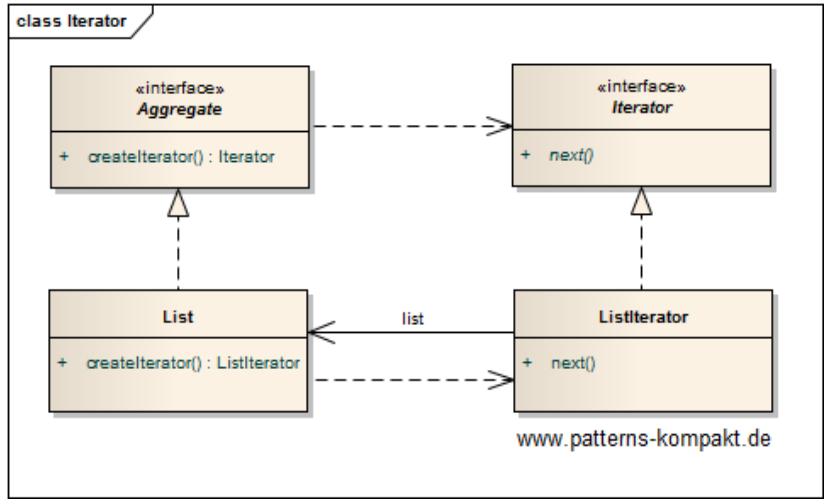
Iterator (Verhaltensmuster)

=> Zweck: Sequenzieller Zugriff auf Elemente einer Sammlung, ohne deren interne Struktur zu kennen



Vorteile:

Kapselung & Sicherheit:

Lose Kopplung zwischen Client und Sammlung

Kontrollierter und konsistenter Zugriff

Flexibilität & Wiederverwendung:

Einheitliche Verwendung für verschiedene Datenstrukturen

Zeitgleich mehrere Iteratoren auf derselben Sammlung

Einfache Erweiterung um neue Iterator-Typen

Design-Prinzipien:

Single Responsibility – Iteration von restlicher Logik getrennt

Open/Closed Principle – Neue Iterator-Typen ohne Änderung bestehender Klassen

Nachteile:

Komplexität und Overhead steigt bei einfachen Anwendungsfällen

Memory-Verbrauch durch zusätzliche Iterator-Objekte

Code-Beispiel:

- Aggregate teilweise auch Container oder IterableCollection genannt

```
type Aggregate[T any] interface {
    GetIterator() Iterator[T]
}
```

```
type Iterator[T any] interface {
    next() T
    hasNext() bool
}
```

Einsatzgebiete:

- Collections/Listen durchlaufen
- Dateisysteme traversieren
- Baumstrukturen durchlaufen