OOP

Software-Entwurfsmuster (weitere)

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()` $\rightarrow$ B-Code

Delegation: `new B().y()` $\rightarrow$ A-Code

Template-Method

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

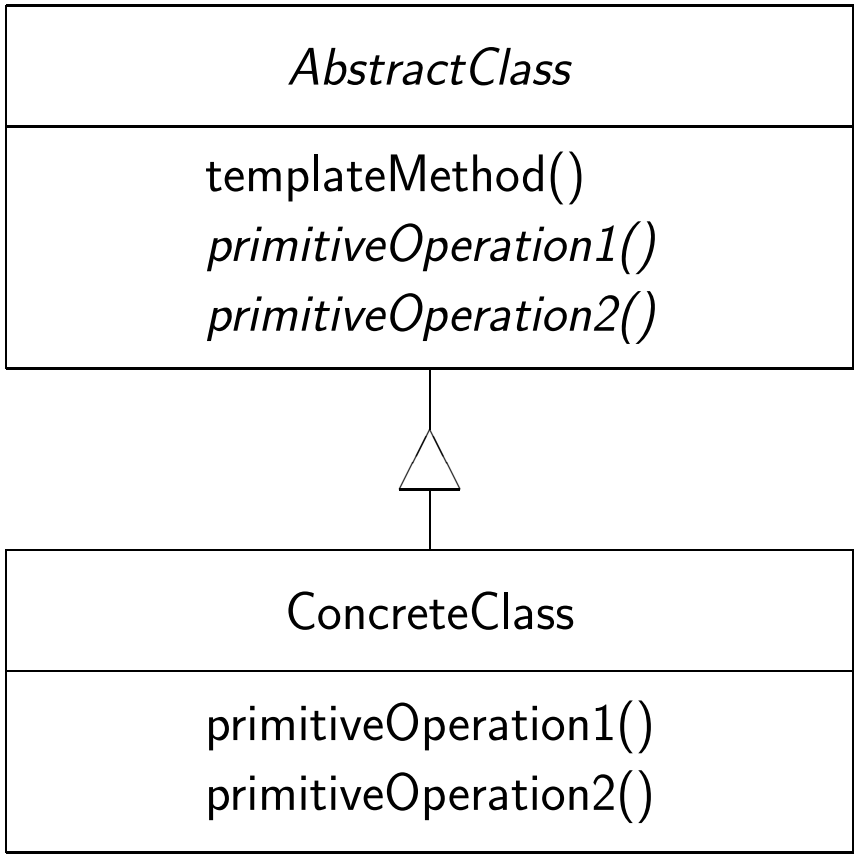
Anwendungsgebiete:

unveränderlicher Teil eines Algorithmus einmal implementiert,
veränderliche Teile in Unterklassen verlagert

gemeinsames Verhalten mehrerer Unterklassen lokal zusammengefasst
(Ergebnis einer Refaktorisierung)

mögliche Erweiterungen durch **Hooks** kontrollieren
(Hooks sind Methoden, die in Unterklassen überschrieben werden sollen)

Template-Method: Struktur



Template-Method: Eigenschaften

fundamentale Technik zur direken 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

Aufgabe: Abstrakte Methode oder Hook

Such Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Fragen:

1. Was unterscheidet abstrakte Methoden von Hooks?
2. Wann sind abstrakte Methoden besser geeignet, wann Hooks?
3. Warum spricht man von Hooks (zum Überschreiben gedachte Methoden), wenn ohnehin fast jede Methode überschrieben werden kann?

Zeit: 2 Minuten

Design-Pattern: Observer

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

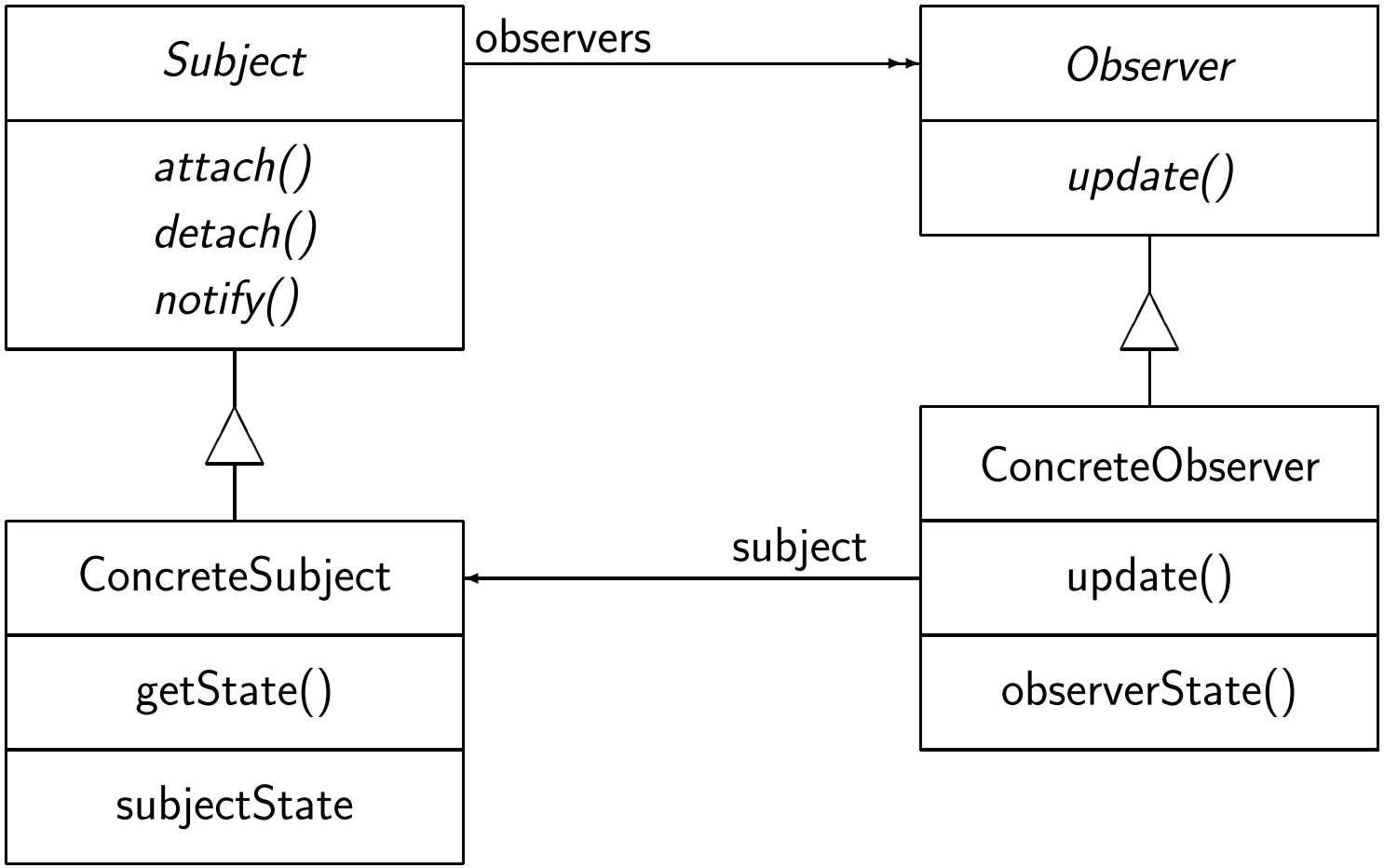
Anwendungsgebiete:

Abstraktion mit zwei Aspekten, einer vom anderen abhängig;
Kapselung in getrennte Objekte macht Aspekte unabhängig voneinander änder- und wiederverwendbar

Zustandsänderung macht weitere Zustandsänderungen notwendig,
wobei man aber nicht statisch weiß, welche Objekte zu ändern sind

Objekten soll etwas mitgeteilt werden ohne zu wissen, wer diese Objekte sind (schwache 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;
zustandsändernde 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

Design-Pattern: State

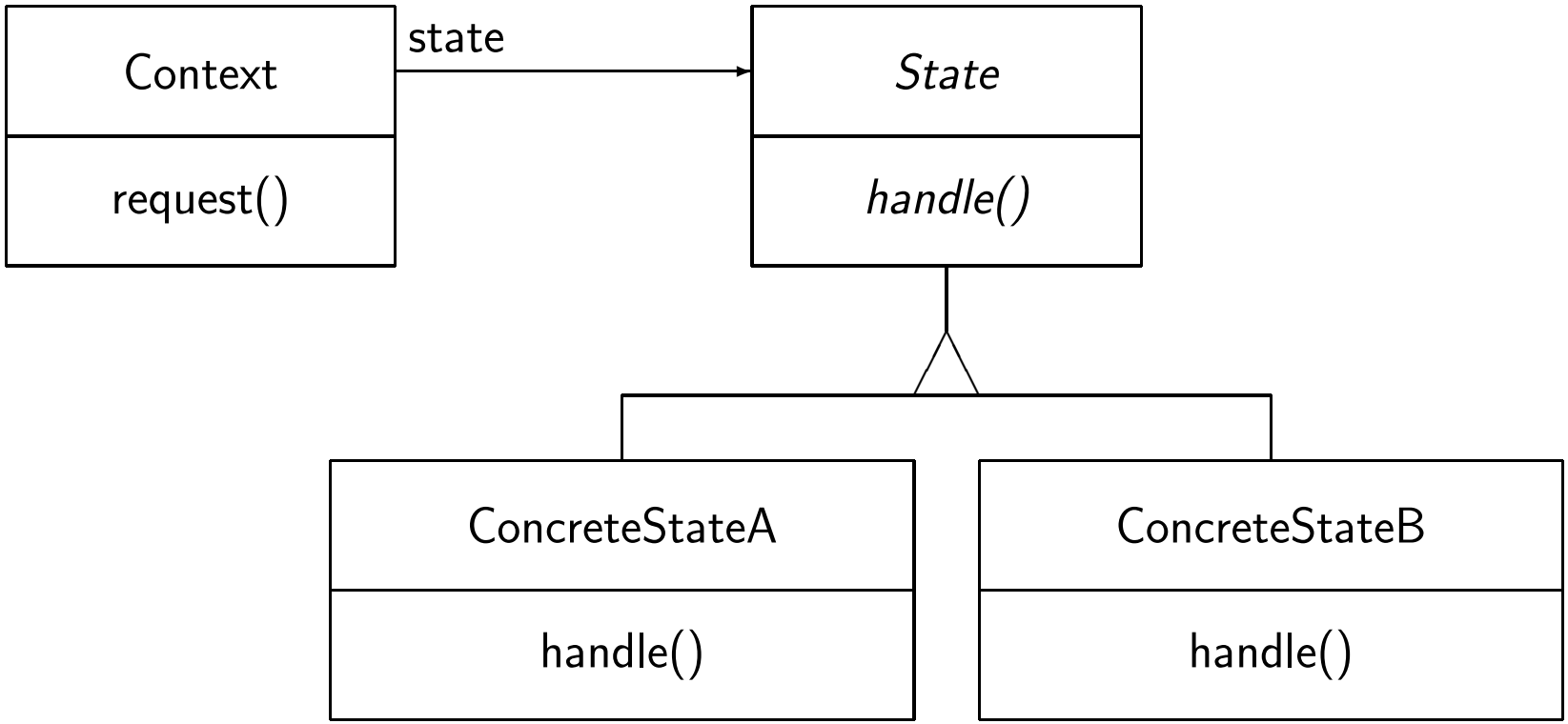
Zweck: Erlaubt einem Objekt eine Verhaltensänderung wenn sich der interne Zustand ändert; scheint Klasse zu ändern

Anwendungsgebiete:

Verhalten hängt vom 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 (Parser, Lexer), kaum direkt wartbar

State-Objekte: oft reicht eine Instanz pro Klasse (Singleton)

dynamische Vererbung als Alternative (Decorator)