DHBW
Duale Hochschule
Baden-Württemberg

Halöle

Intro

Side Effects

TODO: Sieht irgendwie nach zu wenig Text aus

Side Effects sind alle Dinge die eine Funktion macht, die nicht Teil ihres Outputs sind. Das Ändern von globalen Variablen, das Modifizieren von Objekten, das Schreiben in Dateien oder das Aufbauen von Netzwerkverbindungen sind alles Beispiele für Side Effects. Der Nachteil von Side Effects ist, dass sie den Zustand des Programms verändern können, was zu unerwartetem Verhalten führen kann. Dies erschwert das Testen und Debuggen von Code, da die Funktion nicht mehr isoliert betrachtet werden kann. Außerdem leidet die Transparenz des Codes, da der Leser nicht sofort erkennen kann, welche Teile des Codes den Zustand verändern. Der Leser muss den gesamten Kontext verstehen, um die Auswirkungen einer Funktion vollständig zu begreifen. Ein Beispiel für Side Effects ist eine Funktion, die eine globale Variable ändert:

```
counter = 0
def increment():
    global counter
    counter += 1
increment()
print(counter)  # Ausgabe: 1
```

Funktionale Programmierung hingegen strebt danach, Side Effects zu minimieren oder klar vom Rest des Codes zu trennen, um die Vorteile reiner Funktionen zu nutzen. ***Pure Functions*** sind Funktionen, die für dieselben Eingaben immer dieselben Ausgaben liefern und keine Side Effects haben:

$$f : A \to B, x \mapsto y$$

Die Funktion erhält als Parameter einen Wert x vom Typ A und gibt einen Wert y vom Typ B zurück. Dabei wird x selbst nicht verändert, und es gibt keine anderen Auswirkungen auf den Zustand des Programms.

Dies erleichtert das Verständnis, Testen und die Wiederverwendbarkeit von Code erheblich. Ein Beispiel für eine reine Funktion ist:

```
def increment(x):
    return x + 1

counter = increment(0)
print(counter)  # Ausgabe: 1
```

In diesem Beispiel hat die Funktion `increment` keine Side Effects, da sie nur den Wert ihrer Eingabe um 1 erhöht, ohne den Zustand des Programms zu verändern. Außerdem erkennt man sofort, dass `increment(0)` immer `1` zurückgibt, unabhängig vom Kontext.

Jedoch kommt man in der Praxis nicht komplett ohne Side Effects aus, da Programme mit der Außenwelt interagieren müssen um einen Nutzen zu haben (z.B. Benutzereingaben, Datenbanken, Netzwerke). Es ist daher notwendig, Side Effects zu verwalten und zu kontrollieren, um die Vorteile der funktionalen Programmierung zu nutzen, ohne die Notwendigkeit von Side Effects zu vernachlässigen. Es ist außerdem möglich, Side Effects an den Rand des Programms zu verlagern, sodass der Großteil des Codes rein und frei von Side Effects bleibt. Dies kann beispielsweise duch

den Einsatz von Datenbanken ermöglicht werden, indem der Zustand in der Datenbank gespeichert wird und die Hauptlogik des Programms rein bleibt.

Transparenz

TODO: man muss halt iwi interne Implementierung kennen weil SE nicht klar aldf

(wir müssen es schaffen nen convincing case vs. OOP zu machen)

Beim ***OOP*** ist Transparenz im Code oft eingeschränkt, da Methoden den internen Zustand von Objekten verändern zu können. Man muss den die gesamte Klasse und ihre Zustände verstehen, um die Auswirkungen einer einzelnen Methode zu verstehen. Das erschwert das Lesen und Warten, des Codes, da die Auswirkungen einer Methode nicht isoliert betrachtet werden können. In der ***funktionalen Programmierung*** hingegen sind Funktionen reine Funktionen ohne Side Effects. Dadurch können sie isoliert betrachtet und verstanden werden, was die Transparenz des Codes erhöht. Beim Beispiel **TODO: Beispiel referenzieren** muss man wissen, wechen Zustand die Variable `counter`hat, um zu verstehen, was das Ergebnis von `increment()` ist. Man kann erahnen, dass `increment()` den Zustand von Counter erhöht, aber man weiß nicht das konkrete Ergebnis ohne den aktuellen Zustand zu kennen. Hier ist das noch kein großes Problem, aber bei komplexeren Methoden und Klassen wird es schnell unübersichtlich, sodass nicht direkt klar ist, was eine Methode genau macht.

In der ***FP*** hingegen ist die Funktion `increment(x)` aus **TODO: Beispiel referenzieren** eine reine Funktion, die für eine Eingabe immer dieselbe Ausgabe liefert, ohne den Zustand des Programms zu verändern. Beim Aufruf von `increment(0)` sieht man den Eingabewert und kann damit besser nachvollziehen, woher die Eingabe kommt und was die Funktion macht. Dadurch wird der Code transparenter und leichter verständlich. Da reine Funktionen außerdem isoliert betrachtet werden können, ist es einfacher, sie zu verstehen, da man nicht den gesamten Kontext der Klasse oder des Objekts kennen muss um zu verstehen, was die Funktion macht.

Concurrency

TODO: Maybe Beispiele hinzufügen?

TODO: Under construction, die Beispiele werden noch angepasst und an die richtige Stelle gepackt

```
from threading import Thread
N = 100_000

class Acc:
    def __init__(self):
        self.sum = 0  # geteilter, veränderlicher Zustand

    def add(self, v):
        # ohne Lock: race-conditions möglich
        self.sum += v

def worker(acc, items):
    for i in items:
        acc.add(i * i)

if __name__ == "__main__":
```

```
    acc = Acc()
    items = list(range(N))
    half = len(items) // 2
    t1 = Thread(target=worker, args=(acc, items[:half]))
    t2 = Thread(target=worker, args=(acc, items[half:]))
    t1.start(); t2.start()
    t1.join(); t2.join()
    print("OOP (ohne Lock) -> sum =", acc.sum)
    # Erwartet: N*(N-1)*(2N-1)/6; ohne Lock meist kleiner/falsch
```

Bla Bla

```
from concurrent.futures import ThreadPoolExecutor
N = 100_000

def square(x):
    return x * x  # reine Funktion, keine Seiteneffekte

if __name__ == "__main__":
    items = list(range(N))
    with ThreadPoolExecutor(max_workers=4) as ex:
        results = list(ex.map(square, items))  # map-ähnliche FP-Semantik
    total = sum(results)  # Aggregation erfolgt danach (im Hauptthread)
    print("FP (no shared mutation) -> sum =", total)
```

Unter ***Concurrency*** versteht man die Fähigkeit eines Programms, mehrere Aufgaben gleichzeitig auszuführen. Ein Problem im ***OOP*** ist der Umgang mit ***Shared Mutable State***, also mit gemeinsam genutztem, veränderbarem Zustand. Wenn mehrere Threads gleichzeitig auf denselben Zustand zugreifen und diesen ändern, kann es zu Inkonsistenzen kommen. Das führt zu unerwartetem Verhalten, da die Threads sich gegenseitig beeinflussen. In ***OOP*** wird dieses Problem mit ***Locks*** gelöst, die sicherstellen, dass nur ein Thread gleichzeitig auf den Zustand zugreifen kann. Dies kann jedoch zu ***Deadlocks*** führen, bei denen zwei oder mehr Threads sich gegenseitig blockieren und nicht weiterarbeiten können. In der ***funktionalen Programmierung*** hingegen sind Daten ***immutable***. Ein Thread in der ***funktionalen Programmierung*** bekommt also keinen globalen Zustand, den er ändern kann, sondern eine Kopie des Zustands. Wenn ein Thread eine Änderung vornehmen möchte, dann erstellt er eine neue Version des Zustands, anstatt den ursprünglichen Zustand zu ändern. Damit können Threads nicht mehr konkurrieren und sich gegenseitig beeinflussen. In der ***funktionalen Programmierung*** wird das Problem der ***Concurrency*** also vermieden indem es keine gemeinsamen Zustände gibt.

Fundamentale Konzepte des FP

(es wäre schon witzig über Kategorien, Funktorialität und Monoiden zu yappen)

Notation

Im Folgenden werden wir Signaturen von Funktionen basierend auf ihren Datentypen in folgender Form schreiben:

$$x \to y$$

Hier ist x ein Vektor aus Parametern, und y der Rückgabewert der Funktion.

Pure Functions

äh ich glaub Maxim du führst den Begriff einfach bei Side effects ein und gut ist **TODO: Okidoki**

Higher-Order Functions

Als "Higher-Order Function" wird jede Funktion bezeichnet, die entweder durch eine andere Funktion parameterisiert wird, oder eine Funktion als Rückgabewert besitzt. Durch die Pythons dynamic geht dies ohne weiteres:

```
def apply_f_to_x(x: int, f: Callable) -> int:
  return f(x)
```

Dies ist ein Beispiel für den Syntax einer ***HoF*** in Python. Genutzt werden kann diese, indem man `apply_f_to_x` den Bezeichner einer anderen Funktion übergibt:

```
def square(x: int) -> int:
  return x ** 2

assert apply_f_to_x(4, square) == 16
```

Anonymous Functions

Besonders für kleine Funktionen, wie `square` im vorherigen Beispiel, kann es schnell verbos und unleserlich werden, jede dieser Funktionen seperat mit Namen zu initialisieren. In den meisten Sprachen gibt es deshalb einen Weg, Funktionen ohne Namen zu initialisieren, um sie direkt an Higher-Order Funktionen zu übergeben. In Python geschieht dies durch den `lambda` syntax:

```
lambda arg1 [, arg2, arg3, ...]: <result>
```

Das obige Beispiel kann demnach bedeutend kompakter geschrieben werden, ohne die Funktion `square` seperat zu deklarieren:

```
apply_f_to_x(4, lambda x: x ** 2)
```

Currying

Wie im vorherigen Kapitel erwähnt, können ***Higher-Order Functions*** auch Funktionen zurückgeben. Ein Anwendungsfall für dieses Pattern ist eine Art "Konstruktor" für Funktionen. Dies lässt sich gut aufzeigen am Beispiel der vorher eingeführten `square` Funktion: Es soll nun nicht nur quadriert werden, sondern der Exponent soll konfiguerbar sein. Die triviale Lösung hierfür ist eine zweistellige Funktion `(int, int) -> int`, wo der Exponent ein weiterer Parameter ist:

```
def power(base: int, exponent: int) -> int:
    return base ** exponent
```

Wollen wir nun eine Funktion mit dem selben Exponenten häufiger verwenden, können wir die Funktion `power` auch interpretieren als eine Funktion `int -> (int -> int)`, die den Exponenten als Parameter nimmt, und eine Funktion zurückgibt, welche das Potenzieren zu diesem Exponenten durchführt:

```
def c_power(exponent: int) -> Callable[[int], int]:
    return lambda base: base ** exponent
```

TODO: Hier fehlt glaub ich eine eckige Klammer bei Callable

Wir können `c_power` nun nutzen, um mehrere Exponentialfunktionen zu erstellen:

```
square = c_power(2)
cube = c_power(3)
the_answer = c_power(42)
```

Diese Reinterpretation einer Funktion mit mehreren Parametern als eine ***Higher-Order Function*** nennt sich ***Currying***. Zu bemerken ist, dass die zurückgegebene Funktion den Kontext `exponent` beibehält, obwohl sie den Scope der Funktion `c_power` verlässt. Sie "captured" die Variable `exponent`. Capturing ist ein Weg, wie (immutable) State zwischen Funktionen weitergereicht werden kann.
TODO: irgendwas zitieren für den Bullshit den ich da labere (should be like 90% correct)

Monaden

Das Pattern der Monade ist der Weg, wie deterministisch mit Seiteneffekten umgegangen werden kann. Monaden an sich sind ein Konzept aus der Kategorientheorie und entsprechend tief theoretisch verwurzelt. Wir wollen Monaden aber nur aus der Perspektive eines Software-Engineers betrachten, und werden deshalb theoretische Grundlagen auslassen. **TODO: unless we don't still gotta decide**

Eine Monade kann definiert werden als ein Parameterisierter Datentyp `M<T>`, der Methoden mit den Folgenen Signaturen bereitstellt:

1. `unit: T -> M<T>`
2. `bind: (M<A>, A -> M<B>) -> M<B>`

Damit `M` tatsächlich eine Monade ist, muss `unit` als Neutrales Element bezüglich der `bind` Operation agieren, und die Operation `bind` assoziativ sein.

Die Rolle der Methoden `unit` und `bind` können gut anhand des Beispiels der sogenannten "Maybe Monade" demonstriert werden. Diese abstrahiert den Side-Effect der möglichen Nicht-Existenz des enkapsulierten Wertes. Folgendes ist eine beispielhafte Implementierung der Maybe Monade in Python[1]:

```
T, S = TypeVar("T"), TypeVar("S")
class Maybe(Generic[T]):
    value: T

    def __init__(self, value: T):
```

[1] Anzumerken ist, dass sich die Mächtigkeit der Struktur besser aufzeigen ließe in einer Sprache, die Algebraische Summentypen unterstützt. Da Python dies nicht tut, nutzt unsere Implementierung weiterhin das prozedurale null-pattern (`None`) zur Repräsentation eines nicht existierenden Wertes.

```
        self.value = value

    @classmethod
    def unit(cls, value: T) -> "Maybe[T]":
        return cls(value)

    def bind(self, f: Callable[[T], "Maybe[S]"]) -> "Maybe[S]":
        if self.value is None:
            return Maybe(None)
        return f(self.value)
```

Diese Klasse implementiert beide Methoden einer Monade. Die Implementierung von `bind` als Methode eines Objektes ermöglicht die Nutzung der Maybe Monade durch das aneinander-ketten von `bind` aufrufen wie folgt:

```
val = 4
result = Maybe.unit(val)
    .bind(lambda x: Maybe(x - 2))
    .bind(lambda x: Maybe(str(x)))
assert result.value == "2"
```

Diese Kette an Operationen verändert erst einen Integer, und konvertiert ihn dann in einen String. Diese Aufgabe ist trvial genug, dass auch ein rein prozedureller Ansatz ohne unvorhergesehene Fehler durchlaufen könnte. Dies ändert sich allerdings, wenn man die Aufgabe umdreht: Es soll zuerst ein String von stdout eingelesen werden, dann in einen Integer konvertiert und schlussendlich verarbeitet werden.

```
result = Maybe.unit(input(""))
    .bind(lambda s: Maybe(int(s) if s.isdigit() else Maybe(None)))
    .bind(lambda x: x - 2)
```

Gibt der Nutzer eine valide Zahl ein (dies wird überprüft durch Pythons eingebaute Funktion `str.isdigit`), enthält `result.value` das korrekte Ergebnis als Integer. Tut der Nutzer dies allerdings nicht, ist der Wert von `result.value` `None`. In diesem Fall könnte man beispielsweise dem Benutzer eine Fehlermeldung anzeigen. Die hier gezeigte Implementierung ist der Übersichtlichkeit halber primitiv gehalten - in einer tatsächlichen Codebase sollte die Klasse weitere API Methoden enthalten, um Entwicklern eine sinnvolle Nutzung der `Maybe` Klasse mit semantischer Relevanz zu ermöglichen.

Monaden in der Praxis

Monadische Strukturen finden sich in beinahe allen modernen Programmiersprachen (außer halt Python lol). Die Maybe Monade beispielsweise manifestiert sich in Java als die Klasse `Optional`, und in Rust als der `Option` Datentyp. Ein weiteres Prominentes beispiel ist die sogenannte “IO Monade”, die in JavaScript als die Klasse `Promise`, und in Java und Rust als Klasse bzw. der Datentyp `Future` realisiert ist. Die IO Monade enkapsuliert den Side-Effekt, dass ein Wert möglicherweise erst in der Zukunft existiert. Sie findet häufig Anwendung in Web Applikationen, wo Daten über ein Netzwerk geladen werden müssen.

Ebenfalls ein prominentes Beispiel für Monaden, die häufig Anwendung finden ist die List Monade. Sie enkapsuliert den Side-Effekt, dass dieselbe Operation auf mehreren Elementen gleichzeitig ausgeführt werden soll. Die List Monade findet sich beispielsweise in Java in Form der Streams API

und in Rust in Form der Iterator API. In JavaScript ist bereits der Array Datentyp an sich eine Monade. oke keinen bock mal schauen das ding is für list monads müsste man fmap einführen wofür man die Funktorialität von Monaden definieren muss und dann muss man über das ganze theoretische Zeugs schreiben ich hasse wissenschaftliches Schreiben

Beispiel

- ein Beispiel von Kapitel 2 aufgreifen und mit FP fixen
- idealerweise auf alle Konzepte eingehen

ggf. Comparison mit anderen Paradigmen (OOP & Procedural)

Conclusion

Bibliography