# Functional Software Architecture

**Michael Sperber, Active Group GmbH; Peter Thiemann, University of Freiburg**

## Abstract

The design of resilient software architectures is a challenge. In classical object-oriented software projects the complexity rises super linear with the size. The problem can be kept in bay by rigorous programming discipline and frequent refactoring, but mutual dependencies and intricate state transitions prevail over time. Functional software architecture presents a different approach to structuring large systems compared to object-oriented designs. It avoids many sources of complexity and mutual dependencies in a system.

## 1 Aspects of functional software architecture

"Functional Software Architecture" is the result of software design using the means of functional programming. It is characterized by the followings aspects:

- The notion of an object with encapsulated state is replaced by a *function* that operates on *immutable data*.
- Functional languages enable a systematic design of data models and functions driven by *types* - regardless whether the language supports static or dynamic typing.
- Rigid hierarchical inheritance structures are replaced by flexible *domain specific languages* embedded in the functional host language.

Here we concentrate on the first bullet point, modeling with functions on immutable data. We also cast some light on the role of types.

The accompanying code is available in a GitHub repository:

`https://github.com/funktionale-programmierung/hearts/`

## 2 Overview

We explain the making of a functional software architecture with the card game *Hearts*[1], of which we discuss the most important parts. We start by fixing the rules of the game.

Hearts has four players and is played with the standard 52-card pack. During the game each player collects cards in an individual stack.

[1]Hearts on Wikipedia, `https://en.wikipedia.org/wiki/Hearts_(card_game)`

Initially, each player is dealt 13 cards. The game proceeds in 13 rounds. In each round players take turns to contribute one card openly to the trick (the cards on the table). The winner of the trick collects all four cards in his/her stack.

The player holding Two of Clubs leads (starts) the first trick (round) by playing this card. Subsequent players must follow suit of the leading card if able; otherwise, they may play any card. The trick is won by the player who played the highest card of the leading suit. The trick winner leads to the next trick, playing any card in hand.

When all tricks are played out, the winner is the player whose collected tricks have the lowest value. To calculate the value, Queen of Spades is worth 13 points, every card of Hearts is worth one point, and any other card counts 0 points. The value of a stack is the value of all cards in the stack.

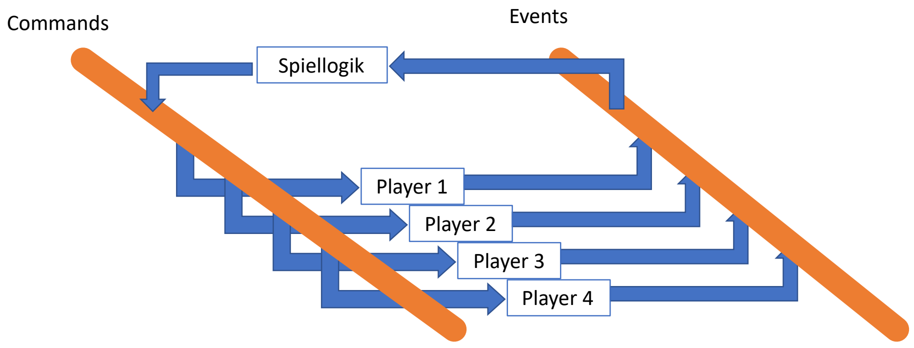

Figure 1: Modeling of game play

We start from a tactical design pattern from *Domain-Driven Design*[2] (DDD) and model the card game using *domain events*. Events represent every noteworthy element of the game play. Commands represent requests of the players. This way, the architecture is open for later adaptation to a client-server environment, microservices, or event-sourcing.

Figure 1 illustrates the overall flow: every player accepts events and generates commands that represents actions of the player. The component for game logic accepts commands, checks them for correctness (is this move legal according to the rules), and transforms them into events. The pattern is the same as in object-oriented DDD, but the realization is different because we rely on immutable data and functions.

Our example implements the communication between components of the architecture using function calls, but they could be replaced by communicating processes or microservices, for instance.

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

## 3 Programming with immutable data

Immutable data means that there is no assignment. A functional program has to generate new objects that reflect the new state after some change. This approach has many advantages

- There are never problems due to side effects (no hidden changes during a method call or due to a concurrently running process).
- There are no inconsistent intermediate states that occur when a program changes attributes sequentially.
- A program can be extended with a memory component that enables us to go backwards in time (i.e., take back moves and explore alternative outcomes).
- There are no hidden dependencies due to communication through shared state.

Java Gurus often advocate the use of *value objects* for the same reasons.

### 3.1 Modeling cards

The concrete modeling begins with the cards. The datatype `Card` is a record type (analogous to a POJO in Java), which fixes that a card has suit and rank.

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

We also require definitions of `Suit` and `Rank`:

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

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

Here is the definition of Two of Clubs based on these datatypes:

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

### 3.2 Card games and hands

Hearts requires a full set of playing cards. We generate this set by first constructing a list of all suits, a list of all ranks, and then a list of all combinations of these:

```
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]
```

We represent the cards on a player's hand as a set.

```
type Hand = Set Card
```

We establish several helper functions to work with data of type `Hand`. They should be self-describing.

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

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

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

As Hearts is based on tricks, there is a type representing a trick. This type must remember who played which card to determine the winner of the track.

```
type PlayerName = String

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

We are using lists because the sequence of cards is important when determining the winner of the trick.

Once the trick is assigned to the winner's stack, we can discard the players. Hence the following function:

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

## 4 Game logic

The game logic is the central structure of the architecture. Event storming yields the following categories of events:

- Cards were dealt.
- Next player's turn.
- Player played a card.
- Player takes the trick.
- Player attempts an illegal move.
- Game over.

These categories translate directly to the following type definition:

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

The event `HandsDealt` carries with it a map from player names to cards.[3] The full game play can be reconstructed from its sequence of events.

There are only two categories of Commands:

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

The first category is the direct counterpart of `HandsDealt`; it starts the game. The second represents the attempt of a player to play a certain card.

Processing the commands to events implements the rules of the game. The rules refer to the *state* of the game: which cards can be played, which player is next, and so on. The state needs to fix the following items:

- Who are the players and in which sequence are they playing?
- What are the cards of each player?
- Which cards are on the stack of each player?
- What's in the current trick?

## 4.1 Game state, take one

We represent the state with a record definition:

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

The list in field `gameStatePlayers` is rotated such that the next player is the first element of the list. The fields `gameStateHands` and `gameStateStacks` must remember cards *for each player*, hence their types are maps:

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

During the game, we do not change the state, but every progression generates a new state. Hence, the type of the central processing function for game commands:

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

[3] `HandsDealt` is not the best choice for a domain event because it is broadcast to *all* players. This way, every player gets to know the hand of every other player. We do that here for simplicity, but it would be better to have separate `HandsDealt` events that are only delivered to the respective player.

As an example, we show the case for the command `DealHands`:

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

As this command comes from the “game director”, it always results in a `HandsDealt`-event. The action of the event on the state is implemented by a function `processGameEvent`:

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

The core logic is implemented in processing the `PlayCard`-command. It relies on helper functions `playValid` (check move for correctness), `whoTakesTrick` (calculates the owner of the trick), and `gameOver` (checks if the game is finished).

```
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])
```

## 4.2 Administering state

Function `processGameCommand` implements a sequence of states by using different variables. This explicit programming is unnecessarily error prone as we know this function implements a sequential process.

Hence, we define a suitable monad and implement a monadic version `processGameCommandM`. It differs from the functional version is several respects:

- State and event list are implicit.

- The helper function `processAndPublishEventM` applies the effect of an event to the state and "saves" the event.
- The helper functions `playValid`, `turnOver`, `currentTrick`, `gameOver` are replaces by monadic version `M` appended to their name. The monadic `ifM` replaces the ordinary `if`.
- A sequence of monadic actions is collected in a `do`-block

```
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))
```

The scope of the monad (i.e., what can be changed) is limited by the constraint in the type of the function:

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

The constraint `GameInterface` is defined as follows:

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

`GameInterface` contains two features that may be used by `processGameCommandM`: it can check the game state (`MonadState GameState`) and it can issue `GameEvents`. No further interaction are possible.

As an example, we show the implementation of `playValidM`:

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

This function demands even less from the monad, it only looks at the state part.

The function `turnOverM`, `currentTrickM`, `gameOverM`, and `nextPlayerM` work in the same manner.

Function `processAndPublishEventM` is true to its name:

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

It uses the other feature in `GameInterface` - `MonadWriter`, where `Writer.tell` stores and publishes the event. It relies on the monadic version of `processGameEvent`. We show its first case as an example:

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

## 5 Player logic

While game logic takes commands and produces events, the player logic works exactly opposite. It takes evens and produces commands for the game logic.

The implementation of the player logic is again based on monads for several reasons: first to improve readibility of the code, second to make explicit the effects that players are allowed to use, and third to restrict access of the player. Effectively, each player runs in an abstract monad where commands can be sent to the game logic and there is access to I/O operations (e.g., to interact via GUI or to call the telephone joker). These features are expressed by constraints:[4]

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

Each player can decide which features theyr want to use. Typically, each player maintains a copy of the game state because it is *not* possible to access the global `GameState` that is maintained by the game logic. This copy is completely independent from the game logic and may be implemented differently for each player. Player logic is represented by a function that is stored in a record along with the player's name:

```
data Player =
  Player {
    playerName :: PlayerName,
    eventProcessor :: forall m . PlayerInterface m => GameEvent -> m Player
  }
```

[4]`MonadIO m` is a predefined constraint (in the standard library) that enables access to all I/O operations. It is possible to define restrictions that only allow certain I/O operations.

Each `Player` is characterized by a name and by a function that serves as an event processor. It interprets a `GameEvent` as an action in the player monad `m`. This monad `m` can be chosen arbitrarily (expressed by `forall m`) and it can rely on the caller supplying an implementation of the `PlayerInterface` , which is in some sense the counterpart to `GameInterface`.

Generally, a player wants to gain information during the game and consult the game history to select the best possible move. To this end, the player will keep some history, which is facilitated by having the `eventProcessor` return a new (updated) `Player`-object, which replaces the previous one in the next round.

It remains to implement a player. To adhere to the rules, every player has to remember their hand and the current trick. Here is a suitable type:

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

The function `playerProcessGameEventM` changes the player state according to the incoming events. It requires the `PlayerInterface` as well as a `PlayerState`:

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

As an example, we implement the initial move “play Two of Clubs in the beginning” if a player holds this card. Otherwise, the function `playAlongCard` chooses a suitable card and adds it to the current trick by calling `Writer.tell` with a `Playcard`-command.

```
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 ()
```

Next, we have to wrap this trivial stragety into a `Player`-object. The strategy accepts an explicit state of type `PlayerState`. The ensuing change of the state is implemented by `playAlongProcessEventM` using the library function `State.execStateT`. The current state is fed into the monad, retrieved after its execution (as `nextPlayerState`), and then returned for the next `Player`.

```
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
```

The invocation of `State.execStateT` demonstrates that the monadic `playAlongProcessEventM` is (also) an ordinary function. As it cannot perform side effects, the program has to invoke it explicitly by passing a suitable initial state to `State.execStateT` and picking up its result explicitly.

## 6 Concluding remark

For brevity, several functions are left out of the description of the game logic and player logic. However, the GitHub repository contains the full runnable code.

### Michael Sperber

`michael.sperber@active-group.de`

Michael Sperber is CEO of Active Group GmbH. He is an avid user and promotor of functional programming for more than 25 years in research, teaching, and industrial development. He is cofounder of the blog `funktionale-programmierung.de` and coorganizer of the yearly developer conference `BOB`.

### Peter Thiemann

`thiemann@informatik.uni-freiburg.de`

Peter Thiemann is a professor for computer science and leads the working group on programming languages at University of Freiburg. He is a leading expert in functional programming, partial evaluation, and domain-specific languages. His current research is on static and dynamic analysis methods for JavaScript as well as on expressive type systems for communication protocols.