Tagesprogramm

Iterator

Template-Method

Observer

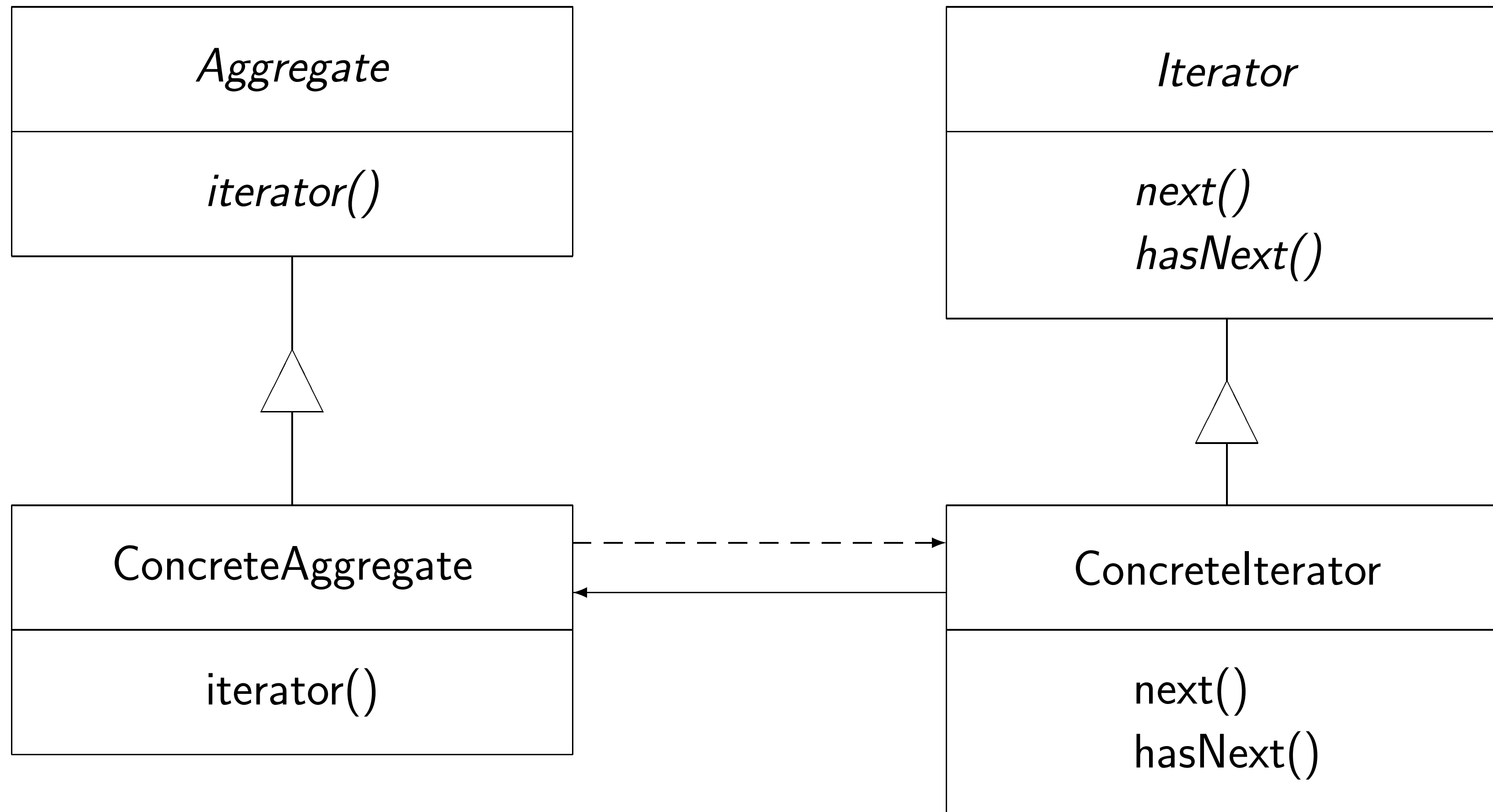
Iterator (Cursor)

Zweck: sequentieller Zugriff auf Elemente eines Aggregats

Anwendungsgebiete:

- Zugriff auf Aggregatinhalt wobei innere Darstellung gekapselt bleibt
- mehrere Abarbeitungen des Aggregatinhalts
- einheitliche Schnittstelle für Abarbeitung verschiedener Aggregatstrukturen (polymorphe Iterationen)

Iterator: Struktur



Eigenschaften

unterstützt unterschiedliche Arten der Abarbeitung von Aggregaten (mehrere Iteratorklassen pro Aggregatklasse)

vereinfacht Schnittstelle von *Aggregate*

mehrere gleichzeitige Abarbeitungen möglich

Iterator: Implementierungshinweise

externe Iteratoren flexibler, interne Iteratoren einfacher

> **extern:** Anwender holt nächstes Element (siehe Beispiele)
> **intern:** Iterator wendet Operation auf alle Elemente an (map, fold, . . .)

interne Iteratoren besser wenn komplexe Beziehungen zwischen Elementen bei Abarbeitung zu berücksichtigen sind

Algorithmus zum Durchwandern des Aggregats in Aggregat oder Iterator definiert (beides gleichzeitig wenn Iterator innere Klasse des Aggregats)

Aggregatänderungen während der Abarbeitung beachten (robuster Iterator)

auch auf leeren Aggregaten brauchbar

Aufgabe: Iteratoren

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

1. Warum verwendet man Iteratoren und greift nicht direkt auf Elemente zu?
2. Wann verwendet man eher interne Iteratoren, wann externe?
3. Warum implementiert man Iteratoren häufig über innere Klassen?

Zeit: 2 Minuten

Template-Method

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

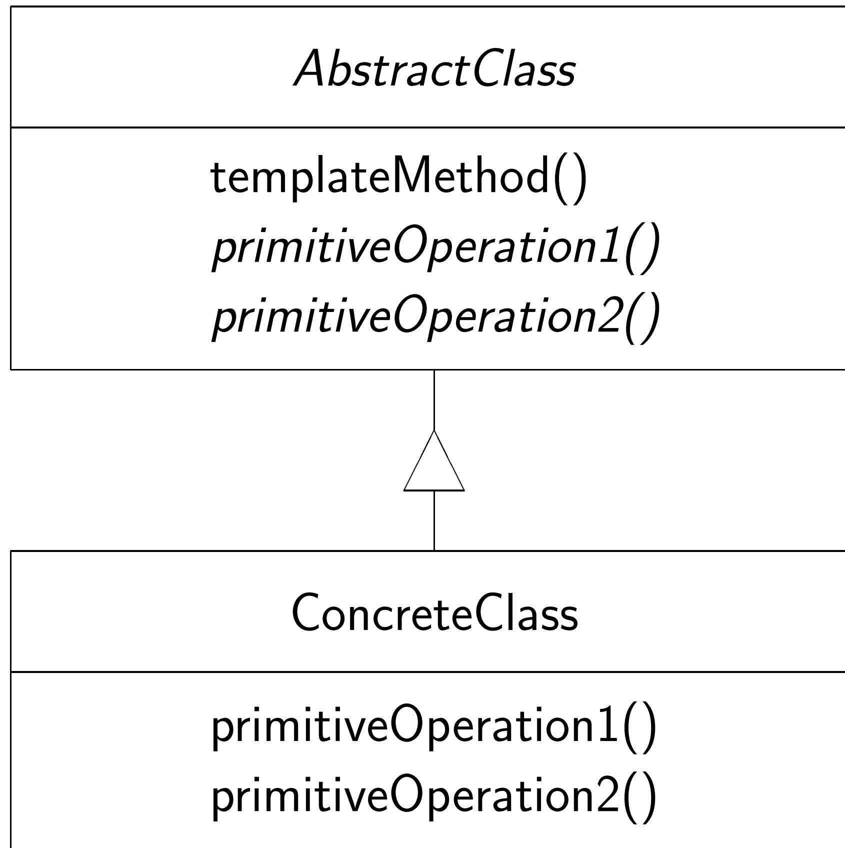
Anwendungsgebiete:

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)

Struktur



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

Implementierung

möglichst wenige primitive Operationen

primitive Operationen sind `protected`

primitive Operationen sind `abstract`, wenn sie überschrieben werden müssen

Template-Method selbst kann `final` sein

Observer

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

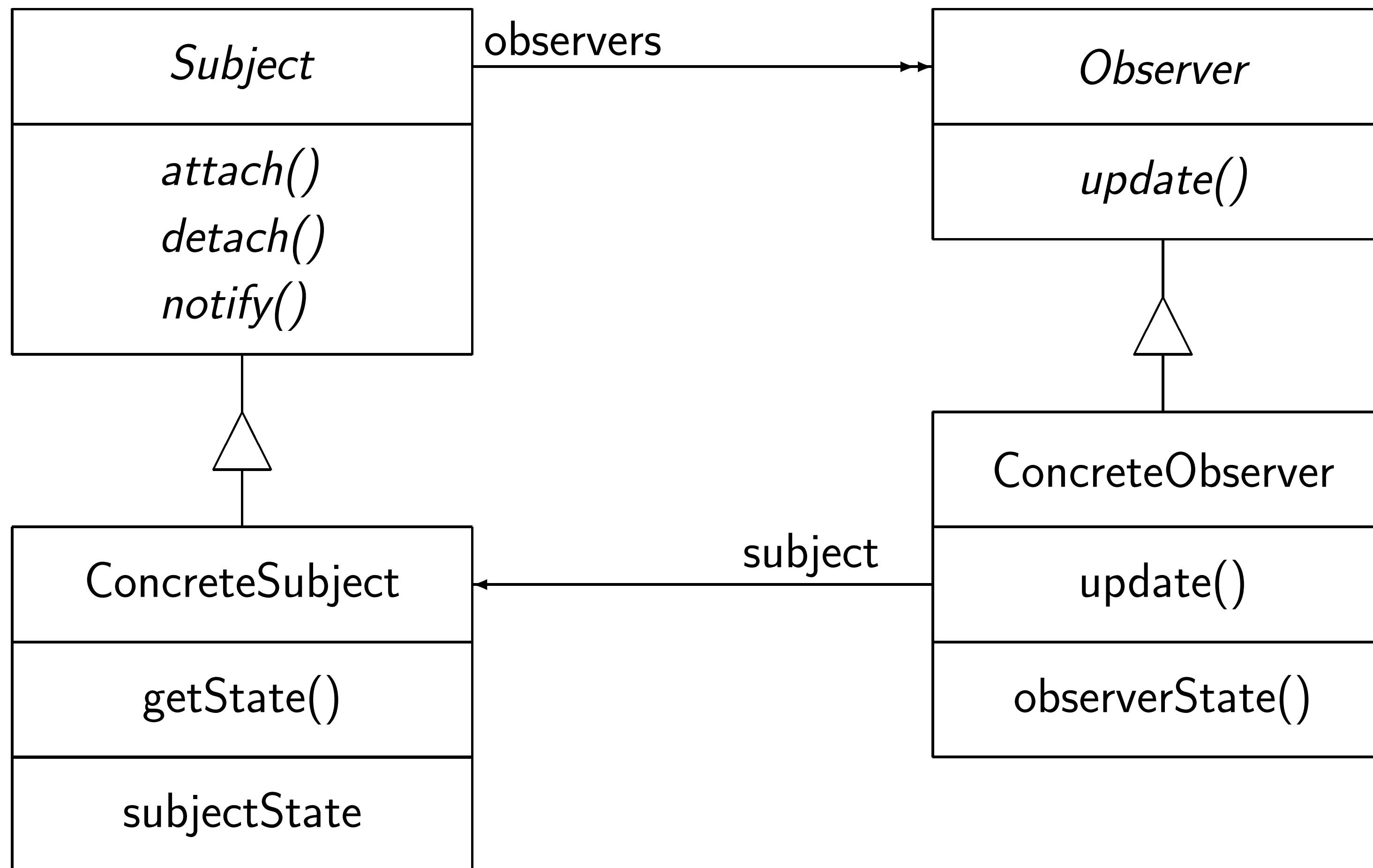
Anwendungsgebiete:

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)

Struktur



Eigenschaften

abstrakte Kopplung zwischen Subject und Observer

$\longrightarrow$ können zu unterschiedlichen Layers gehören

Broadcast erfolgt automatisch

$\longrightarrow$ Observer jederzeit hinzufügen und wegnehmen

unerwartete Updates durch fehlende Information möglich

$\longrightarrow$ Ursachen unerwünschter Updates schwer zu finden

$\longrightarrow$ oft hohe Kosten von Updates schwer abschätzbar

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