Technische Hochschule Ingolstadt

Fakultät für Elektrotechnik und Informatik

Zukunft in Bewegung

Entwurfsmuster

Software Engineering

Prof. Dr. Bernd Hafenrichter

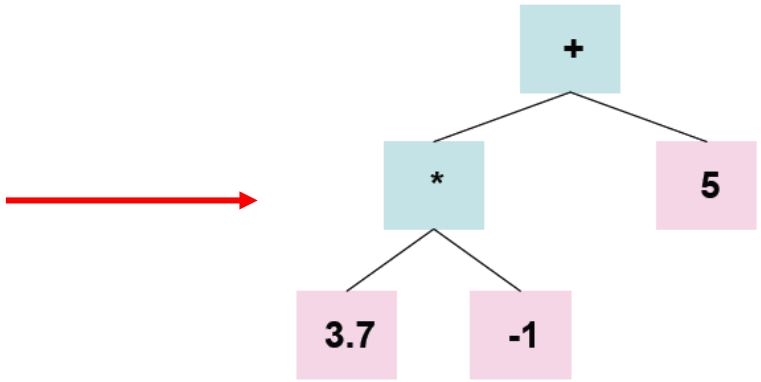
Motivation

- Objekte haben neben statischen Beziehung auch dynamisches Verhalten
- Mit Hilfe von Verhaltensmustern sollen typische Interaktionen und Verantwortlichkeiten zwischen (komplexen-) Objektkompsitionen beschrieben werden

Visitor

- **Beispiel aus dem Compilerbau**:

```
3.7 * (-1) + 5
```



- **Typische Fragestellung**:
  - Code-Generierung
  - Typisierung
  - Rechenergebnis (Interpreter)

- **Gemeinsamkeit**:
  - Alle erfordern das Traversieren des Baumes
  - Die Reihenfolge des Traversierens hängt von der Struktur ab

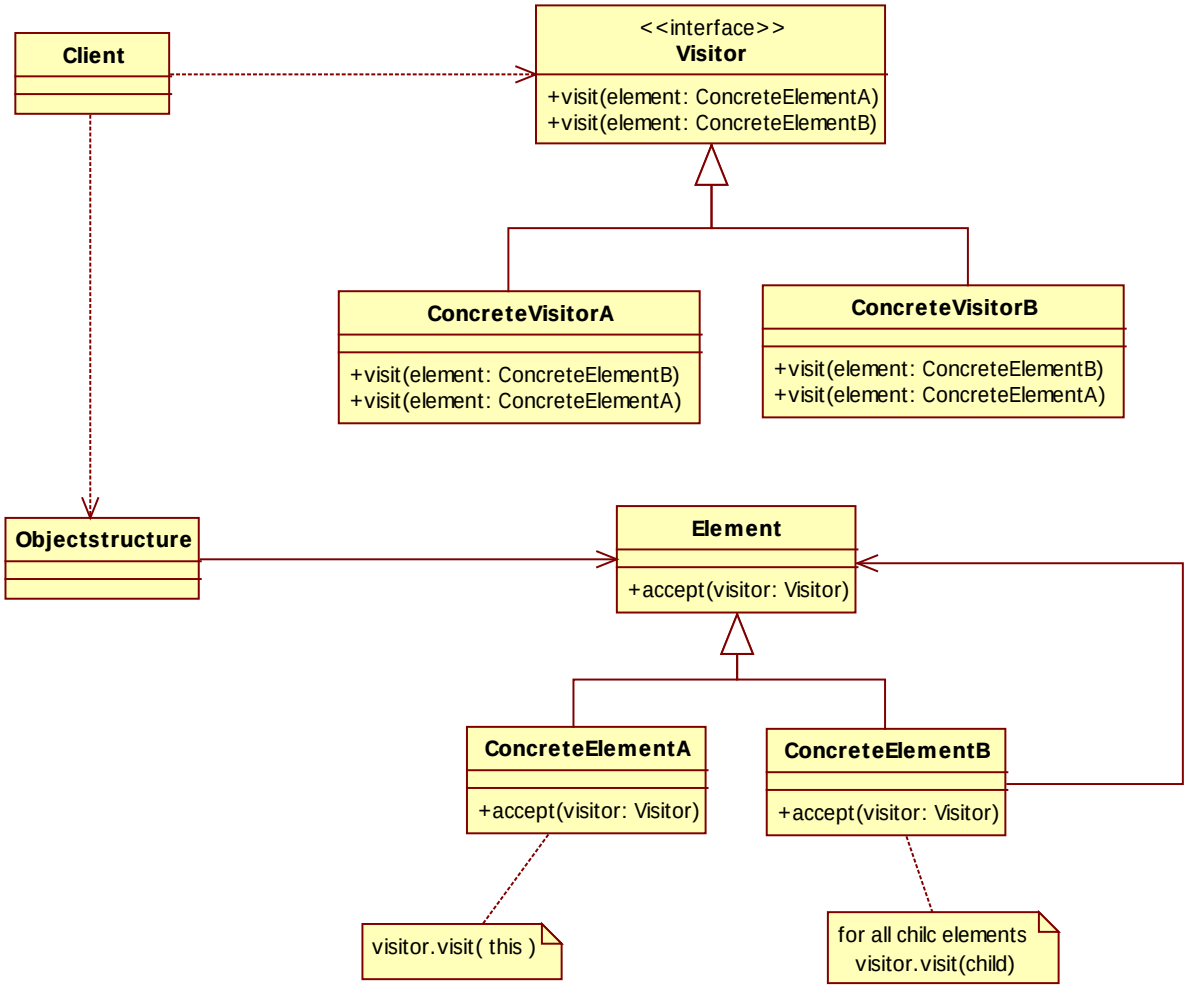
Visitor

- **Zweck**:
  - Stelle eine allgemeine Möglichkeit zur Traversierung einer beliebigen Datenstruktur zur Verfügung
  - Trenne die Datenstruktur von den Operationen welche auf der Datenstruktur auszuführen sind

- **Beispiel:**
  - Compilerbau: Erzeugen des Quellcodes
  - Interpreter: Berechne dynamisch das Ergebnis eines Ausdrucks

Visitor

- **Lösung:**



Visitor

Visitor:

- deklariert für jede Klasse konkreter Elemente eine Besuchsfunktion

ConcreteVisitor :

- implementiert Besuchsfunktionen
- jede Besuchsfunktion ist ein Fragment des Algorithmus, welcher auf der gesamten Objektstruktur angewendet wird
- lokaler Zustand dient als Kontext für den Algorithmus

Element:

- deklariert eine Schnittstelle zum Empfang eines Besuchers

ConcreteElement

- implementiert den Empfang eines Besuchers
- Definiert wie das konkrete Element traversiert wird
- Gibt Visitor an Kindelement weiter

Visitor

Bewertung:

Vorteile:

- Neue Operationen lassen sich leicht durch die Definition neuer Besucher hinzufügen.
- Besucher können über mehreren Klassenhierarchien arbeiten.

Nachteil:

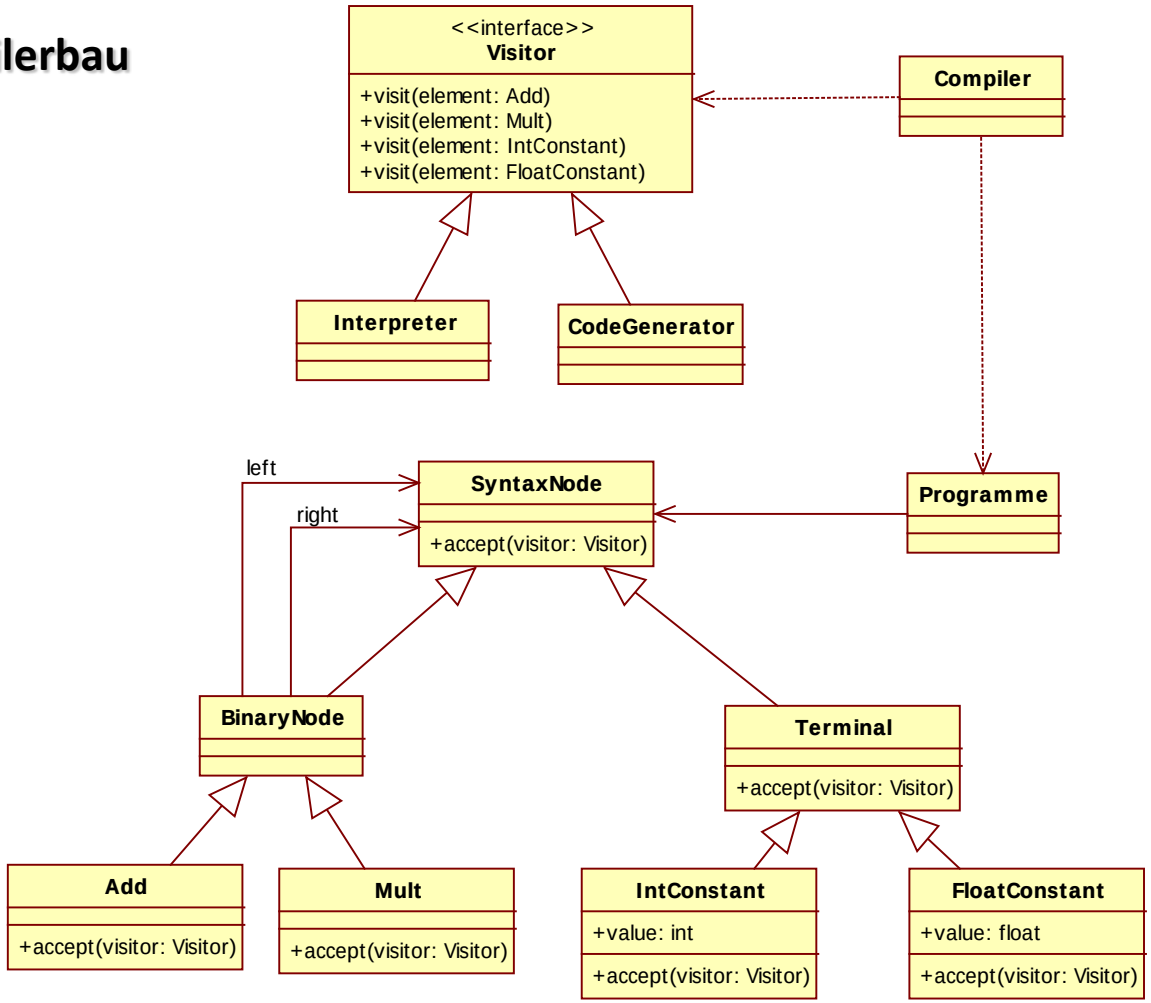
- Die gute Erweiterungsmöglichkeit der Klassen von Besuchern muss mit einer schlechten Erweiterbarkeit der Klassen der konkreten Elemente erkauft werden.
- Neue Klassen konkreter Elemente erfordern Erweiterungen in allen Besuchern.

Visitor

- **Verwendung (allgemein):**
  - Viele unterschiedliche (nicht verwandte) Operationen sollen auf die gleiche Datenstruktur angewandt werden
  - Neu Operationen sollen einfach hinzugefügt werden können
  - Die Objektstruktur/Datenstruktur ändert sich selten

Visitor

- **Beispiel: Compilerbau**



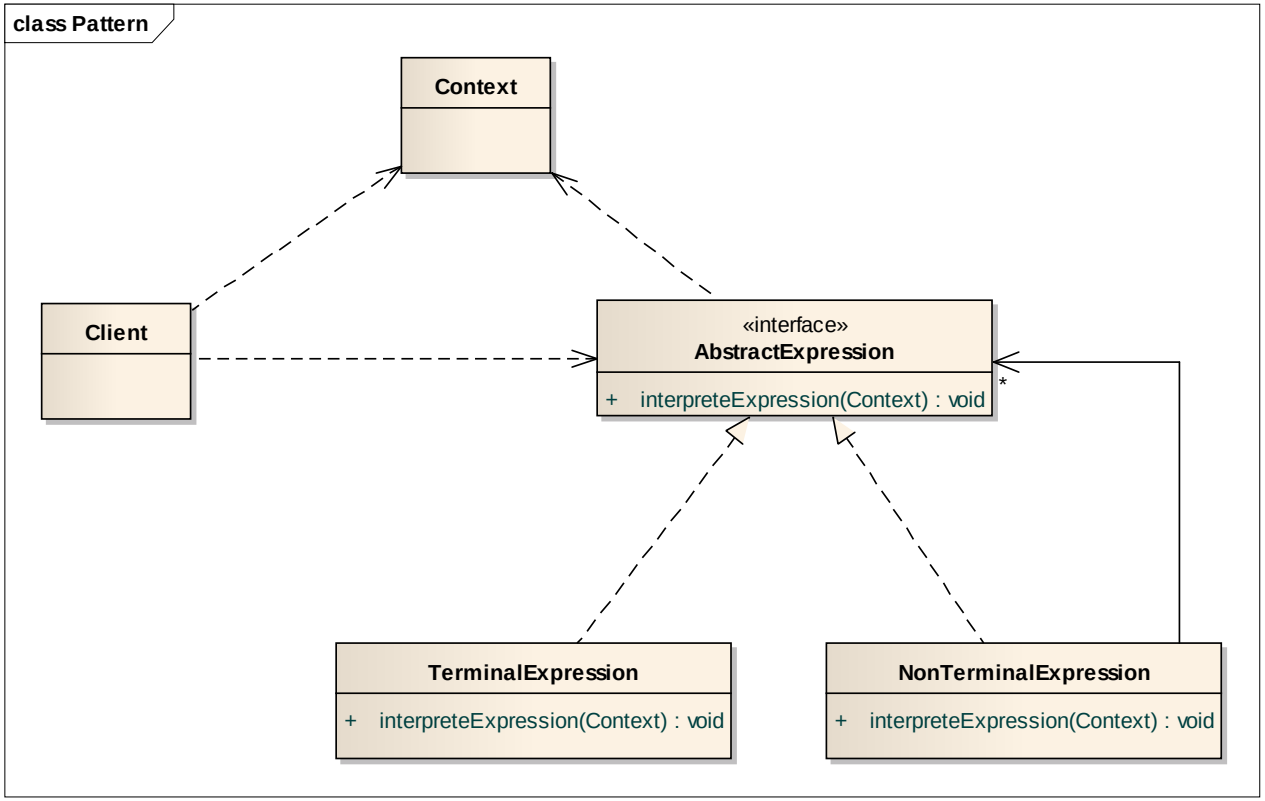
Interpreter

- **Zweck/Ziele**:
  - Interpretiere die Sätze einer Grammatik dynamisch

- **Beispiel:**
  - Ein wissenschaftlicher Taschenrechner der komplexe Formeln berechnen kann
  - Eine Skript-/Makrosprache welche zur Erweiterung eines Programms genutzt werden kann

Interpreter

- **Lösung:**



Interpreter

Context:

- Enthält Information für die Berechnung welche zwischen den Ausdrücken übergeben werden kann

AbstractExpression:

- Abstrakte Sicht auf einen Ausdruck. Der Ausdruck berechnet ein Ergebnis. Das wie ist den konkreten Implementierungen überlassen.

TerminalExpression:

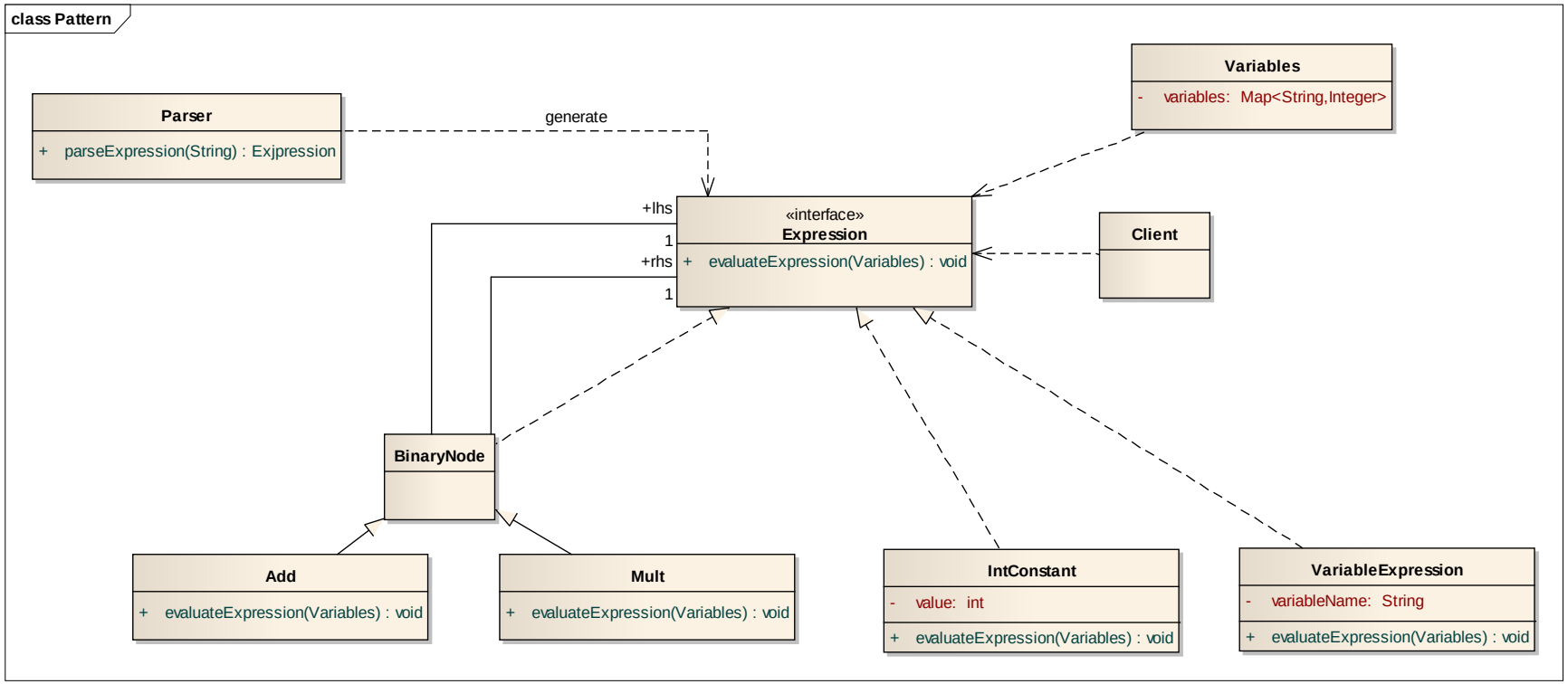
- Ein einfacher Ausdruck welcher durch sich selbst beschrieben ist und keine weiteren Unterausdrücke benötigt.

NonTerminalExpression:

- Ein zusammengesetzter Ausdruck dessen Ergebnis sich aus den Teilergebnissen der Unterausdrücke berechnet.

Interpreter

- **Beispiel Berechnung einer Formel:**



Interpreter

Bewertung:

Vorteile:

- Die Grammatik kann durch dieses Entwurfsmuster leicht geändert oder erweitert
- Derselbe Satz oder Ausdruck kann durch Ändern des Kontextes immer wieder auf neue Art und Weise interpretiert werden

Nachteil:

- Für komplexe Grammatiken und sehr große Sätze ist das Interpretermuster ungeeignet, da die Klassenhierarchie zu groß wird und die Effizienz bei großen Syntaxbäumen leidet
- Sollen komplexe Grammatiken verarbeitet werden, eignen sich Parsergeneratoren besser

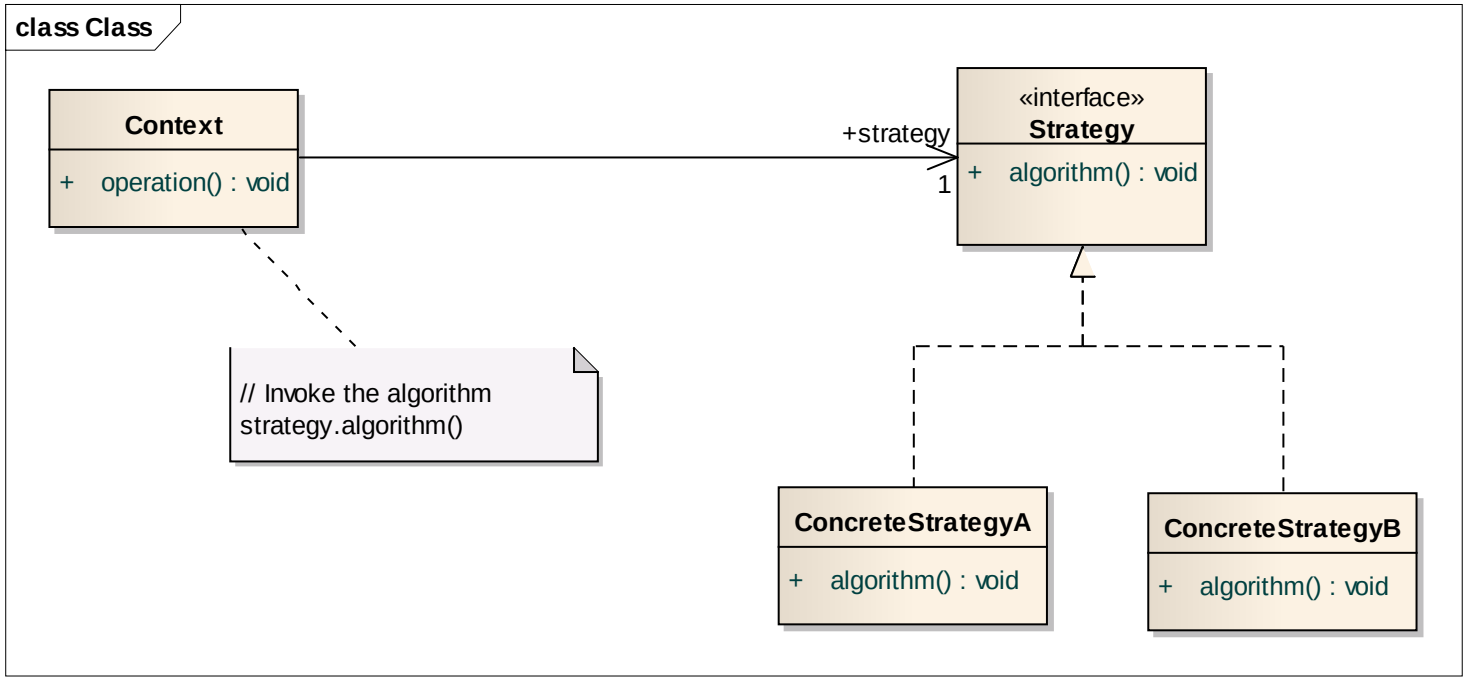
Strategy

- **Zweck/Ziele**:
  - Das Strategiemuster soll es erlauben dass ein ganzer Algorithmus in Form einer Kapsel ausgetauscht wird
  - Eine Anwendung möchte verschiedene (Lösungs-)Strategien für ein Problem anbieten

- **Beispiel:**
  - Landesabhängige Berechnung von Steuern
  - Ein Packer (zip-tool) das verschiedene Kompressionsalgorithmen bieten soll
  - Layout von Komponenten in einer graphischen Oberfläche

Strategy

- **Lösung:**



Strategy

Context:

- Stellt eine oder mehrere Methoden bereit welche den Algorithmus der Strategie benutzen
- Dadurch bleiben die Methoden flexibel

Strategy:

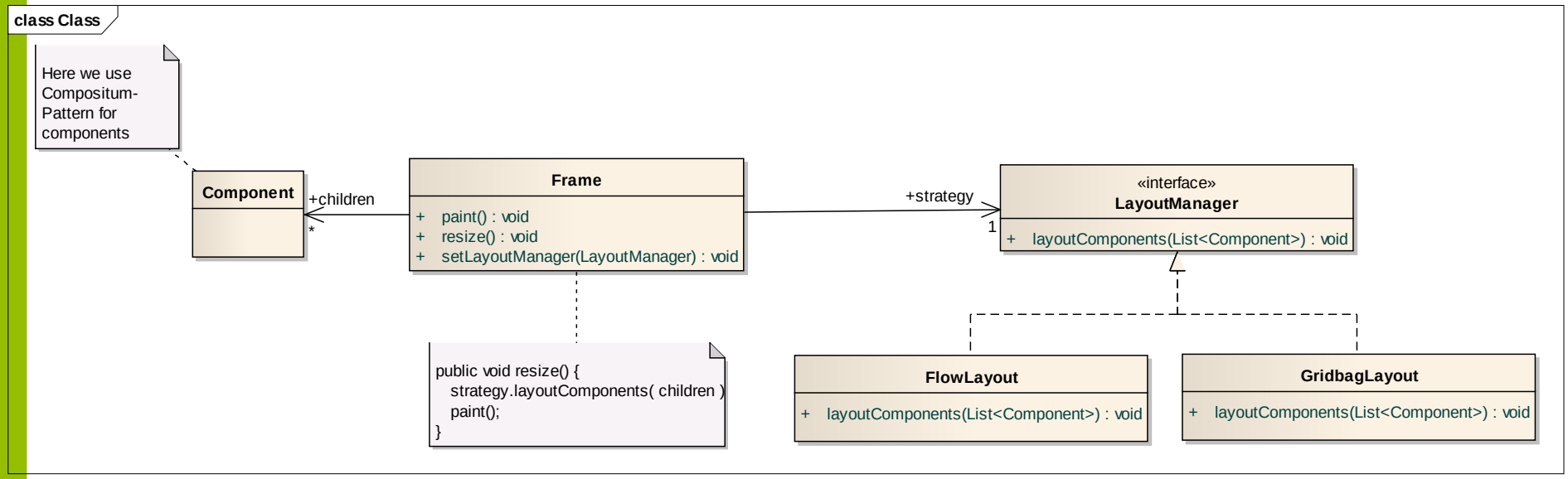
- Basisinterface welches von allen konkreten Algorithmen implementiert wird

ConcreteStrategy<X>:

- Implementierung einer Lösungsstrategie für den geforderten Algorithmus

Strategy

- **Beispiel LayoutManager:**



Strategy

Bewertung:

Vorteile:

- Durch das Strategiemuster wird die Flexibilität des nutzenden Kontext erhöht
- Es ist möglich Familien von Algorithmen zu definieren. Jeder Algorithmus ist gekapselt und kann beliebig ausgetauscht werden
- Der nutzende Kontext ist unabhängig von einer Konkreten Implementierung

Nachteil:

- Die Applikation muss die zu nutzende konkrete Strategie kennen (Konfiguration)
- Evtl. zu viele Klassen/Strategien welche selten genutzt werden und dadurch den Testaufwand erhöhen
- Parameterübergabe zwischen Kontext und Strategie ist aufwendig

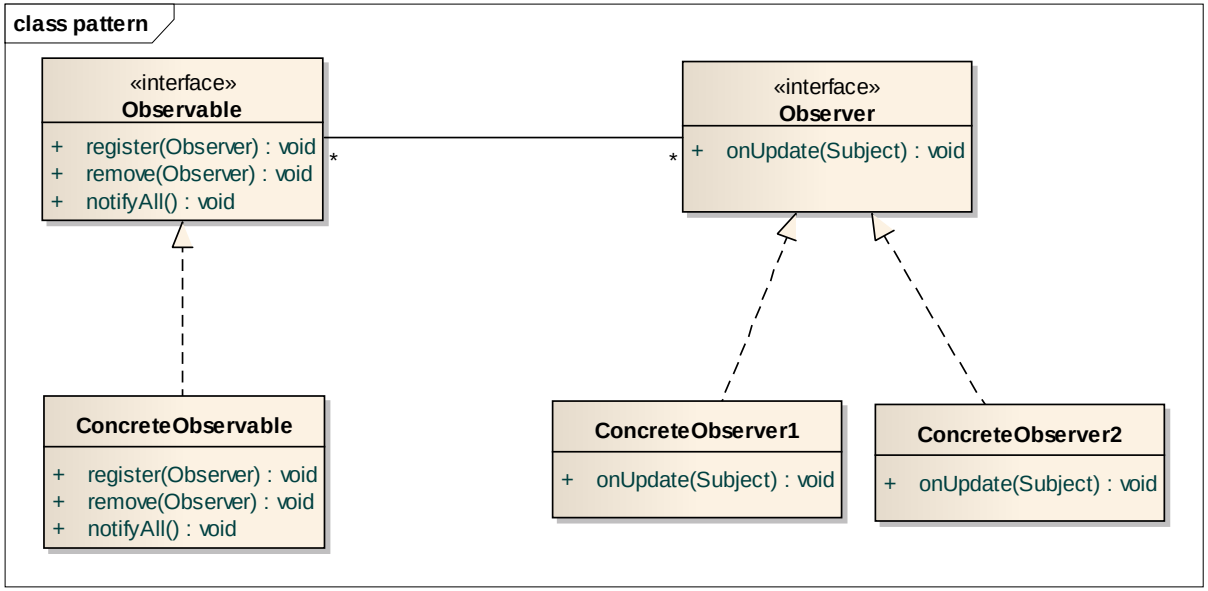
Observer

- **Zweck/Ziele**:
  - Ein Objekt möchte andere Objekte über eine Veränderung des eigenen Zustandes informieren
  - Lose Koppelung zwischen den Objekten
  - Die Menge der informierten Objekte soll sich ändern können.

- **Beispiel:**
  - Aktualisierung einer Anzeige wenn sich ein Eingabewert ändern
  - Neu kompilieren einer Klasse wenn der Quelltext geändert wird

Observer

- **Lösung:**



Interpreter

Observable:

- Stellt ein Objekt dar das beobachtet werden kann

Observer:

- Über diese Callback-Interface können andere Objekte bei einer Änderung benachrichtig werden.

ConcreteObservable:

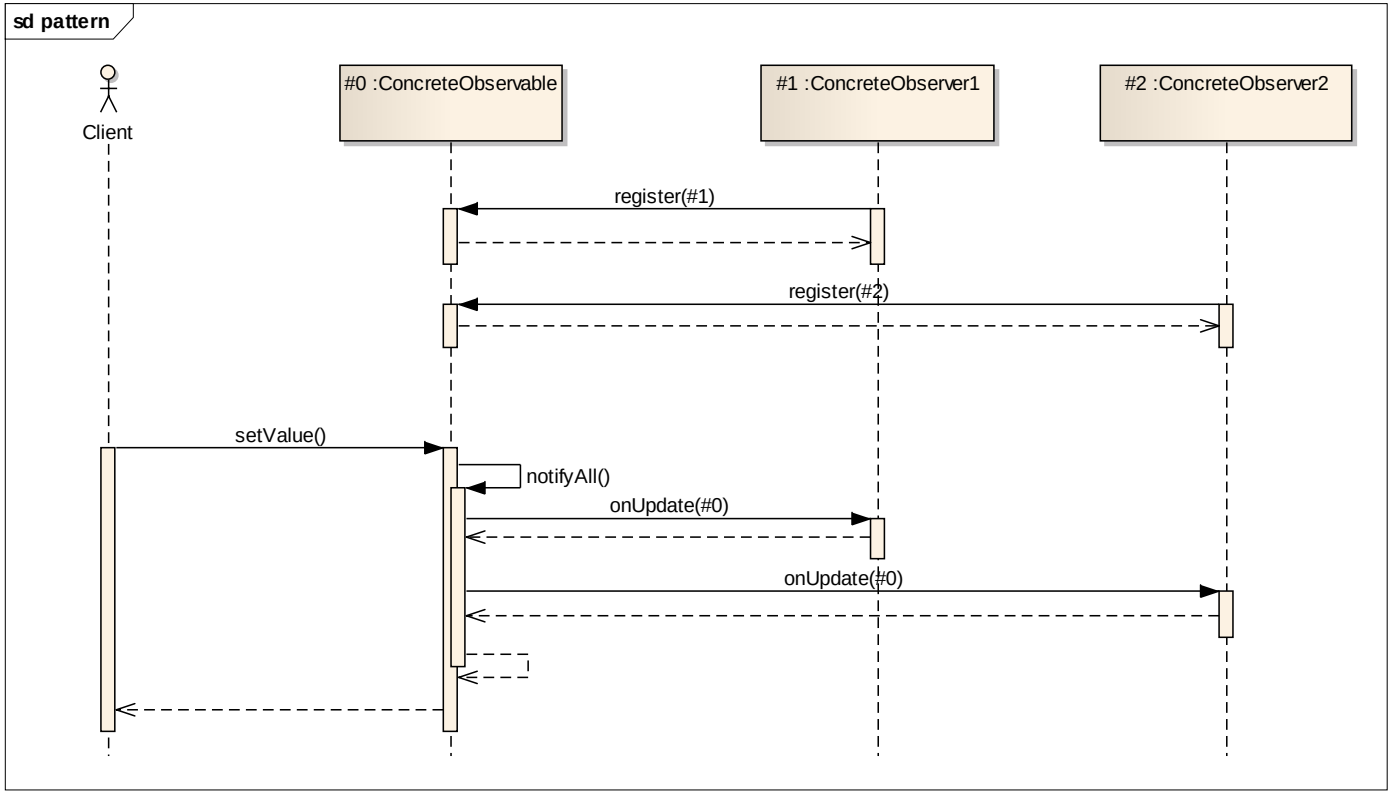
- Ein Objekt das beobachtet werden kann. Wir der Zustand über setValue geändert kann über notifyAll alle registrierten Beobachter benachrichtigt werden

ConcreteObserver1, ConcreteObserver2:

- Zwei konkrete Implementierung welche bei Änderung von ConcreteObservable benachrichtig werden.

Observer

- **Dynamisches Verhalten**



Observer

Bewertung:

Vorteile:

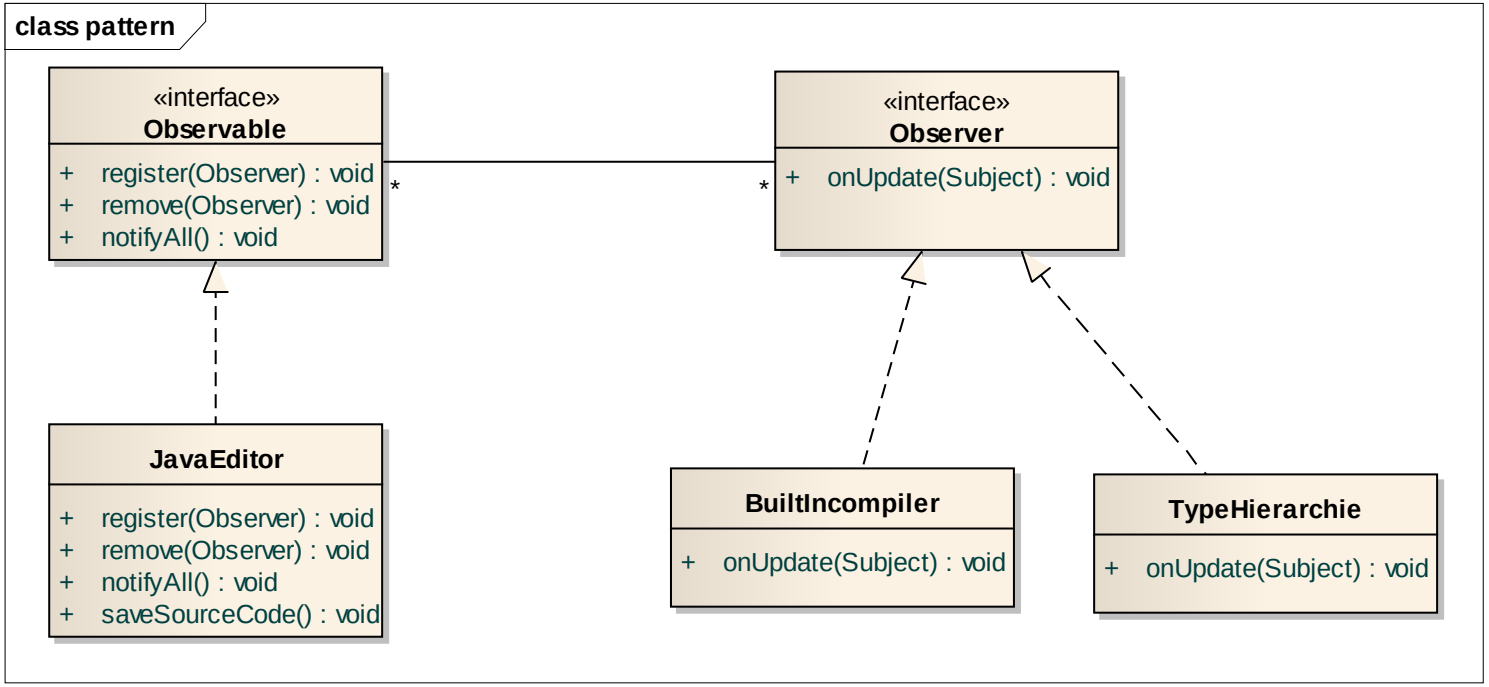
- Neue Observer können jederzeit hinzugefügt werden
- Observer und Observable sind unabhängig voneinander
- Unabhängige Weiterentwicklung von Observer und Observable
- Unabhängige Wiederverwendung ist möglich

Nachteil:

- Bei vielen Observer-Objekten kann die Benachrichtigung lange dauern
- Problem der Endlosschleife wenn die Zustand von Observable durch Observer geändert wird
- Jeder Beobachter wird informiert auch wenn er die Nachricht nicht benötigt

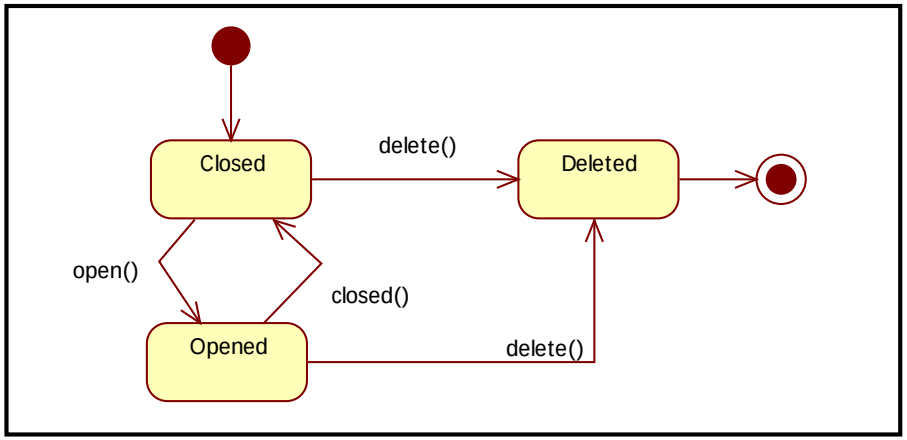
Observer

Beispiel:



State

- **Beispiel File-Objekt**:



- **Typische Fragestellung**:
  - Wie reagiert das Objekt auf verschieden Methodenaufrufe in Abhängigkeit vom internen Zustand??
  - Wie kann der Zustandsautomat einfach erweitert werden

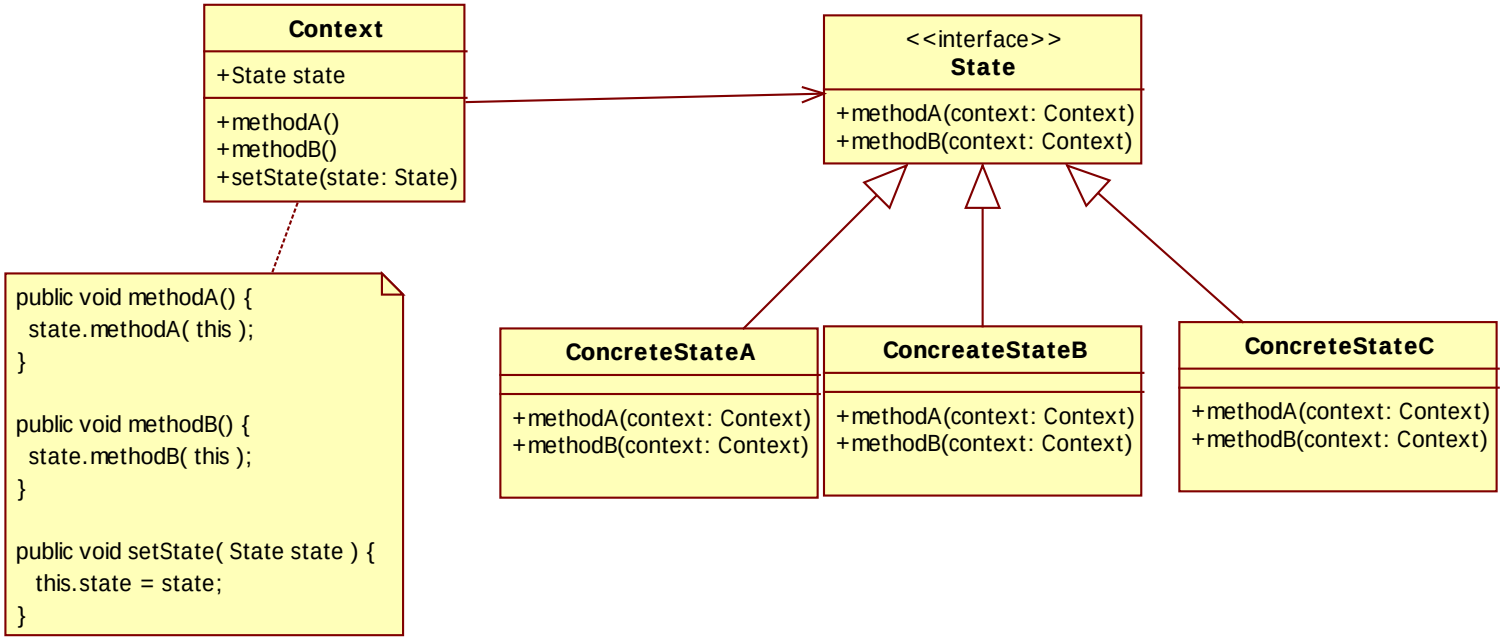
State

- **Zweck/Ziele**:
  - Einfache Änderung des Objektverhaltens zur Laufzeit
  - Andere Klassen sollen nicht davon beeinflusst werden

- **Beispiel:**
  - Das File-Objekt reagiert in Abhängigkeit vom Zustand

State

- **Lösung:**



State

Context:

- Verwaltet den aktuellen Zustand
- Definiert die clientseitige Schnittstelle
- Verwaltet die Zustandsklassen

State :

- Definiert eine Standardschnittstelle für einen Zustand
- Implementiert ggf. ein Standardverhalten

ConcreteState:

- Impelmentiert das zustandsbehaftete Verhalten

State

Bewertung:

Vorteile:

- komplexe und schwer zu lesende Bedingungsanweisungen vermieden werden können
- neue Zustände und neues Verhalten auf einfache Weise hinzugefügt werden.
- Die Wartbarkeit wird erhöht und Zustandsobjekte können wiederverwendet werden.

Nachteil:

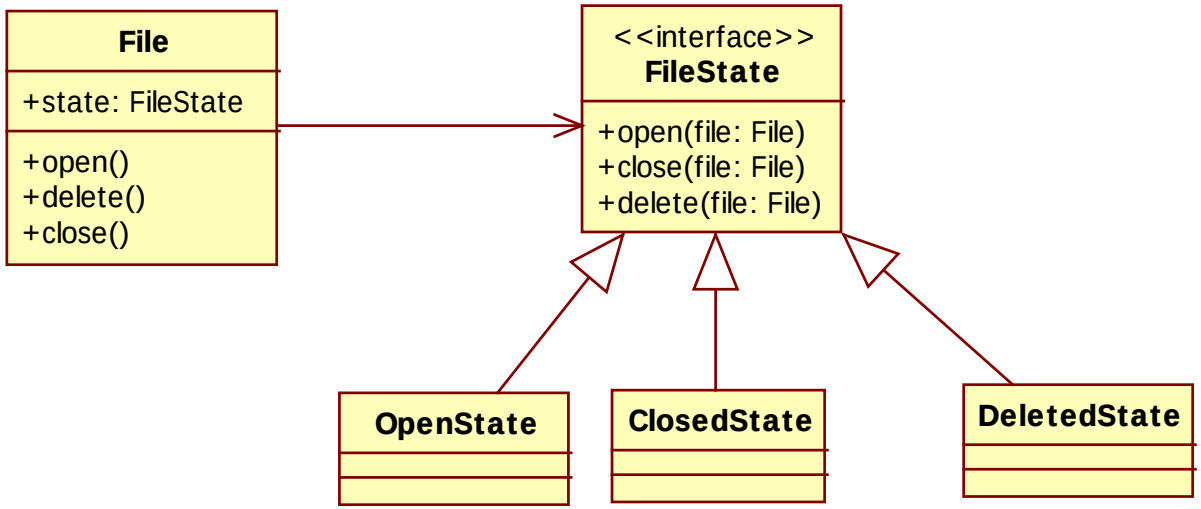
- Hoher Implementierungsaufwand
- Evtl. zu hoher Aufwand bei einfachen Zustandsmodellen

State

- **Verwendung (allgemein):**
  - Das Verhalten eines Objektes ist abhängig von dessen Zustand
  - Die „traditionalle“ Implementierung über switch-Anweisungen soll vermieden werden
  - Statt mehrere switch-Anweisungen soll das Zustandsbezogene Verhalten in eigene Klassen ausgelagert werden

State

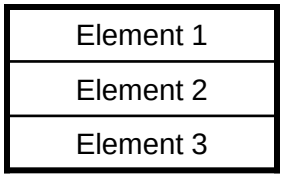
- **Beispiel: File**



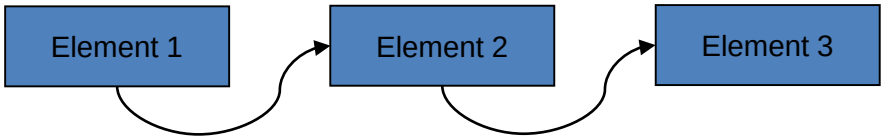
Iterator

- **Zweck**:
  - Durchlaufe alle Elemente einer Objektansammlung genau einmal

- **Beispiel:**
  - Iteration über Collections bzw. die Elemente eines abstrakten Datentyps (ADT) unabhängig von dessen Implementierung



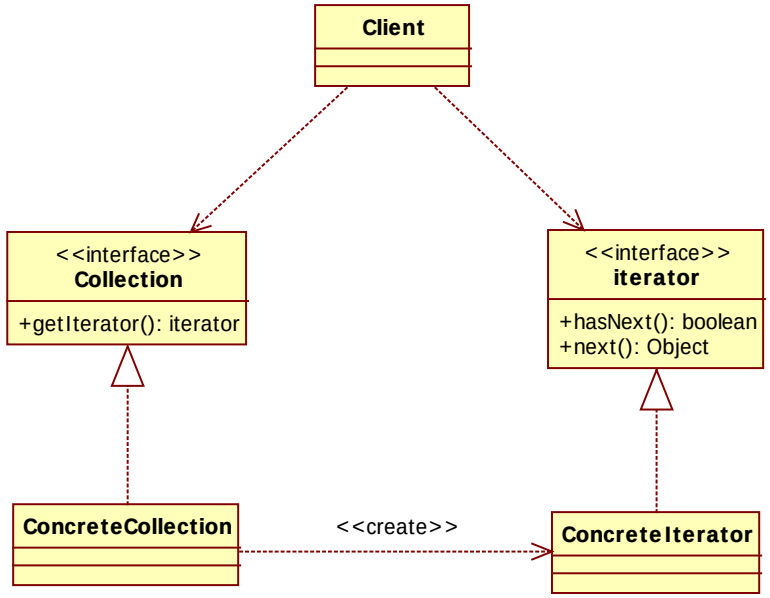
Implementierung
des ADT als Array



Implementierung des ADT als verkettete Liste

Iterator

- **Lösung:**



Iterator

Iterator:

- definiert die Schnittstelle zum Zugriff auf die Elemente und zum Traversieren der Collection

ConcreteIterator:

- implementiert die Iterator Schnittstelle
- Führt einen Positionszeiger auf das aktuelle Element

Collection:

- definiert die Schnittstelle zum Erzeugen eines Iterators

ConcreteCollection

- Implementiert die Schnittstelle der Collection

Iterator

Bewertung:

Vorteile:

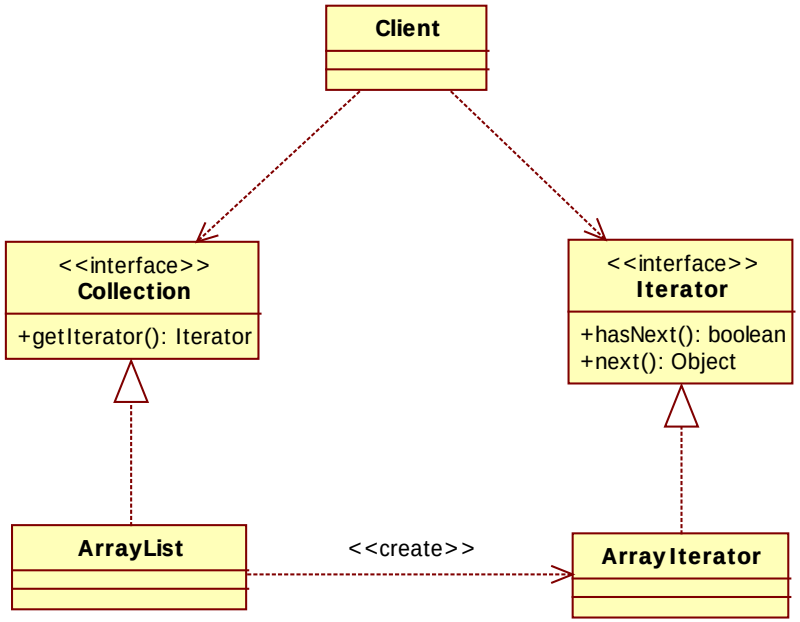
- Die Implementierung der zu Grunde liegenden Datenstruktur bleibt verborgen.

Nachteil:

- Je nach Variante der Implementierung können sich Nachteile durch erhöhte Laufzeit- und Speicherkosten ergeben.

Iterator

- **Beispiel: Java-Collections**



Iterator

- **Verwendung (allgemein):**
  - Zusammenfassung von Objekten innerhalb von Collections
  - Auf die Elemente solch einer Sammlung soll möglichst generisch und ohne Rücksicht auf die Implementierungsdetails zugegriffen werden können.