Funktionale Softwarearchitektur

Michael Sperber, Active Group GmbH; Peter Thiemann, Universität Freiburg

Abstract

Der Entwurf von nachhaltigen Softwarearchitekturen ist eine Herausforderung: Mit der Größe steigt in vielen klassisch objektorientierten Softwareprojekten die Komplexität überproportional an. Durch viel Disziplin und regelmäßige Refaktorisierungen lässt sich das Problem eine Weile in Schach halten, aber die wechselseitigen Abhängigkeiten und komplexen Abläufe von Zustandsveränderungen nehmen mit der Zeit trotzdem zu. Die funktionale Softwarearchitektur geht an die Strukturierung großer Systeme anders heran als objektorientierte Ansätze und vermeidet so viele Quellen von Komplexität und Wechselwirkungen im System.

Aspekte funktionaler Softwarearchitektur

"Funktionale Softwarearchitektur" steht für das Ergebnis eines Softwareentwurfs mit den Mitteln der funktionalen Programmierung. Sie zeichnet sich unter anderem durch folgende Aspekte aus:

- An die Stelle des Objekts mit gekapseltem Zustand tritt die *Funktion*, die auf *unveränderlichen Daten* arbeitet.
- Funktionale Sprachen (ob statisch oder dynamisch) erlauben ein von *Typen* getriebenes, systematisches Design von Datenmodellen und Funktionen.
- Statt starrer hierarchischen Strukturen entstehen flexible, in die funktionale Programmiersprache *eingebettete domänenspezifische Sprachen*.

Wir konzentrieren uns in diesem Artikel auf den ersten Punkt, also den Umgang mit Funktionen und unveränderlichen Daten. Dabei werden wir auch die Rolle von Typen beleuchten.

Der Code zu diesem Artikel ist in einem Github-Repositorium abrufbar: https://github.com/funktionale-programmierung/hearts/

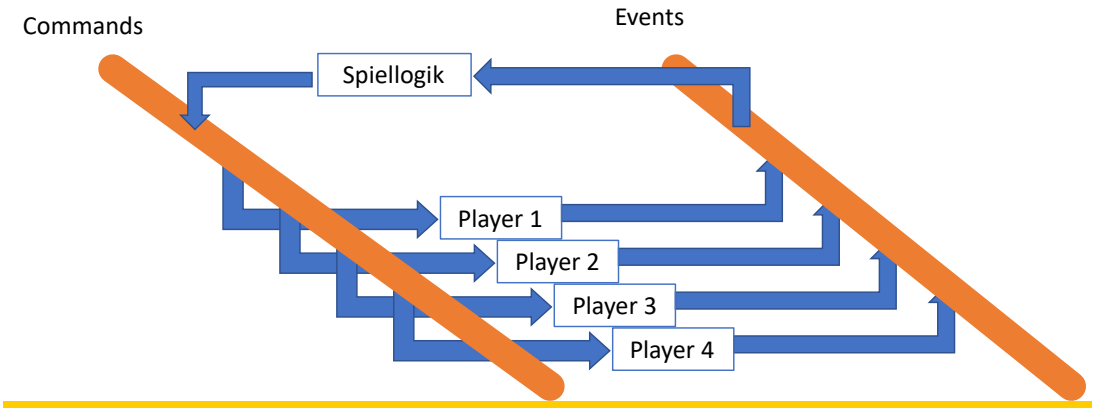
Funktionale Programmiersprachen

Funktionale Softwarearchitektur ist in (fast) jeder Programmiersprache möglich, aber in einer funktionalen Sprache wie Haskell, OCaml, Clojure, Scala, Elixir, Erlang, F# oder Swift ist diese Herangehensweise besonders natürlich. Funktionale Softwarearchitektur wird in der Regel als Code ausgedrückt, also nicht in Form von Diagrammen. Entsprechend benutzen wir für die Beispiele in diesem Artikel die funktionale Sprache Haskell [1], die besonders kurze und elegante Programme ermöglicht. Keine Sorge: Wir erläutern den Code, so dass er auch ohne Vorkenntnisse in Haskell lesbar ist. Wer dadurch auf Haskell neugierig geworden ist, kann sich eine Einführung in funktionale Programmierung [2], ein Buch zu Haskell [3] oder einen Onlinekurs [4] zu Gemüte führen.

Überblick

Wir erklären den Entwurf einer funktionalen Softwarearchitektur anhand des Kartenspiel *Hearts* [5], von dem wir nur die wichtigsten Teile umsetzen.

Hearts wird zu Viert gespielt. In jeder Runde eröffnet eine Spielerin, indem sie eine Karte ausspielt. (Zu Beginn des Spiels muss das die Kreuz Zwei sein.) Die nächste Spielerin muss, wenn möglich, eine Karte mit der gleichen Farbe wie die Eröffnungskarte ausspielen. Anderenfalls darf sie eine beliebige Karte abwerfen. Haben alle Spielerinnen eine Karte ausgespielt, muss die Spielerin den Stich einziehen, deren Karte die gleiche Farbe wie die Eröffnungskarte hat sowie den höchsten Wert. Ziel ist, mit den eingezogenen Karten einen möglichst geringen Punktestand zu erreichen. Dabei zählt die Pik Dame 13 Punkte und jede Herzkarte einen Punkt; alle weiteren Karten 0 Punkte.



Gameplay.pdf

Abb. 1: Spielablauf durch Commands und Events

Modellierung des Spielablaufs

Als Basis des Entwurfs verwenden wir ein klassisches taktisches Entwurfsmuster aus dem *Domain-Driven Design* [6] (DDD) und modellieren das Kartenspiel auf der Basis von *domain events*. Die Events repräsentieren jedes Ereignis, das im Spielverlauf passiert ist - die Commands repräsentieren Wünsche der Beteiligten, dass etwas passiert. Die Architektur ist so offen für spätere Umstellung auf Client-Server-Betrieb, Mikroservices oder Event-Sourcing.

Abbildung 1 zeigt den Ablauf: Jede Spielerin nimmt Events entgegen - was im Spiel gerade passiert ist - und generiert dafür Commands, die Spielzüge repräsentieren. Die Spiellogik-Komponente nimmt die Commands entgegen, überprüft sie auf Korrektheit (War die Spielerin überhaupt dran? War der Spielzug regelkonform?) und generiert ihrerseits daraus wieder Events. Das Entwurfsmuster ist das gleiche wie in “objektorientiertem DDD”, aber die Umsetzung unterscheidet sich durch die Verwendung von unveränderlichen Daten und Funktionen.

Im Beispiel stellen wir die Kommunikation zwischen den einzelnen Komponenten der Architektur direkt mit Funktionsaufrufen her, aber auch andere Mechanismen - nebenläufige Prozesse oder Mikroservices - sind möglich.

Programmieren mit unveränderlichen Daten

Eine Vorbemerkung: “Unveränderliche Daten” bedeutet, dass es keine Zuweisungen gibt, die Attribute von Objekten verändern können. Wenn Veränderung modelliert werden soll, so generiert ein funktionales Programm neue Objekte. Diese Vorgehensweise bietet enorme Vorteile:

- Es gibt niemals Probleme mit verdeckten Zustandsänderungen durch Methodenaufrufe oder nebenläufige Prozesse.
- Es gibt keine inkonsistenten Zwischenzustände, wenn ein Programm erst das eine Feld, dann das nächste etc. setzt.
- Das Programm kann problemlos durch ein Gedächtnis erweitert werden, das zum Beispiel zu früheren Spielständen zurückkehrt, wenn eine Spielerin ihren Zug zurücknimmt.
- Es gibt keine verstecken Abhängigkeiten durch die Kommunikation von Zustand hinter den Kulissen.

Aus den gleichen Gründen ist in Java das Programmieren mit Value-Objekten oft nützlich.

Karten modellieren

Die konkrete Modellierung beginnt mit den Spielkarten. Der folgende Datentyp *Card* ist ein Record-Typ (analog zu einem POJO in Java) und legt damit fest, dass eine Karte eine Farbe (“suit”) und einen Wert (“rank”) hat.

```
data Card = Card { suit :: Suit, rank :: Rank }
```

Weiter werden noch Definitionen von Suit und Rank benötigt:

```
data Suit = Diamonds | Clubs | Spades | Hearts

data Rank = Numeric Integer | Jack | Queen | King | Ace
```

Hier handelt es sich um Aufzählungen (vergleichbar mit enum in Java) - das | steht für “Oder”, entsprechend steht dort: Ein Suit ist Diamonds oder Clubs oder Spades oder Hearts. Bei *Rank* ist es ähnlich - zusätzlich hat eine der Alternativen, *Numeric*, ein Feld vom Typ *Integer*, das den Wert einer Zahlenspielkarte angibt.

Hier ist die Definition der Kreuz Zwei auf Basis dieser Datentypdefinition in Form einer Gleichung:

```
twoOfClubs = Card Clubs (Numeric 2)
```

Das Beispiel zeigt, dass Card als *Konstruktorfunktion* agiert. Außerdem auffällig: In Haskell werden Funktionsaufrufe ohne Klammern und Komma geschrieben, Klammern dienen nur zum Gruppieren.

Die Deklaration von Cards definiert auch die “Getter-Funktionen” *suit* und *rank*, die wie Funktionen verwendet werden.

Kartenspiele und Hände

Für Hearts wird ein kompletter Satz Karten benötigt. Der wird durch folgende Definitionen generiert, die jeweils eine Liste aller Farben, eine Liste aller Werte und schließlich daraus eine Liste aller Karten (also aller Kombinationen aus Farben und Werten) konstruiert:

```
allSuits :: [Suit]
allSuits = [Spades, Hearts, Diamonds, Clubs]

allRanks :: [Rank]
allRanks = [Numeric i | i <- [2..10]] ++ [Jack, Queen, King, Ace]

deck :: [Card]
deck = [Card suit rank | rank <- allRanks, suit <- allSuits]
```

Jede Definition wird von einer *Typdeklaration* wie *allSuits :: [Suit]* begleitet. Das bedeutet, dass *allSuits* eine *Liste* (die eckigen Klammern) von Farben ist. Die Definitionen für *allRanks* und *deck* benutzen sogenannte Comprehension-Syntax1, um die Werte und schließlich alle Karten aufzuzählen.

Für die Umsetzung eines Kartenspiels müssen die Karten repräsentiert werden, die eine Spielerin auf der Hand hat: als Menge (“set”) von Karten.

```
type Hand = Set Card
```

Der Typ *Set* ist ein generischer Typ, der von der Standardbibliothek importiert wird. Listing 1 zeigt die Typsignaturen der Funktionen aus *Set*.

Listing 1: Standard-Funkionen für *Set*

```
Set.null :: Set -> Bool
Set.member :: a -> Set a -> Bool
Set.insert :: a -> Set.Set a -> Set.Set a
Set.delete :: a -> Set a -> Set a
```

Ende

Für Hand wird eine *type*-Deklaration verwendet, die ein Typsynonym definiert.

Einige Hilfsdefinitionen erleichtern den Umgang mit dem Typ Hand. Die Funktion *isHandEmpty* hat den Typ *Hand -> Bool*, was eine “Funktion, die eine Hand als Eingabe nimmt und

1 Pythons Comprehensions sind von den funktionalen Sprachen Haskell und Miranda inspiriert.

ein Bool als Ausgabe liefert" beschreibt. In der Gleichung für *isHandEmpty* steht nach dem Funktionsname der Name des Parameters *hand*:

```
isHandEmpty :: Hand -> Bool
isHandEmpty hand = Set.null hand
```

Die Funktion *containsCard* nimmt zwei Argumente und prüft mit Hilfe der Library-Funktion Set.member, ob eine gegebene Karte zu einer Hand gehört:

```
containsCard :: Card -> Hand -> Bool
containsCard card hand = Set.member card hand
```

Der Typ *Card -> Hand -> Bool* wird erst verständlich, wenn er von rechts geklammert wird: *Card -> (Hand -> Bool)*. Das bedeutet, dass die Funktion zunächst eine Karte akzeptiert und dann *eine Funktion liefert*, die ihrerseits eine Hand akzeptiert und dann einen booleschen Wert zurückliefert. Streng genommen kennt Haskell nur einstellige Funktionen und "simuliert" mehrstellige Funktionen durch diese Technik2.

Die nächste Funktion zeigt beispielhaft, wie der Umgang mit unveränderlichen Daten funktioniert - removeCard entfernt eine Karte aus einer Hand:

```
removeCard :: Card -> Hand -> Hand
removeCard card hand = Set.delete card hand
```

In idiomatischem Java hätte das Hand-Objekt eine Methode *void removeCard(Card card)*, die den Zustand des Hand-Objekts entsprechend verändert. Nicht so in der funktionalen Programmierung, wo *removeCard* eine neue Hand liefert und die alte Hand unverändert lässt. Nach:

```
hand2 = removeCard card hand1
```

ist *hand1* immer noch die "alte" Hand und *hand2* die neue.

Das ist in Haskell nicht nur eine Konvention: Eine Funktion *kann* nicht einfach so Objekte "verändern", es handelt sich im Sprachgebrauch der funktionalen Programmierung immer um eine "reine" oder "pure" Funktion. In der *rein funktionale Programmierung* ist die Typsignatur enorm nützlich, weil sie alles aufführt, was in die Funktion hineingeht und wieder hinausgeht: Es gibt keine versteckten Abhängigkeiten zu globalem Zustand und alle Ausgaben stehen hinter dem rechten Pfeil der Signatur. Daher sind die rechten und linken Seiten einer Definitionsgleichung überall im Programm jederzeit austauschbar.

Typvariablen und Higher-Order-Funktionen

Da Hearts ein sogenanntes "Stichspiel" ist, hat der Code einen Typ für den Stich, auf englisch "Trick". Dieser muss mitführen, wer welche Karte ausgespielt hat, um nach einer Runde zu entscheiden, wer den Stich einziehen muss:

```
type PlayerName = String

type Trick = [(PlayerName, Card)]
```

Die Typdefinition von *Trick* besagt, dass ein Stich eine Liste (die eckigen Klammern) von Zwei-Tupeln (die runden Klammern innendrin) ist. Wir verwenden hier Listen, da die Reihenfolge der Karten wichtig ist, wenn es darum geht, welche Spielerin den Stich bekommt. Listen werden in funktionalen Sprachen "von hinten nach vorn" aufgebaut, die zuletzt ausgespielte Karte ist also vorn.

Wenn der Stich eingezogen wird, zählen nur noch die Karten, nicht mehr, von wem sie stammen. Dafür ist folgende Funktion nützlich:

```
cardsOfTrick :: Trick -> [Card]
cardsOfTrick trick = map snd trick
```

---

2 Funktionale Programmierer sprechen von *curried functions* und *currying*, https://de.wikipedia.org/wiki/Currying.

Was die Funktion macht ist wieder aus der Typsignatur ersichtlich. Interessant ist hier aber auch die Implementierung, weil sie die eingebaute *Higher-Order-Funktion map* bemüht, deren Typsignatur so aussieht:

```
map :: (a -> b) -> [a] ->[b]
```

Das *a* und das *b* sind *Typvariablen*, das Pendant zu Java-Generics. Ausgesprochen steht dort: *map* akzeptiert eine Funktion, die aus einem *a* ein *b* macht, sowie eine Liste von *a*s und liefert eine Liste von *b*s. Bei *cardsOfTrick* ist *a* das Zwei-Tupel *(PlayerName, Card)* und *b* ist *Card*. Die Funktion *snd* extrahiert aus dem Zwei-Tupel die zweite Komponente und hat deshalb folgenden Typ:

```
snd :: (a, b) -> b
```

Solche generischen Funktionen gibt es (inzwischen) auch in Java, aber in funktionalen Sprachen kommen sie im Zusammenhang mit Higher-Order-Funktionen viel häufiger zur Anwendung. Sie sind ein wichtiger Aspekt funktionaler Architektur: Diese macht nicht an der konkreten Modellierung von fachlichem Wissen halt, sondern erlaubt den Entwicklerinnen, Abstraktionen zu bilden, die das fachliche Wissen verallgemeinern.

Das Zusammenspiel von generischen Higher-Order-Funktionen und anderen reinen Funktionen liefert die Basis für ein extrem mächtiges Konstruktionsprinzip: Weil Funktionen nie etwas "Verstecktes" machen, können sie bedenkenlos zu immer größeren Gebilden zusammengestöpselt werden. In OO-Sprachen wächst jedoch mit der Größe das Risiko, dass versteckte Effekte unerwünschte Wechselwirkungen haben. Deshalb sind Disziplin und eigene architektonische Patterns notwendig, um die resultierende Komplexität in den Griff zu bekommen. In der funktionalen Programmierung ist das nicht so. Dementsprechend ist dort das "Programmieren im Großen" dem "Programmieren im Kleinen" ziemlich ähnlich.

Spiellogik

Die Spiellogik bildet den Mittelbau der Architektur. Ein Event-Storming liefert folgende Event-Klassen:

- Die Karten wurden ausgeteilt.
- Eine neue Spielerin ist an der Reihe.
- Eine Spielerin hat eine bestimmte Karte ausgespielt.
- Eine Spielerin hat den Stich aufgenommen.
- Eine Spielerin hat versucht, eine unzulässige Karte auszuspielen.
- Das Spiel ist vorbei.

Diese Klassen lassen sich direkt in eine Typdefinition übersetzen:

```
data GameEvent =
    HandsDealt (Map PlayerName Hand)
  | PlayerTurn PlayerName
  | CardPlayed PlayerName Card
  | TrickTaken PlayerName Trick
  | IllegalMove PlayerName
  | GameOver
```

Das *HandsDealt*-Event trägt eine "Map" zwischen Spielernamen und ihren Karten mit sich.3
Der Verlauf eines Spiels kann immer aus dessen Folge von Events rekonstruiert werden.
Es gibt nur zwei Klassen von Commands:

---

3 HandsDealt ist als Domain-Event eigentlich ungünstig, weil es an alle Spielerinnen verteilt wird, die so nicht nur ihre eigene Hand erfahren, sonderen auch die der anderen. Wir haben es hier aus Gründen der Einfachheit gewählt, aber besser wären separate HandDealt-Events für jeweils nur eine Spielerin, die von denen jede Spielerin nur jeweils "ihres" bekommt.

```
data GameCommand =
    DealHands (Map PlayerName Hand)
  | PlayCard PlayerName Card
```

Die erste Klasse ist das direkte Pendant zu *HandsDealt*; sie setzt das Spiel zu Beginn in Gang. Die zweite repräsentiert den Versuch einer Spielerin, eine bestimmte Karte auszuspielen.

Die Spielregeln werden durch die Verarbeitung von Commands zu Events implementiert. Die Regeln beziehen sich auf den *Zustand* des Spiels: welche Karten ausgespielt werden dürfen, welche Spielerin als nächstes dran ist etc. Von diesem Zustand wird folgendes verlangt:

- Wer sind die Spieler und in welcher Reihenfolge spielen sie?
- Was hat jede Spielerin auf der Hand?
- Welche Karten hat jede Spielerin eingezogen?
- Was liegt auf dem Stich?

Das alles wird durch eine weitere Record-Definition repräsentiert:

```
data GameState =
  GameState
  { gameStatePlayers :: [PlayerName],
    gameStateHands   :: PlayerHands,
    gameStateStacks  :: PlayerStacks,
    gameStateTrick   :: Trick
  }
```

Die Liste im Feld *gameStatePlayers* wird dabei immer so rotiert, dass die nächste Spielerin vorn steht. Für die beiden Felder *gameStateHands* und *gameStateStacks* müssen jeweils Karten *pro Spieler* vorgehalten werden, darum sind die dazugehörigen Typen Synonyme für Maps:

```
type PlayerStacks = Map PlayerName (Set Card)
type PlayerHands  = Map PlayerName Hand
```

Auch bei der Umsetzung der Spielregeln macht sich die funktionale Architektur bemerkbar: Während des Spiels wird der Zustand nicht verändert, sondern jede Änderung erzeugt einen neuen Zustand. Die zentrale Funktion für die Verarbeitung eines Commands hat deswegen folgende Signatur:

```
processGameCommand :: GameCommand -> GameState -> (GameState, [GameEvent])
```

Mit anderen Worten Command rein, (Repräsentation des) Zustand vorher rein, Tupel aus (Repräsentation des) neuem Zustand und Liste resultierender Events raus. Hier ist die Implementierung der Gleichung für das *DealHands*-Command:

```
processGameCommand (DealHands hands) state =
  let event = HandsDealt hands
  in (processGameEvent event state, [event])
```

Da dieser Befehl von der “Spielleitung” kommt, führt er immer zu einem *HandsDealt*-Event. Der Effekt des Events auf den Zustand wird durch die Funktion *processGameEvent* berechnet, deren Definition aus Platzgründen fehlt, aber deren Arbeitsweise sich wieder gut an der Typsignatur ablesen lässt:

```
processGameEvent :: GameEvent -> GameState -> GameState
```

Die Kernlogik ist in der Gleichung für das *PlayCard*-Command in Listing 2. Diese verlässt sich auf Hilfsfunktionen *playValid* (die einen Spielzug auf Korrektheit überprüft), *whoTakesTrick* (die berechnet, wer den Stich einziehen muss) und *gameOver* (die feststellt, ob das Spiel vorbei ist). Der Code führt eine Reihe von Fallunterscheidungen in Form von *if ... then ... else* durch und bindet lokale Variablen - insbesondere Events *event1*, *event2* etc. und Zwischenzustände *state1*, *state2* - mit dem Sprachkonstrukt *let*:

Listing 2: Funktion *processGameCommand*

```
processGameCommand (PlayCard player card) state =
  if playValid state player card
  then
    let event1 = CardPlayed player card
        state1 = processGameEvent event1 state
    in  if turnOver state1 then
          let trick = gameStateTrick state1
              trickTaker = whoTakesTrick trick
              event2 = TrickTaken trickTaker trick
              state2 = processGameEvent event2 state1
              event3 = if gameOver state2
                       then GameOver
                       else PlayerTurn trickTaker
              state3 = processGameEvent event3 state2
          in (state3, [event1, event2, event3])
        else
          let event2 = PlayerTurn (nextPlayer state1)
              state2 = processGameEvent event2 state1
          in (state2, [event1, event2])
  else
    (state, [IllegalMove player, PlayerTurn player])
```

Ende

Es ist deutlich zu sehen, dass der Zustand niemals verändert wird und dass alle Zwischenzustände separate, voneinander unabhängige Objekte sind.

Zustand verwalten

Die Funktion *processGameCommand* bildet die Abfolge von Zuständen durch unterschiedliche Variablen und die Abhängigkeiten dazwischen ab. Das ist in vielen Situationen gerade richtig, hier aber unnötig fehleranfällig - wenn zum Beispiel irgendwo statt *state3* mal *state2* steht. Unnötig ist es deshalb, weil die Funktion einen sequenziellen Prozess abbildet, und dem wäre besser durch eine sequenzielle Notation gedient.

Sequenzielle Prozesse lassen sich gut durch *Monaden* beschreiben, ein typisch funktionales Entwurfsmuster. Mit Monaden entstehen immer noch Funktionen, aber die Notation wechselt in eine sequenzielle Form. Diese hat Zugriff auf einen *Kontext*, der sowohl gelesen als auch beschrieben werden kann. Anders als in Java aber ist genau definiert, was in dem Kontext alles steht und was also die monadische Form kann und was nicht.

In Listing 3 steht eine monadische Version von *processGameCommandM*. Sie unterscheidet sich von der "funktionalen Version" folgendermaßen:

- Der Zustand und die "Event-Liste" sind implizit.
- Die Hilfsfunktion *processAndPublishEventM* verarbeitet den Effekt eines Events auf den Zustand und "speichert" das Event ab.
- An die Stelle der Hilfsfunktionen *playValid*, *turnOver*, *currentTrick*, *gameOver* treten monadische Versionen mit *M* hinten am Namen jeweils ohne state-Argument, die den Spielzustand aus dem Kontext beziehen. Auch anstelle des "normalen" *if* tritt die monadische Version *ifM*.
- Wenn mehrere monadische Aktionen hintereinander laufen, werden sie in einem do-Block untergebracht, ähnlich der geschweiften Klammern in Java. Dort wird <- benutzt, um das Ergebnis einer monadischen Operation an eine Variable zu binden.

Das Ergebnis in Listing 3 sieht zumindest strukturell fast wie ein Java-Programm aus.

Listing 3: Funktion *processGameCommandM*

```
processGameCommandM (DealHands playerHands) =
  processAndPublishEventM (HandsDealt playerHands)
processGameCommandM (PlayCard playerName card)
  ifM (playValidM playerName card)
    (do
      processAndPublishEventM (CardPlayed playerName card)
      ifM turnOverM
        (do
          trick <- currentTrickM
          let trickTaker = whoTakesTrick trick
          processAndPublishEventM (TrickTaken trickTaker trick)
          ifM gameOverM
            (processAndPublishEventM (GameOver))
            (processAndPublishEventM (PlayerTurn trickTaker)))
        (do                       -- not turnOver
          nextPlayer <- nextPlayerM
          processAndPublishEventM (PlayerTurn nextPlayer)))
    (do                           -- not playValid
      nextPlayer <- nextPlayerM
      processAndPublishEventM (IllegalMove nextPlayer)
      processAndPublishEventM (PlayerTurn nextPlayer))
```

Endfe

Die Funktion sieht zwar aus, als würde sie Änderungen *durchführen* - tatsächlich aber liefert sie nur eine *Beschreibung* dieser Änderungen. Diese Beschreibung muss dann explizit ausgewertet werden. Im Fall von Zustand heißt das zum Beispiel, dass aus der Beschreibung eine Funktion wird, die einen Zustand als Eingabe akzeptiert und einen neuen Zustand als Ausgabe liefert.

Welche Änderungen in der Monade gemacht werden können ist durch den Typ der Funktion eingeschränkt:

```
processGameCommandM :: GameInterface m => GameCommand -> m ()
```

Der Typ zerfällt in zwei Teile, die durch den Doppelpfeil => getrennt sind. Im rechten Teil taucht eine Typvariable *m* auf, die für die Monade steht. Links davon steht ein sogenannter *Constraint*, der sagt, was für Eigenschaften m haben muss. In diesem Fall steht dort, dass *m* das Interface *GameInterface* erfüllen muss - dazu gleich. Wenn dem so ist, so liefert *processGameCommandM* zu jedem Kommando eine Berechnung, die *()* produziert, das steht für "kein explizites Ergebnis" - die Ergebnisse sind alle implizit.

GameInterface hat folgende Definition:

```
type GameInterface m = (MonadState GameState m, MonadWriter [GameEvent] m)
```

Das *GameInterface* enthält zwei Features, die *processGameCommandM* benutzen darf - sie darf auf den Spielzustand zugreifen (*MonadState GameState*) und sie darf *GameEvent*s bekannt geben. Mehr *kann processGameCommandM* nicht tun. Da ist also der wesentliche Unterschied zu Java.[4] Da der Zustand nicht mehr explizit herumgereicht wird, muss sich eine Aktion wie *playValidM* den aktuellen *GameState* erst besorgen, indem sie die Funktion *State.get* benutzt, die zu *MonadState* gehört:

```
playValidM :: MonadState GameState m => PlayerName -> Card -> m Bool
playValidM playerName card = do
  state <- State.get
  return (playValid state playerName card)
```

4 Das heißt nicht ganz: In Java gibt es ja die *throws*-Klausel an Methoden, die besagt, welche Exceptions eine Funktion werfen kann.

Diese Funktion hat noch weniger Anforderungen an die Monade, denn sie verlangt nur den Zustandsanteil. Die Aktion *state <- State.get* besorgt den aktuellen Zustand, der für die folgenden Aktionen in der Variable state zur Verfügung steht. Das verbleibende *return* gibt den Zustand unverändert weiter und liefert als Ergebnis den Wert von *playValid*.

Auf die gleiche Art und Weise funktionieren auch die Aktionen *turnOverM*, *currentTrickM*, *gameOverM* und *nextPlayerM*.

Die Funktion *processAndPublishEventM* verarbeitet Events und verschickt sie:

```
processAndPublishEventM :: GameInterface m => GameEvent -> m ()
processAndPublishEventM gameEvent =
  do processGameEventM gameEvent
     Writer.tell [gameEvent]
```

Die Funktion benutzt das andere Feature in *GameInterface* - *MonadWriter*, wo es eine Funktion *Writer.tell* gibt, die das Event speichert und bekannt gibt. Außerdem benutzt sie die monadische Version von *processGameEvent* an, deren erste Gleichung folgendermaßen lautet:

```
processGameEventM :: GameInterface m => GameEvent -> m ()
processGameEventM (HandsDealt playerHands) =
  do gameState <- State.get
     State.put (gameState { gameStateHands = playerHands })
```

Sie holt sich also den impliziten Zustand und schreibt dann eine neue Version zurück, in der die verteilen Karten vermerkt sind.

Alle Funktionen mit Effekten in solche monadische Form zu bringen ist anstrengend. Es sorgt aber dafür, dass an den Funktionssignaturen deutlich zu sehen ist, welches Arsenal von Effekten sie benötigen. Außerdem ist das Resultat jeweils immer noch eine Funktion, die unter kontrollierten Umständen aufgerufen werden kann.

Spielerlogik

Während die Spiellogik Kommandos entgegennimmt und dafür Events generiert, funktioniert die Spielerlogik genau andersherum. Sie nimmt Events entgegen und liefert als Antwort Kommandos, die an die Spiellogik weitergegeben werden.

Die Implementierung der Spielerlogik funktioniert wieder mit Monaden, um die Formulierung zu erleichtern und gleichzeitig explizit zu machen, welcher Effekte sich eine Spielerin bedient. Das heißt, jede Spielerin wird in einer abstrakten Monade gestartet, von der sie nur weiß, dass sie Kommandos an die Spiellogik schicken kann und dass sie Zugriff auf I/O-Operationen hat - zum Beispiel um über ein GUI zu interagieren oder den Telefonjoker anzurufen. Diese Features werden als Constraints ausgedrückt:5

```
type PlayerInterface m = (MonadIO m, MonadWriter [GameCommand] m)
```

Jede Spielerin kann nun für sich selbst entscheiden, welche weiteren Features sie lokal verwenden möchte. Typischerweise verwaltet jede Spielerin ihre eigene Version vom Spielzustand, weil sie *nicht* auf den GameState der Spiellogik zugreifen kann. Die Ausgestaltung dieses Spielzustands ist der Spiellogik völlig gleichgültig und kann auch von jeder Spielerin anders gehandhabt werden. Diese Spielerlogik wird - natürlich - durch eine Funktion repräsentiert, die zusammen mit dem Namen der Spielerin in ein Record verpackt wird:

```
data Player =
  Player {
    playerName :: PlayerName,
```

5 MonadIO m ist ein vordefiniertes Constraint, das Zugriff zu allen I/O-Operationen erlaubt. Selbstverständlich kann man Einschränkungen definieren, sodass nur bestimmte I/O-Operationen zulässig sind.

```
    eventProcessor :: forall m . PlayerInterface m => GameEvent -> m Player
  }
```

Im *Player* hat eine Spielerin einen Namen und eine Event-Prozessor Funktion, die ein *GameEvent* als Aktion in einer Spielermonade m interpretiert. Dieses *m* kann beliebig gewählt werden (das wird durch das *forall m* ausgedrückt) und kann sich darauf verlassen, dass das *PlayerInterface* vom Aufrufer zur Verfügung gestellt wird, also das Gegenstück zu *GameInterface*.

In der Regel wird eine Spielerin im Spiel dazulernen wollen - also ihre Beobachtung des Spielverlaufs benutzen, um möglichst gute Spielzüge auszuwählen. Dazu muss sie sich Dinge merken, und das tut sie, indem ihr *eventProcessor* ein neues *Player*-Objekt zurückliefert, das in der nächsten Runde den Platz des alten einnimmt.

Es bleibt die Implementierung einer Spielerin. Um die Regeln einzuhalten muss jede Spielerin sich daran erinnern, welche Karten sie auf der Hand hat und was auf dem Stich liegt. Der Typ dazu sieht so aus:

```
data PlayerState =
  PlayerState { playerHand  :: Hand,
                playerTrick :: Trick }
```

Hilfreich ist die Funktion *playerProcessGameEventM*, die den Spielerzustand entsprechend der eingehenden Events ändert. Sie benötigt das normale *PlayerInterface*, aber auch einen *PlayerState* als Zustand:

```
playerProcessGameEventM :: (MonadState PlayerState m, PlayerInterface m) =>
PlayerName -> GameEvent -> m ()
```

Der Code in Listing 4 implementiert eine Spielerin, die bei Spielbeginn die Kreuz Zwei ausspielt, falls sie die Karte hat. Danach wählt sie jeweils eine passende Karte mit der Hilfsfunktion *playAlongCard* aus und spielt diese aus. Das tut sie, indem sie *Writer.tell* aufruft mit einem *PlayCard*-Command. Um verschiedene Events zu unterscheiden, benutzt die Funktion ein case-Konstrukt, das analog zu *switch* in Java funktioniert - der letzte Zweig _ entspricht dem Java-*default*.

Listing 4: *playAlongProcessEventM*

```
playAlongProcessEventM ::
  (MonadState PlayerState m, PlayerInterface m) =>
    PlayerName -> GameEvent -> m ()
playAlongProcessEventM playerName event =
  do playerProcessGameEventM playerName event
     playerState <- State.get
     case event of
       HandsDealt _ ->
         if Set.member twoOfClubs (playerHand playerState)
         then Writer.tell [PlayCard playerName twoOfClubs]
         else return ()

       PlayerTurn turnPlayerName ->
         if playerName == turnPlayerName
         then do card <- playAlongCard
                 Writer.tell [PlayCard playerName card]
         else  return ()

       _ -> return ()
```

Ende

Diese Spielstrategie wird von der folgenden Funktion in einem *Player*-Objekt verpackt. Die Strategie akzeptiert einen expliziten Zustand vom Typ *PlayerState*. Die dazugehörige Beschreibung der Zustandsveränderung wird in *playAlongProcessEventM* mit Hilfe der eingebauten Funktion

State.execStateT explizit gefüttert und auch wieder herausgeholt, unter dem Namen *nextPlayerState* - und der wird dann im nächsten Player verwendet.

```
playAlongPlayer :: PlayerName -> PlayerState -> Player
playAlongPlayer playerName playerState =
  let nextPlayerM event =
        do nextPlayerState <-
             State.execStateT (playAlongProcessEventM playerName event) playerState
           return (playAlongPlayer playerName nextPlayerState)
  in Player playerName nextPlayerM
```

Der Aufruf *State.execStateT* zeigt, dass *playAlongProcessEventM* trotz der monadischen Form eine “ganz normale” Funktion ist. Sie mutiert keine Variablen wie das in Java der Fall wäre. Stattdessen muss ein Programm sie explizit durch *State.execStateT* mit einem Anfangszustand aufrufen und bekommt dann einen expliziten Resultatzustand zurück.

Zusammenfassung

Es fehlen noch einige Funktionen, um die Spiellogik mit der Spielerlogik zu verdrahten. Diese fehlen aus Platzgründen hier, aber der vollständige Code ist im github-Repositorium vorhanden.

Wichtig war uns zu demonstrieren, wie Monaden Effekte beschreiben und in den Typsignaturen einschränken. Funktionale Softwarearchitektur bedeutet dementsprechend erst einmal, dass ein Entwickler nicht so leicht Architekturprinzipien verletzen kann: Nicht mal so eben schnell ein Attribut verändern oder - speziell in Haskell - externe Effekte auslösen, weil es gerade so passt. Stattdessen führen neue Zustände immer gleich zu neuen Objekten und alle Effekte müssen explizit deklariert werden. Das ist für Entwicklerinnen mit OO-Hintergrund gewöhnungsbedürftig. Diese Einschränkungen bringen aber Verbesserungen der Architekturqualität mit sich: geringere Kopplung, weniger Abhängigkeiten, deklarierte und kontrollierte Effekte - das alles steigert die Robustheit, Flexibilität und Wartbarkeit des Codes.

Quellen

[1] https://www.haskell.org/

[2] Michael Sperber, Herbert Klaeren: *Schreibe Dein Programm!*, https://www.deinprogramm.de/

[3] Hutton, Graham: *Programming in Haskell*, 2nd edition, 2016.

[4] z.B. Joachim Breitners *Haskell for Readers* http://haskell-for-readers.nomeata.de/

[5| Hearts auf Wikipedia, https://de.wikipedia.org/wiki/Hearts

[6] Vaughn, Vernon: *Domain-Driven Design Distilled*, Pearson, 2016.

Michael Sperber
michael.sperber@active-group.de
Michael Sperber ist Geschäftsführer der Active Group GmbH. Er wendet seit über 25 Jahren funktionale Programmierung in Forschung, Lehre und industrieller Entwicklung an. Er ist Mitbegründer des Blogs funktionale-programmierung.de und Mitorganisator der jährlichen Entwicklerkonferenz BOB. Er ist außerdem Mitautor des iSAQB-Advanced-Curriculums „FUNAR - Funktionale Softwarearchitektur“.

Peter Thiemann
thiemann@informatik.uni-freiburg.de
Peter Thiemann ist Professor für Informatik an der Universität Freiburg und leitet dort den Arbeitsbereich Programmiersprachen. Er ist einer der führenden Experten zur funktionalen Programmierung, der partiellen Auswertung, domänenspezifischen Sprachen und zahlreichen

anderen Themen der Softwaretechnik. Seine aktuelle Forschung beschäftigt sich mit statischen und dynamischen Analysemethoden für JavaScript sowie Typsystemen für Protokolle.