# Übungsaufgabe

Unterschiede zwischen funktionaler und objektorientierter Programmierung: Zustand, Seiteneffekte, Abstraktionen.

| | |
|---|---|
| **Universität:** | Technische Universität Berlin |
| **Kurs/Modul:** | Programmieren II für Wirtschaftsinformatiker |
| **Erstellungsdatum:** | September 6, 2025 |

# Aufgabe 1: Unterschiede zwischen funktionaler und objektorientierter Programmierung: Zustand, Seiteneffekte, Abstraktionen

In dieser Aufgabe werden zentrale Unterschiede in Bezug auf Zustand, Seiteneffekte und Abstraktionen zwischen funktionaler Programmierung und objektorientierter Programmierung untersucht. Arbeiten Sie eigenständig, erläutern Sie die Konzepte und geben Sie kurze Beispiele bzw. Skizzen an.

a) Definieren Sie knapp die Begriffe Zustand und referenzielle Transparenz in der funktionalen Programmierung. Beschreiben Sie dazu das Gegenstück in der Objektorientierung. Warum sind diese Konzepte in der Praxis relevant?

b) Was versteht man unter Seiteneffekten? Geben Sie Beispiele für typische Seiteneffekte. Diskutieren Sie, wie funktionale Sprachen und OO-Entwürfe mit Seiteneffekten umgehen bzw. sie vermeiden bzw. kontrollieren.

c) Abstraktionen: Vergleichen Sie die Abstraktionsebenen in funktionaler Programmierung (Funktionen, Pure Functions, Higher-Order Functions) mit den Abstraktionen in der Objektorientierung (Klassen, Objekte, Komponenten). Welche Vorteile ergeben sich jeweils in Bezug auf Wiederverwendbarkeit, Wartbarkeit und Testbarkeit?

d) Geben Sie zwei kurze Beispiele (Pseudocode oder in Scala/Haskell) an, die

- eine rein funktionale Variante zeigen, die keinen Zustand ändert und keine Seiteneffekte hat, und
- eine OO-Variante zeigen, in der der Zustand in Klassen kapselt ist und Seiteneffekte auftreten.

Hinweise: Verwenden Sie eigenständige Bezeichnungen und verzichten Sie auf eine Lösung in dieser Aufgabe.

e) Welche Rolle spielen Nebenläufigkeit und Parallelität in Bezug auf Zustand und Seiteneffekte? Skizzieren Sie Grundideen, wie man in funktionalen vs. OO-Ansätzen sichere Nebenläufigkeit erreichen könnte.

# Aufgabe 2: Praxisbezug – Modellierung von Zustand, Seiteneffekten und Abstraktionen

Betrachten Sie typische Szenarien aus der Wirtschaftsinformatik. Diskutieren Sie, wie die genannten Konzepte in beiden Parademen modelliert werden könnten. Formulieren Sie klare Aufgabenstellungen bzw. Diskussionspunkte, ohne Lösungen anzugeben.

a) Szenario: Transaktionsverarbeitung in einem Finanzsystem. Welche Unterschiede ergeben sich in der Modellierung von Zustand, Seiteneffekten und Abstraktionen zwischen funktionaler und OO-Programmierung?

b) Skizzieren Sie zwei kurze Beschreibungen (keine Implementierung) zu folgenden Ansätzen:

> - Eine rein funktionale Variante, die Transaktionen modelliert, ohne globalen Zustand zu verändern. - Eine OO-Variante, in der Konten als Objekte mit internem Zustand modelliert sind.

c) Welche Herausforderungen ergeben sich bei Nebenläufigkeit in den beiden Ansätzen? Nennen Sie mögliche Muster oder Konzepte (ohne Lösung) zur Gewährleistung von Konsistenz und Korrektheit.

# Lösungen

# Lösung zu Aufgabe 1: Unterschiede zwischen funktionaler und objektorientierter Programmierung: Zustand, Seiteneffekte, Abstraktionen

a) Zustand und referenzielle Transparenz

- **Zustand** bezeichnet die Gesamtheit der zur Laufzeit gespeicherten Werte, die das Verhalten eines Programms beeinflussen können. In Sprachen mit mutablem Zustand ändern sich diese Werte im Verlauf der Ausführung.
- **Referenzielle Transparenz** bedeutet, dass ein Ausdruck bei gegebenen Eingaben immer denselben Ausgabewert liefert und keine Beobachtungen außerhalb des Ausdrucks (Zustandsveränderungen, I/O etc.) berücksichtigt werden müssen. Solche Funktionen sind rein mathematisch.
- **Gegenstück in der Objektorientierung (OO)**: Der Zustand wird typischerweise in Feldern/Attributen von Objekten verwaltet. Methoden können diesen Zustand verändern (Seiteneffekte). Dadurch hängt das Verhalten stärker vom internen Objektzustand ab, und Funktionsweise ist weniger referenztransparent.
- **Relevanz in der Praxis**: Reine Funktionen sind leichter zu testen, zu verstehen und parallel auszuführen. In OO-Designs erfordert der Umgang mit mutablem Zustand oft explizite Synchronisierung und sorgfältige Nebenläufigkeitskontrollen.

b) Seiteneffekte

- **Definition**: Ein Seiteneffekt liegt vor, wenn der Aufruf einer Funktion bzw. Methode nicht nur einen Wert erbringt, sondern auch außerhalb ihres Rumpfes etwas verändert (z. B. mutierte Variablen, I/O, Dateisystemzugriffe, Netzkommunikation).
- **Typische Beispiele**: Schreiben in Dateien, Verändern globaler Variablen, Ausgabe auf der Konsole, Modifikation eines Objektsatzes, der von anderen Teilen des Programms gelesen wird.
- **Umgang in funktionalen Sprachen**: Funktionssprachen vermeiden oder strikt kontrollieren, z. B. durch reine Funktionen und Monaden (IO-Monade, State-Monade) sowie durch explizite Effekte.
- **Umgang in OO-Entwürfen**: Seiteneffekte entstehen oft durch Mutationen des Objektzustands. Zur Kontrolle dienen Prinzipien wie Information Hiding, gezielte Nebenläufigkeitssteuerung, Immutable-Objekte an Stellen, wo möglich, sowie Muster wie Command- oder Observer-Pattern.

c) Abstraktionen

- **Funktionale Programmierung:** Abstraktionsebenen umfassen Funktionen, rein funktionale (Pure Functions) und Higher-Order Functions. Wichtige Konzepte: Funktionskomposition, Currying/Partial Application, Unveränderlichkeit (Immutability).

- **Objektorientierte Programmierung:** Abstraktionen basieren auf Klassen, Objekten, Komponenten, Schnittstellen (Polymorphie, Vererbung, Kapselung). Vorteile: klare Strukturierung von Domänenmodellen, weniges Rauschen durch Mangel an globalem Zustand, gute Wartbarkeit durch Modulgrenzen.
- **Vorteile für Wiederverwendbarkeit, Wartbarkeit und Testbarkeit:**
  - Funktional: einfache Wiederverwendung durch Composition, leichter Testbarkeit aufgrund reiner Funktionen.
  - OO: klare Domänenmodelle, bessere Abstraktion durch Interfaces/Subsysteme, oft bessere Trennung von Verantwortlichkeiten durch Klassenhierarchien.

d) Zwei kurze Beispiele (Pseudocode oder in Scala/Haskell)

- Rein funktionale Variante (keine Zustandsänderung, keine Seiteneffekte)

```
// Haskell (rein funktional)
sumList :: [Int] -> Int
sumList xs = foldl (+) 0 xs
```

- OO-Variante (Zustand kapselt in Klassen, Seiteneffekte auftreten)

```
// Scala-Variante (Zustand in Objekt)
class BankAccount(private var balance: Double) {
  def deposit(amount: Double): Unit = { balance += amount }
  def withdraw(amount: Double): Unit = { balance -= amount }
  def current: Double = balance
}
```

e) Rolle von Nebenläufigkeit und Parallelität

- **Funktionale Sprachen**: Durch referenzielle Transparenz lässt sich Neben- und Parallelität sicher realisieren, da keine versteckten Zustandveränderungen existieren. Typische Muster: MapReduce, data-parallel processing, Futures/Promises, bzw. in Scala korrespondierende Ansätze. Effekte werden kontrolliert (z. B. IO-Monaden, STM in manchen Sprachen).
- **Objektorientierte Entwürfe**: Nebenläufigkeit erfordert explizite Synchronisierung (Locks, volatile Felder, Atomics) oder das Muster des aktorischen Modells (z. B. Akka) zur verlässlichen Synchronisation und zum Verhindern von Race Conditions. Transaktionen, konsistente Zustandsübergänge über Objekte hinweg stellen oft zusätzliche Infrastruktur (Locks, Semaphoren, MVCC in Persistenzschichten) bereit.
- **Grundideen für sichere Nebenläufigkeit**: - In funktional orientierten Ansätzen: unveränderlicher Zustand, reine Funktionen, isolierte Effekte, stabiles Framework-Ökosystem für parallele Verarbeitung. - In OO-Ansätzen: isolierte Objekte, synchronisierte Zugriffe, Actor-/Message-Passing-Modelle, Transaktionskonzepte in Persistenzschichten.

# Lösung zu Aufgabe 2: Praxisbezug – Modellierung von Zustand, Seiteneffekten und Abstraktionen

a) Szenario: Transaktionsverarbeitung in einem Finanzsystem

- **Funktionale Modellierung**: Der Ledger besteht aus unveränderlichen Datenstrukturen (z. B. Map von Konten auf Salden). Transaktionen werden als Ereignisse beschrieben, und eine reine Funktion wendet eine Sequenz von Transaktionen auf den aktuellen Ledger an und erzeugt einen neuen Ledger. Nebenläufigkeit wird durch reinen Funktionszusammenhang, Event Sourcing oder MVCC in der Persistenzschicht umgesetzt. Keine gemeinsamen Mutationen im Hauptpfad; Synchronisation erfolgt über unveränderliche Daten und isolierte Operationen.
- **OO-Modellierung**: Konten werden als Objekte modelliert, die internen Zustand (saldo) kapseln. Transaktionen manipulieren direkt die Konten durch Methoden wie transfer, withdraw, deposit. Die Konsistenz wird durch Synchronisierung oder Transaktionsgrenzen sichergestellt. Mehrere Konten können in einer einzigen Operation betroffen sein, was zu requires atomare Absturzsicherheit führt (z. B. durch Transaktionsmanager oder verteilte Transaktionen).

b) Beschreibungen (keine Implementierung)

- Rein funktionale Variante: Transaktionen werden als unveränderliche Datenstrukturen beschrieben. Der Ledger ist eine unveränderliche Abbildung von Konten zu Salden. Eine Funktion applyTxn(s Ledger, txn) erzeugt einen neuen Ledger. Mehrere Transaktionen werden als eine Folge von Funktionsanwendungen modelliert (z. B. fold). Konsistenz wird durch Unveränderlichkeit, Event Sourcing oder MVCC in der Persistenzschicht gewährleistet.

- OO-Variante: Konten sind Objekte mit internem Zustand (Saldo). Eine Transaktion erfordert zwei Kontoobjekte (Abheben von einem Konto, Einzahlen auf ein anderes) und wird durch Methodenaufrufe umgesetzt. Die Transaktion muss atomar sein; Mechanismen wie Locks oder ein Transaktionsmanager sichern Konsistenz über mehrere Objekte hinweg.

c) Herausforderungen bei Nebenläufigkeit; Muster zur Gewährleistung von Konsistenz und Korrektheit

- **Funktionale Perspektive**:
  - Vorteile: Unveränderlichkeit erleichtert Parallelismus; geringes Risiko von Race Conditions.
  - Herausforderungen: Externe Side Effects (z. B. Datenbankzugriffe) müssen als Effects modelliert werden; Einsatz von Monaden/Effect Systems (IO, State) oder Event Sourcing, um Konsistenz sicherzustellen.
- **OO-Perspektive**:

  - Herausforderungen: Race Conditions, Deadlocks, inkonsistente Zwischenzustände, wenn Transaktionen Konten übergreifend verändert.
  - Muster/Lösungsansätze: Verwendung von @synchronized/Locks oder Atomic-References, Verwendung des Actor-Modells (z. B. Akka) für nicht-blockierende Kommunikation, Transaktions- bzw. ACID-Sicherheiten in Persistenzschichten, MVCC in Datenbanken, eventbasierte Replikation/Commit-Protokolle.

- **Allgemeine Muster**:
  - Transaktionsgrenzen festlegen; Atomarität über beteiligte Objekte hinweg sicherstellen.
  - Immutability dort einsetzen, wo möglich (z. B. Transfer als neue State-Instanzen statt direkter Mutationen).
  - Event Sourcing und Logging zur Nachverfolgbarkeit und Wiederherstellbarkeit von Konsistenzzuständen.