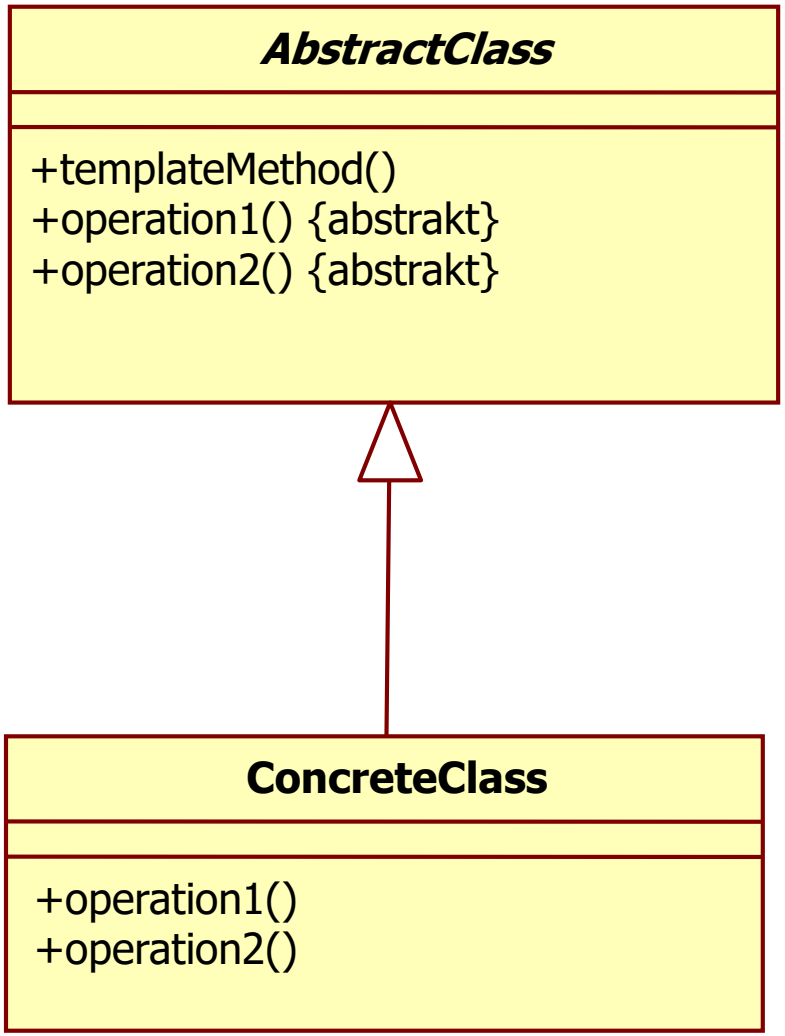
Funktionale Design-Pattern

Funktionale Design-Pattern
2 Kategorien

- **in OOS:**
  - einige OO-Design-Pattern kennengelernt
  - benutzen OO-Mechanismen: Vererbung, Überlagerung, …
- **hier:**
  - Funkt. Konzepte kennengelernt: Higher-Order, Pattern Matching, ..
  - Verwenden Funkt. Konzepte um:
    - OO-Design-Pattern weiter zu vereinfachen:
      - Template Method
      - Strategy
    - neue Funktionale Design-Pattern zu entwickeln:
      - Filter – Map – Reduce
      - Tail-Recursion
      - Function Builder/Factory

Funktionale Design-Pattern
OO -> Funktional: Template Method OO



Verarbeitung der abstrakten Methoden 1 & 2 in templateMethod()

Bsp.: in templateMethod() Addition der beiden Werte von operation1() und operation2()

Implementierung der Methoden 1 & 2 und Erben von templateMethod()

Funktionale Design-Pattern
OO -> Funktional: Template Method Funktional

- **Überführung in Funktionales Design**
  - templateMethod -> Higher-Order-Funktion mit Parameter
    - operation1
    - operation2
- **Bsp.** von voriger Folie

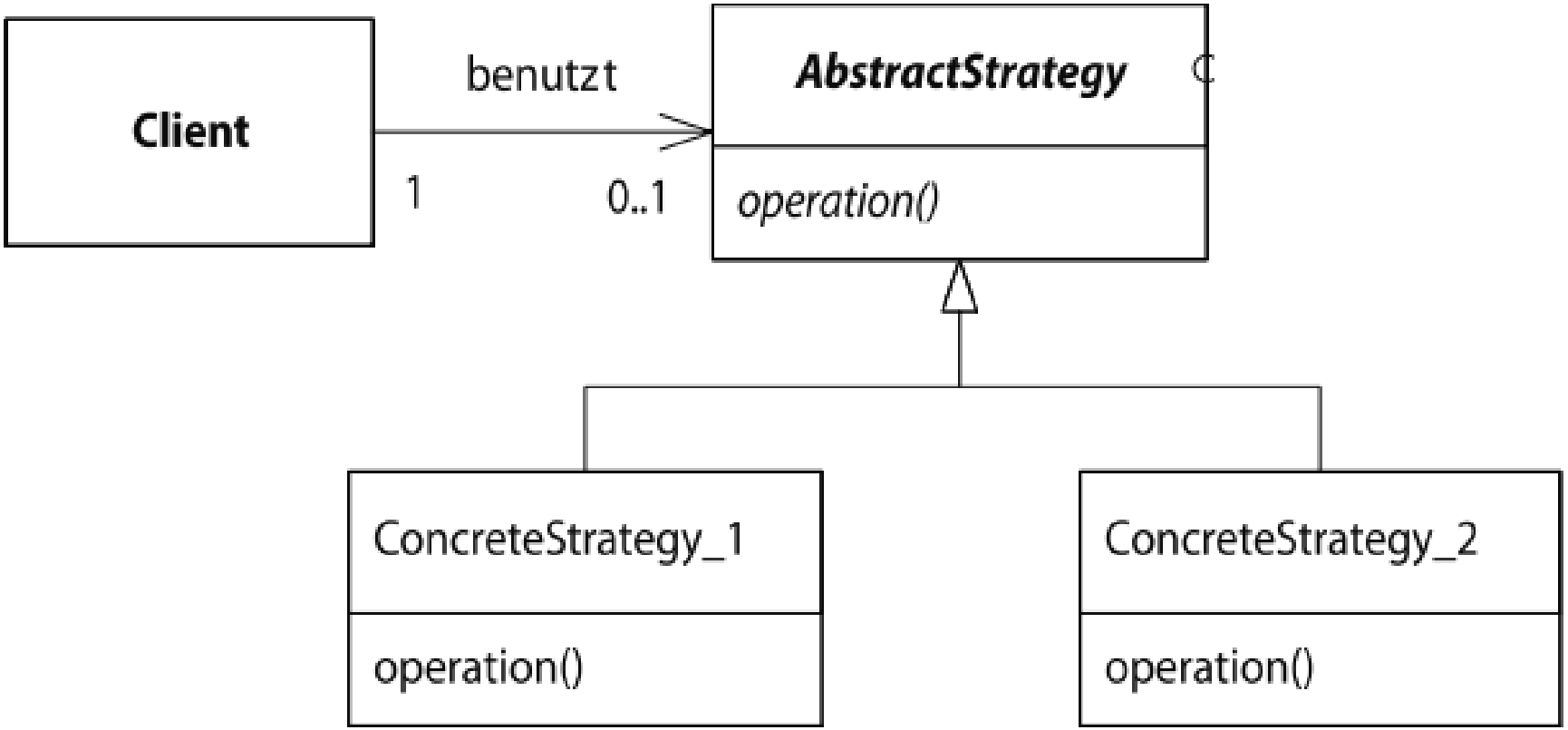
```
def templateMethod(input:Int,op1:Int=>Int,op2:Int=>Int) :
        Int =  op1(input) + op2(input)
```

- **Verwendung für square(x) = (x-1) * (x-1) + x + x – 1**

```
def square(input:Int) : Int =
        templateMethod(input,x=>(x-1)*(x-1),x=>x+x-1)
```

- **Vorteile:**
  - 2 Klassen weniger
  - keine Vererbung notwendig
  - kein dynamisches Binden

Funktionale Design-Pattern
OO -> Funktional: Strategy OO



- **Bsp.:** Berechne n. größtes Element einer Integerliste
- Client nutzt SortierFunktion der AbstractStrategy. Diese wird in den konkreten Klassen implementiert durch z.B. BubbleSort, QuickSort

Funktionale Design-Pattern
OO -> Funktional: Strategy Funktional

- **Überführung in Funktionales Design**
  - Implementierungsfunktion des Client

  -> Higher-Order-Funktion mit Strategy-Operation als Parameter
- **Bsp.** von voriger Folie

```
def nInListe(n:Int,sort:List[Int]=>List[Int],xs:List[Int])
 : Int = sort(xs) match {
     case y::ys => if (n==1) y else nInListe(n-1,sort,ys)
   }
```

- **Verwendung für Quicksort** (Foliensatz ScalaBesonderheiten)

```
nInListe(3,sortOOFun,2::4::1::Nil) -> 4
```

- **Vorteile:**
  - 2 Klassen weniger
  - keine Vererbung notwendig
  - kein dynamisches Binden

Funktionale Design-Pattern
Funktional: Filter – Map - Reduce

- **bei Listenbearbeitung häufig eins der drei Muster:**
  - Filter spezielle Elemente aus Liste : `filter`
  - Verknüpfe Listenelemente : `fold` (hier: **`reduce`**)
  - und wende eine Funktion auf alle Listenelemente an: `map`
- **werden oft verknüpft angewendet**
- **Bsp.:**
  - Berechne aus Liste von Preisen den Gesamtrabatt
  - Dabei erhält man einen Rabatt von 10% für
  - alle Preise ab 20€
- **Vorgehensweise:**
  - **filter:** alle Preise ab 20€
  - **map:** Berechne Rabatte
  - **fold (reduce):** Summiere die Rabatte

Funktionale Design-Pattern
Funktional: Tail-Rekursion

- **siehe:** Foliensatz FirstOrderProgrammierung
- **Rekursiver Aufruf muss letzte Aktion sein**
- **Dann:** jeweils gleicher Stackframe verwendbar
- **oder:** Umwandlung in Iteration (in Scala: @tailrec)
- **Umsetzung:**
  - gehe von iterativem Programm aus
  - lokalen Variablen -> Parameter
  - Variablenzuweisungen -> in Parametern ausführen
- **Bsp.:**

```
def fibTR(n: Int) : Int = fibHelp(n,0,1)
def fibHelp(n: Int, vorvor: Int, vor: Int) : Int
 =  if (n==0) vorvor
    else fibHelp(n-1,vor,vorvor+vor)
```

Funktionale Design-Pattern
Funktional: Function Builder/Factory

- **siehe:** Foliensatz CurryingPartial Application
- **Funktionen als Rückgabe** einer higher-order Funktion **möglich**
- **Ermöglicht:** Currying & Partial Application
- **Nutze dieses, um aus einer Funktion viele zu bauen**
- **Bsp.:** `filter(1::2::3::4::Nil,isGerade) -> 2::4::Nil`

```
def isGerade(n:Int): Boolean =          n%2==0
def isTeilbarDurch3(n:Int): Boolean =   n%3==0
```

- **Definiere Higher-Order Funktion**

```
def isTeilbarDurchK(k:Int)(n:Int): Boolean = n%k==0
```

- **Erzeuge obige Funktionen durch currying auf dieser**

```
def isGerade: Int => Boolean =    isTeilbarDurchK(2)
def isTeilbarDurch3 : Int => Boolean =
                                  isTeilbarDurchK(3)
```

- **Weitere Anwendung: Funktion, die aus Prädikat Gegenteil macht**

```
isGerade -> isUngerade, isKonsonant -> isVokal
```