Tag 1

Imperativ vs. Deklarativ Programmieren:

Imperativ:

Befehlsorientiert

Beschreibt genaue Schritte zur Ausführung

Kontrollstrukturen wie Schleifen und Bedingungen werden verwendet

Fokus auf dem "Wie" der Aufgabenlösung

Beispiele: C, C++, Java

Deklarativ:

Beschreibt das "Was" der Aufgabenstellung

Definiert das gewünschte Ergebnis, nicht den genauen Ablauf

Weniger Kontrollstrukturen, mehr Ausdrücke und Deklarationen

Abstrahiert von Implementierungsdetails

Beispiele: SQL, HTML, CSS, Prolog

Fachbegriffe

Call by reference:

- Parameter als Referenz übergeben
- Änderungen wirken sich außerhalb der Funktion aus
- Potenzielle Seiteneffekte

Call by value:

- Parameter als Kopie übergeben
- Änderungen haben keine Auswirkungen außerhalb der Funktion
- Bevorzugt in funktionaler Programmierung

Eager evaluation:

- Sofortige Auswertung von Ausdrücken während der Definition
- Gegenteil von lazy evaluation

Impure functions:

- Verletzen Regeln für pure functions

- Haben Seiteneffekte oder mehrere Rückgabewerte

Lazy evaluation:

- Ausdrücke werden erst bei Bedarf ausgewertet
- Kann Performance verbessern

Pure functions:

- Erfüllen drei Regeln: Ein Rückgabewert, abhängig von Parametern, keine Veränderung von Daten

Referenzielle Transparenz:

- Funktion kann durch Rückgabewert ersetzt werden, ohne Verhaltensänderung

Rekursion:

- Funktion ruft sich selbst auf

Benötigt Abbruchkriterium, um Endlosschleifen zu vermeiden

Tag 2

Unterschied zwischen imperativer und deklarativer Programmierung:

Imperativ: Befehlsorientiert, beschreibt genaue Schritte zur Ausführung

Deklarativ: Beschreibt das "Was" der Aufgabenstellung, nicht den genauen Ablauf

Transfer von Anforderungen:

Imperativ: Anforderungen in Schritte umsetzen

Deklarativ: Anforderungen als gewünschtes Ergebnis definieren

Anforderungen als Funktionen:

Identifizierung von Anforderungen als abstrakte Funktionen

Umsetzung der Anforderungen als Funktionsaufrufe

Ableitung des "Was":

Analyse von Beschreibungen auf gewünschte Ergebnisse

Extraktion von Hauptzielen und gewünschten Zuständen

Regeln pureFunctions:

Gibt nur einen Wert zurück (Wenn die Funktion einen Array zurückgibt, gilt das dennoch als ein Wert - nämlich ein Array)

Berechnet den Rückgabewert nur aufgrund der ihr übergebenen Parameter

Verändert keine existierenden Werte (neue in der Funktion selber definierte Werte dürfen verändert werden)

Vorteile pureFunctions:

- **Keine Seiteneffekte:** Bei der Verwendung einer pure function treten keine Seiteneffekte auf. Sie könnten - ohne die Programmlogik zu verändern - auch den Rückgabewert anstelle des Funktionsaufrufs in den Code packen (referenziell Transparent). D. h. der Funktionsaufruf löst nichts Unerwartetes bzw. keine Statusänderung aus.

- **Einfach zu testen:** Weil die pure function nur von den Parametern abhängt, reicht der Funktionsaufruf, um diese zu testen. Bei Funktionen die auf globale Werte zugreifen und diese verwenden müssen diese globalen Werte für einen Test der Funktion zuerst noch mit sinnvollen Werten instanziiert werden.
- **Code wird klarer:** Weil keine Seiteneffekte und globalen Werte für das Codeverständnis wichtig sind, wird der Code viel einfacher zu lesen und ist sehr viel verständlicher.

Nachteile pureFunctions:

- **Performance:** Die vielen Kopien der Daten brauchen einerseits Speicherplatz andererseits aber auch Zeit für den Kopiervorgang. Das führt dazu, dass die Ausführung des Programmcodes in der Regel weniger performant sein wird. Um dem entgegenzuwirken, gibt es das Konzept der lazy evaluation, wo Ausdrücke erst dann ausgewertet werden, wenn diese auch tatsächlich benötigt werden.

- **Rekursion:** Rekursive Funktionen können nicht ganz so intuitiv zu verstehen sein. Wenn der Punkt keine existierenden Werte zu veränder mit rekursiven Funktionen gelöst wird, kann das die Verständlichkeit des Codes auch wieder etwas erschweren. Zudem kann eine sehr tiefe Rekursion dazu führen, dass der Verbrauch des Zwischenspeichers massiv anwächst und erst wieder freigegeben werden kann, wenn das Abbruchkriterium für die rekursive Funktion erreicht ist. Rekursion sollte entsprechend nur dort eingesetzt werden, wo es auch wirklich sinnvoll ist (beispielsweise dort wo Sie in der klassischen Programmierung mit while- oder for-Loops durch eine Liste von Elementen iterieren).