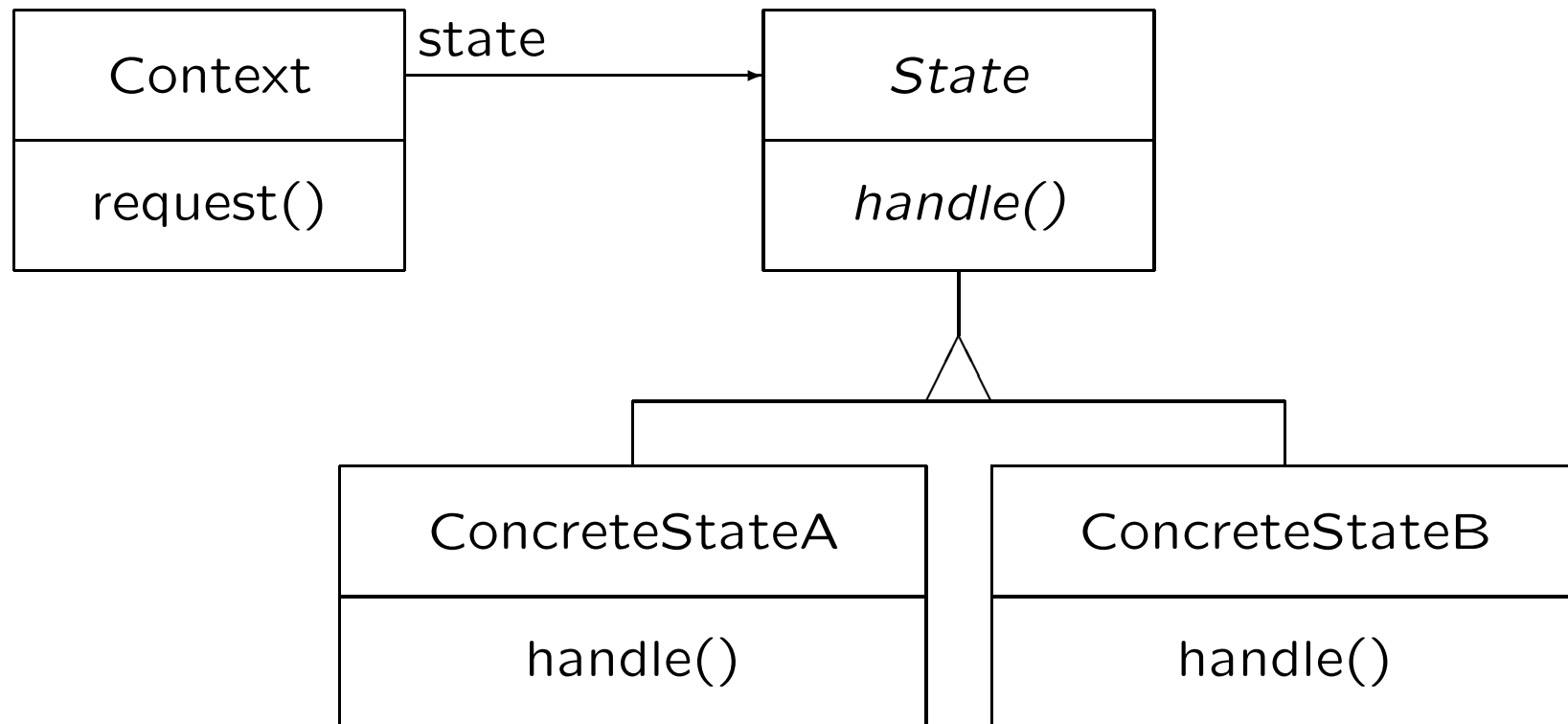
Zweck: Erlaubt einem Objekt eine Verhaltensänderung wenn sich der interne Zustand ändert; scheint Klasse zu ändern

Anwendungsgebiete:

- Verhalten hängt von Zustand ab und ändert sich zur Laufzeit entsprechend dem Zustand
- zur Vermeidung bedingter Anweisungen, die vom Objektzustand (oft durch Konstanten dargestellt) abhängen; jeder Zweig der bedingten Anweisung in eigener Klasse

---

# State: Struktur



---

# State: Eigenschaften

- lokalisiert zustandsspezifisches Verhalten und trennt Verhalten in unterschiedlichen Zuständen
  - leicht um Zustände und Zustandsübergänge erweiterbar
  - Code auf viele Klassen verteilt
- macht Zustandsübergänge explizit
  - nur konsistente Zustände durch atomare Übergänge
- gemeinsame Zustandsobjekte möglich



---

# State: Implementierung

- wer definiert Zustandsübergänge?  
Context: Folgezustand nicht zustandsabhängig  
State: starke Abhängigkeiten zwischen Unterklassen
- Sprungtabelle als Alternative:
  - Fokus auf Zustandsübergängen (nicht Verhalten)
  - Sprungtabelle leicht generierbar und änderbar
- Erzeugung der State-Objekte
  - oft reicht eine Instanz pro Klasse (Singleton)
- dynamische Vererbung als Alternative (Self)

---

# Design Pattern: Observer

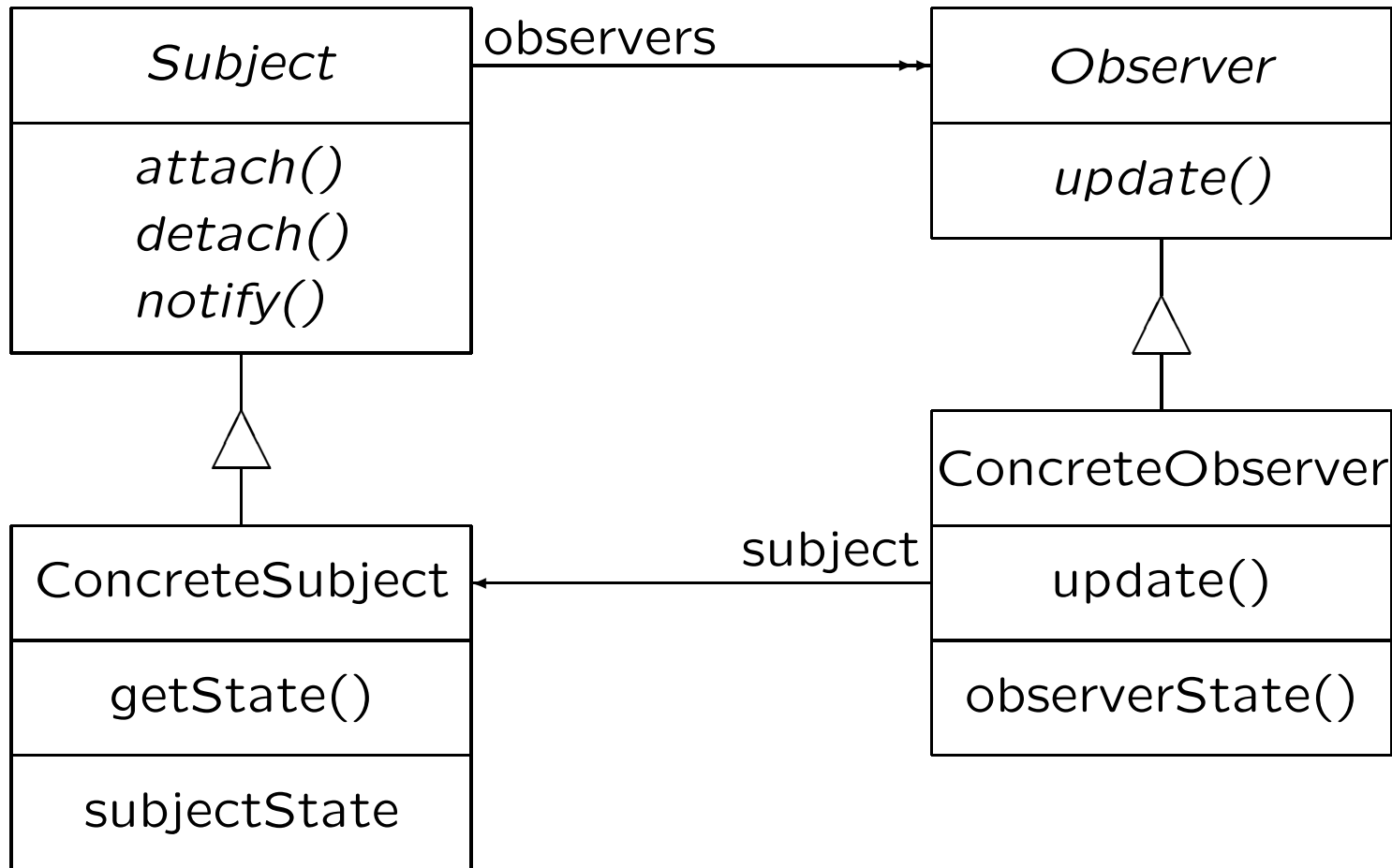
Zweck: Definiert 1-zu-n-Beziehung zwischen Objekten, damit abhängige Objekte benachrichtigt werden, wenn sich der Zustand eines Objekts ändert

Anwendungsgebiete:

- Eine Abstraktion mit zwei Aspekten, einer vom anderen abhängig; Kapselung in getrennte Objekte macht Aspekte unabhängig voneinander änder- und wiederverwendbar
- Wenn eine Zustandsänderung weitere notwendig macht und statisch unbekannt ist, welche Objekte zu ändern sind
- Wenn Objekten etwas mitgeteilt werden soll ohne zu wissen, wer diese Objekte sind (keine enge Kopplung)

---

# Observer: Struktur



---

# Observer: Eigenschaften

- abstrakte Kopplung zwischen Subject und Observer
  - können zu unterschiedlichen layers gehören
- broadcast erfolgt automatisch
  - Observer jederzeit hinzufügen und wegnehmen
- unerwartete Updates durch fehlende Information möglich
  - Ursachen unerwünschter Updates schwer zu finden
  - oft hohe Kosten von updates schwer abschätzbar

---

# Observer: Implementierung

- Subject als Argument von update (zur Unterscheidung)
- Wer triggert notify?
  - Client → fehleranfällig;
  - zustandsänd. Subject-Operationen → unnötige updates
- Subject-Zustand soll vor notify konsistent sein
- Subjects entfernen → auf Referenzen in Observers achten
- update mit viel/wenig Information (push/pull-Modell)
- Observers registrieren sich nur für bestimmte Aspekte