関数型プログラミングの設計テクニック

猪股 健太郎

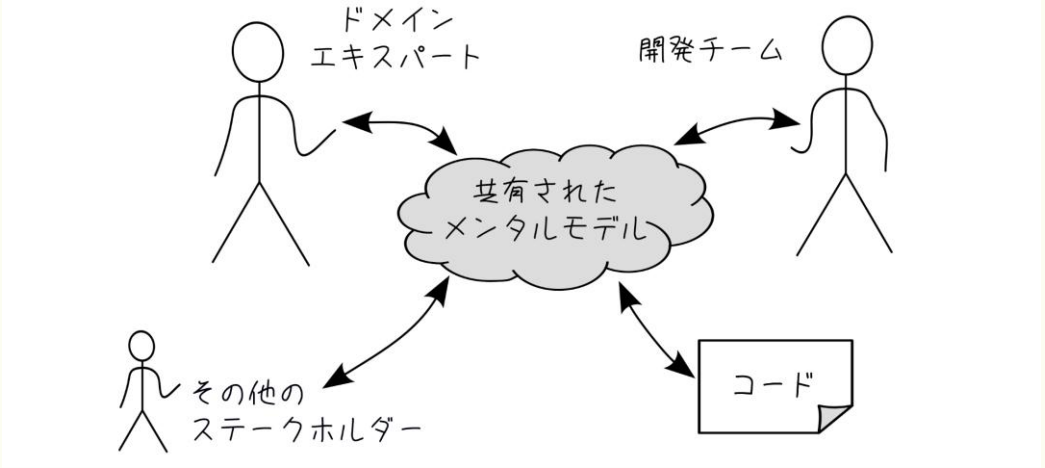
2024/09/07 Learn Languages 2024

『関数型ドメインモデリング』の紹介

- 『関数型ドメインモデリング
  - ドメイン駆動設計とF＃で
  ソフトウェアの複雑さに立ち向かおう』
  (アスキードワンゴ)
- 電子書籍もあります。
  - Amazon (Kindle)
  - 達人出版会 (PDF/EPUB)

『関数型ドメインモデリング』の主張 (1)

- ドメイン知識のメンタルモデルを共有
- 現実のドメインを解決空間に再作成する＝ドメインモデリング
- 再作成されたモデル＝ドメインモデル



『関数型ドメインモデリング』の主張 (2)

- ドメインモデルには特定技術を持ち込まない
- ドメインモデルとコードのモデルは一致させる
- 関数型プログラミング（とF#）で実現可能

  ※いつでもやるべきとは言っていない

(本書の)ドメインモデル

- ドメインやサブドメインを表現する「(境界づけられた)コンテキスト」
- ビジネスプロセスの一部を表現する「ワークフロー」と各ステップ
  - 入力と出力、依存関係、副作用

純粋関数

- ビジネスプロセスで扱うデータ構造
  - 単純な値
  - 値の組み合わせ(AND)、値の選択肢(OR)
  - さまざまな制約条件
  - ライフサイクル(状態遷移)

イミュータブルな
値と型

代数的データ型

F#の型

- 静的型づけ、非純粋（純粋関数推奨）
- 代数的データ型とパターンマッチング
  - 直積型（タプル、レコード）、直和型（判別共用体）
- 高階関数、ジェネリクス、カリー化

F#の代数的データ型

- 直積型(タプルとレコード)
  - ドメインモデルをコードに落とす際はレコードがおすすめ

```
// タプルに型定義は不要
let person: string * int = ("Yamada", 28)
```

```
type Person = { Name: string; Age: int }
let person: Person = { Name = "Yamada"; Age = 28 }
```

- 直和型(判別共用体)

```
type Hand = Rock ¦ Scissors ¦ Paper
```

```
type JsonPrimitive =
    ¦ String of string
    ¦ Number of decimal
    ¦ Bool of bool
    ¦ Null
```

省略可能な値、エラー、存在しない値

- 値が省略可能

```
type Option<'T> =
    | Some of 'T
    | None
```

- エラーの可能性がある

```
type Result<'T,'TError> =
    | Ok of ResultValue: 'T
    | Error of ErrorValue: 'TError
```

- 値が存在しない

```
let debugWrite () = // unit -> unit
    printfn "debug"
    ()
```

- 例外はドメインモデルをコードに落とすときは推奨しない

型を活用して表現する

- このContact型を、型を活かす設計に変えていく

```
type Contact =
    { FirstName: string
      MiddleInitial: string
      LastName: string

      EmailAddress: string
      IsEmailVerified: bool

      Address1: string
      Address2: string
      City: string
      State: string
      Zip: string
      IsAddressValid: bool }
```

構造を見つける

- 意味のある単位でまとめる
- 一貫性や整合性の単位を考慮
- ドメインの言葉に近づける

```
type Contact =
    { Name: PersonalName
      EmailContactInfo: EmailContactInfo
      PostalContactInfo: PostalContactInfo }
```

```
type EmailContactInfo =
    { EmailAddress: string
      IsEmailVerified: bool }
```

```
type PostalAddress =
    { Address1: string
      Address2: string
      City: string
      State: string
      Zip: string }

type PostalContactInfo =
    { Address: PostalAddress
      IsAddressValid: bool }
```

単純な値でもプリミティブ型は使わない

- メールアドレス、州名、郵便番号は別の型として定義したい
  - F#では単一ケースの判別共用体でラップ
- 型の混同を防ぎ、コンパイルエラーで検出可能に

```
type EmailAddress = EmailAddress of string

type EmailContactInfo =
    { EmailAddress: EmailAddress
      IsMailVerfied: bool }
```

```
type ZipCode = ZipCode of string
type StateCode = StateCode of string

type PostalAddress =
    { Address1: string
      Address2: string
      City: string
      State: StateCode
      Zip: ZipCode }
```

不正な状態は表現できないようにする (1)

- 「正しい状態」「不正な状態」はビジネスルールで決定
- 例:「連絡先にはメールアドレスまたは郵便住所のどちらかが必要」

3通りの可能性を並べた例

```
type ContactInfo =
    ¦ EmailOnly of EmailContactInfo
    ¦ PostOnly of PostalContactInfo
    ¦ EmailAndPost of EmailContactInfo * PostalContactInfo

type Contact =
    { Name: PersonalName
      ContactInfo: ContactInfo }
```

ビジネスルールを見直した例

```
type ContactMethod =
    ¦ Email of EmailContactInfo
    ¦ Post of PostalContactInfo

type Contact =
    { Name: PersonalName
      PrimaryContactInfo: ContactMethod
      SecondaryContactInfo: ContactMethod list }
```

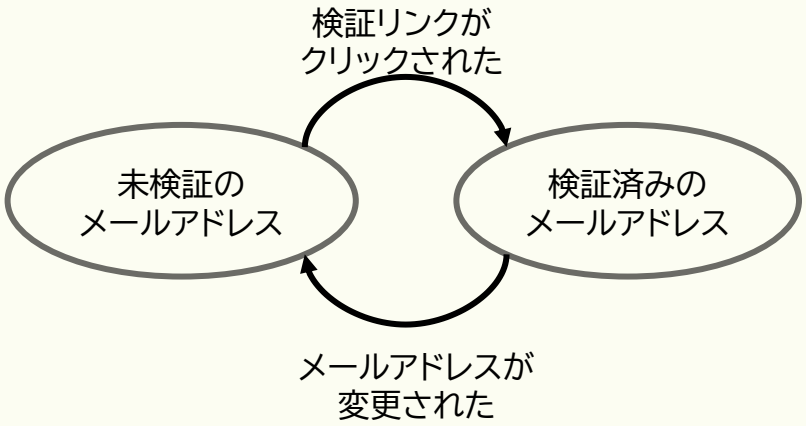
不正な状態は表現できないようにする (2)

- 例:「郵便番号は数字3桁+ハイフン+数字4桁」
- コンストラクタ関数をプライベートにし、`createZipCode` 関数を公開する
  - 戻り値の型は `Result<ZipCode, string>`

```
module ZipCode

open System.Text.RegularExpressions

type ZipCode = private ZipCode of string

let createZipCode (s: string) =
    let pattern = "^\\d{3}-\\d{4}$"

    if Regex.IsMatch(s, pattern) then
        Ok(ZipCode s)
    else
        Error "format error"
```

状態遷移を明示的に扱う

- メールアドレスの検証状態をステートマシンとして扱う
- 状態ごとに別の型で表現




```
type EmailAddress =
    EmailAddress of string

type VerifiedEmailAddress =
    private VerifiedEmailAddress of EmailAddress

type EmailContactInfo =
    | UnverifiedState of EmailAddress
    | VerifiedState of VerifiedEmailAddress
```

一貫性や整合性の単位 (DDD)

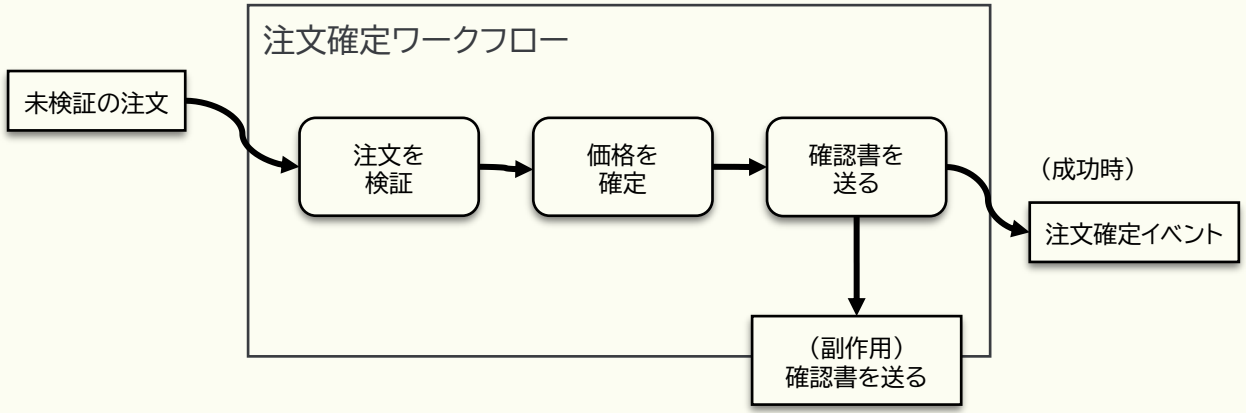
右の例：注文と注文明細行と顧客

- ドメインの「エンティティ」は識別子（ID）を持つ
- 「集約」は永続化、トランザクション、データ転送におけるアトミックな処理単位
- 識別子を介すことで疎結合に

```
type CustomerId = CustomerId of string
type OrderId = OrderId of string

type Customer =
    { Id: CustomerId
      Name: PersonName
    // etc
    }

type OrderLine =
    { Quantity: Quantity
    // etc
    }

type Order =
    { Id: OrderId
      Customer: CustomerId // 顧客に対する参照
      OrderLines: OrderLine list
    // etc
    }
```

ワークフローとステップを純粋関数で表す

- ワークフローやサブステップの「関数の型」を設計
- サブステップ関数を合成して、ワークフロー関数にする
  - パイプラインのイメージ
  - 型を合わせる



ワークフローとステップの入力と出力

- 出力が複数：直積型、直和型、またはコレクション
- 入力が複数：複数の引数、または直積型
- 別の入力（依存関係）：
  - 複数の引数として、関数を受け取る
- 別の出力（副作用）：
  - エラーになり得る　`Result<'T, 'TError>`
  - 非同期　`Async<‘T>`
  - エラーになり得る非同期　`Async<Result<'T, 'TError>>`

```fsharp
// 多用するので別名を付ける
type AsyncResult<'T, 'TError> =
    Async<Result<'T, 'TError>> // ResultをAsyncでくるむ
```

ステップ関数を合成してワークフロー関数にする

- ステップは入力として依存関係の関数も受け取る
- 依存関係の関数に副作用があると、ステップの出力も副作用を持つ
- 型が合わない！

注文を検証するステップ
ValidateOrder

入力:
　　UnvalidatedOrder
出力:
　　ValidatedOrder
　　or ValidationError

依存関係:
　　CheckAddressExists

価格を確定するステップ
PriceOrder

入力:
　　ValidatedOrder
出力:
　　PricedOrder

依存関係:
　　GetProductPrice

確認書を送るステップ
SendAcknowledgment

入力:
　　PricedOrde
出力:
　　なし

依存関係:
　　SendMessageToCustomer

型を合わせて関数合成する：入力

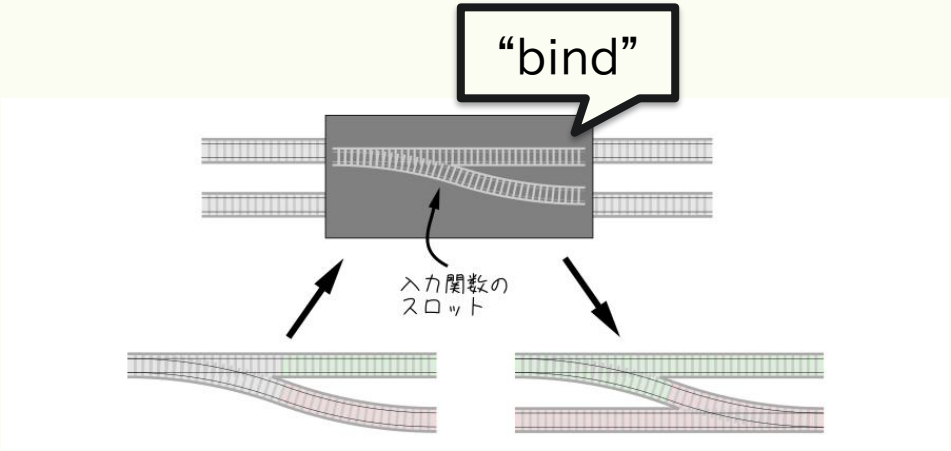
- ワークフロー関数の型は依存関係によらないように設計する
- 実装するとき、依存関係をすべて受け取ってワークフロー関数を返す関数を書く（部分適用）

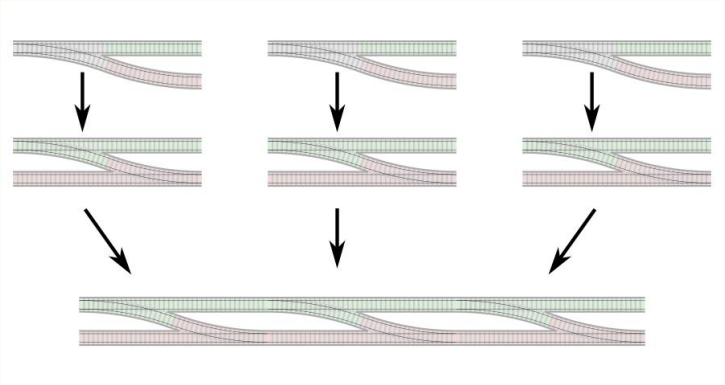
```
type PlaceOrderWorkflow =
    UnvalidatedOrder
        -> AsyncResult<PlaceOrderEvent, PlaceOrderError>
```

```
let placeOrder
    checkAddressExists // 依存関係
    getProductPrice // 依存関係
    sendMessageToCustomer // 依存関係
    : PlaceOrderWorkflow = // ワークフロー関数を返す
        // 実装
        // サブステップ関数に依存関係を部分適用
        // 出力の型を合わせながら合成
        // ...
```

型を合わせて関数合成する:出力

- 普通の値を受け取って`Result`を返す関数➡1入力・2出力のイメージ
- 2入力・2出力に変換すれば連結できる

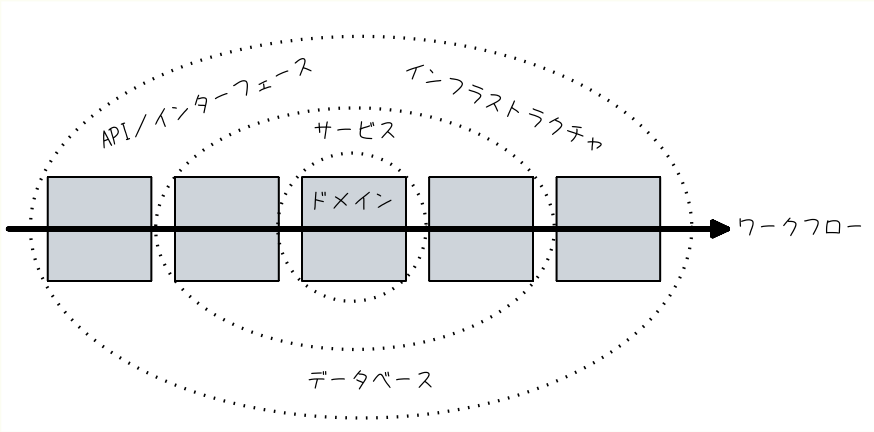




- エラーの型が合わないときは、設計上の抽象度に合わせる

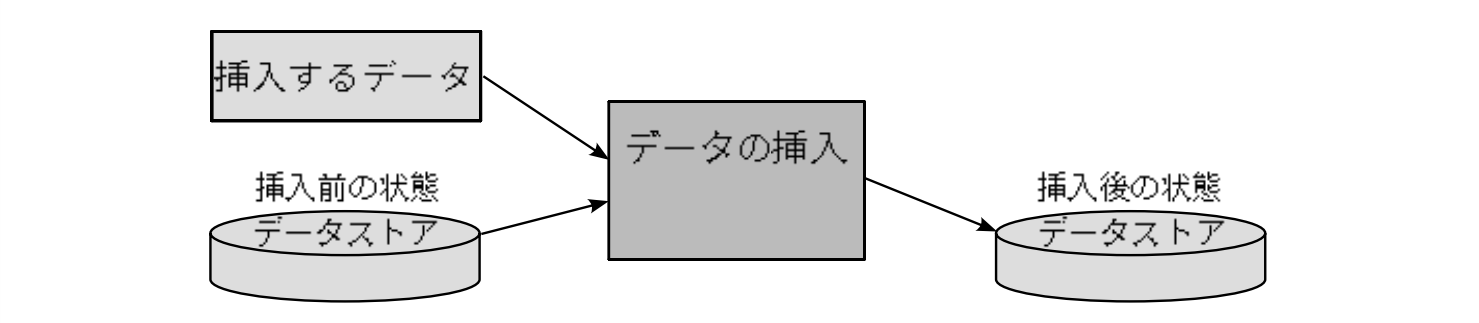
I/Oを端に追いやる(1)

- オニオンアーキテクチャ、クリーンアーキテクチャ



I/Oを端に追いやる(2)

- データベースの読み書き



I/Oを端に追いやる(2)

- Web API