SAR
DOT
NET

F# Patterns

Рустам Шехмаметьев

Паттерны

ООП

- Factory
- Command
- Builder
- ...

ФП

- Function
- Function
- Function
- ...

Функциональные паттерны

- Functor
- Monad
- Applicative
- Функциональное внедрение зависимостей

Обёртка

- Option // wrapper<'value>
- Result
- List

Option

```
type Option<'a> = Some of 'a | None
```


```
let getUserById: Guid -> User option
```

Result

```
type Result<'value, 'error> =
    Ok of 'value | Error of 'error
```

```
let validateUsername: string ->
Result<string, UsernameErrors>
```

List

[1;2;3;4;5]

```
type List<'a> =
    EmptyList
    | Item of 'a * List<'a>

let bootlegList = Item (1, Item (2, Item (3,
EmptyList)))
```

Задача

```
type User = {
   Id: Guid
   Username: string
   DateOfBirth: DateTime
}

type UserDto = {
   Id: Guid
   Username: string
   DateOfBirth: DateTime
}
```

```
let getUserById: Guid -> User option
let createUser: User -> Guid
let deleteUser: Guid -> bool
let updateUser: Guid -> User -> bool

let fromDto: UserDto ->
Result<User, UserErrors list>
```

Exception vs Result

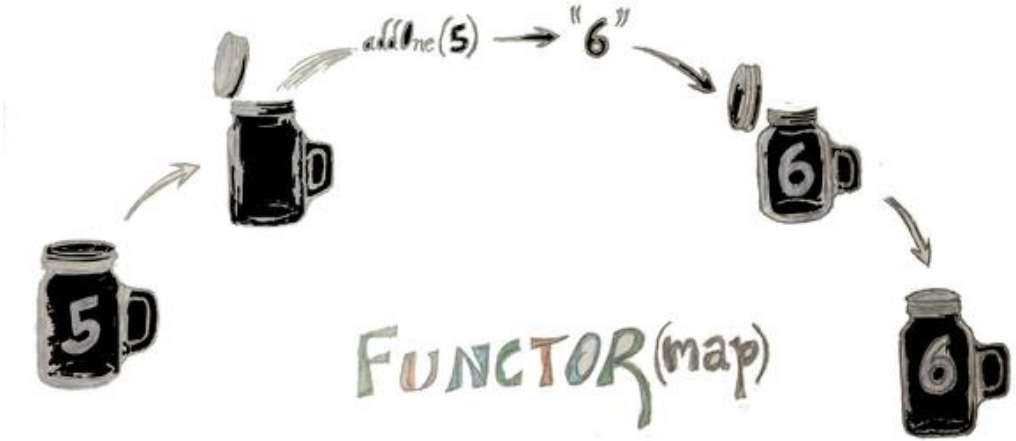
- Сложно агрегировать ошибки
- Сложно отследить все возможные исключения
- Валидационная ошибка не является исключительной ситуацией

- Ошибки хорошо агрегируются при помощи паттернов
- Все виды ошибок можно понять по сигнатуре функции
- Ошибочный Result - ожидаемая ситуация

Functor (Mappable)

Определение

```
let map: ('a -> 'b) -> Wrapper<'a> -> Wrapper<'b>
```



List

```
let listSelectFunctional lst =
    lst
    |> List.map add1
```

```
let chainCalls lst =
    lst
    |> List.map add1
    |> List.map mul2
```

```
List<int> ListSelectLinq(List<int> lst) =>
lst.Select(Add1).ToList()
```

```
List<int> ChainCallsLinq(List<int> lst) =>
lst.Select(Add1).Select(Mul2).ToList()
```

Связь между List и Option

```
type ListWithSingleElement<'a> = Item of 'a | EmptyList

type Option<'a> = Some of 'a | None


let singleListMap = List.map add1 [1] // [2]
let singleListEmptyMap = List.map add1 [] // []
```

Option

```
module Option =
// ('v -> 'v2') -> ‘v option -> v2 option
   let map f v = match v with
                 | Some v -> Some (f v)
                 | None -> None
```

```
let optionMap =
    Some 1
    |> Option.map add1 // Some 2

let optionMapNone =
    None
    |> Option.map add1 // None
```

Функция get

Нужно:

- Получить dto из источника данных
- Создать доменного пользователя из dto
- Вернуть или пользователя, или ничего

```
let getUserFromDb: Guid -> UserDto option

let fromDtoGet: UserDto -> User

let getUserById id =
    let userDto = getUserFromDb id
    match userDto with
    | Some userDto ->
        let user = fromDtoGet userDto
        Some user
    | None -> None
```

Функция get

```
let getUserById id =
   let userDto = getUserFromDb id
   match userDto with
   | Some userDto ->
      let user = fromDtoGet userDto
      Some user
   | None -> None
```

```
let getUserById id =
   let userDto = getUserFromDb id
   userDto |>  Option.map fromDtoGet
```

```
let getUserById id =
    id
    |> getUserFromDb
    |> Option.map fromDto
```

Result

```
module Result =
    let map f v =
        match v with
        | Ok v -> Ok (f v)
        | Error err -> Error err
```

```
let validMap = Ok 10
                |> Result.map add1
                |> Result.map mul2 // Ok 22

let invalidMap = Error "some error"
                  |> Result.map add1
                  |> Result.map mul2 // Error ...
```

Result

```
module Result =
    let mapError f v =
            match v with
            | Ok v -> Ok v
            | Error err -> Error (f err)
```

```
let validMapError =
    Ok 10
    |> Result.mapError add1 // Ok 10

let invalidMapError =
    Error "some error"
    |> Result.mapError (sprintf "[ERROR] %s")
```

Для чего нужен Functor

Для изменения обёрнутого значения без необходимости понимания структуры обёртки

Monad (Bindable, Chainable)

$$0 = \varnothing;$$
$$1 = \{0\} = 0 \cup \{0\} = \{\varnothing\};$$
$$2 = \{0, 1\} = 1 \cup \{1\} = \{\varnothing, \{\varnothing\}\};$$
$$3 = \{0, 1, 2\} = 2 \cup \{2\} = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\};$$
$$\ldots$$

Определение

```
let bind: ('a -> Wrapper<'b>) -> Wrapper<'a> -> Wrapper<'b>

let return: 'a -> Wrapper<'a>
```

List

```
module List =
   // 'v -> 'v list
   let return = List.singleton
   // ('v -> 'v2 list) -> 'v list -> 'v2 list
   let bind = List.collect


let selectManyFunctional lst =
    lst
    |> List.bind (fun x ->[x; x + 1])

selectManyFunctional [1..5]
// [1;2; 2;3 3;4 4;5 5;6]
```

```
private List<int> SelectManyLinq(List<int> lst) =>
lst.SelectMany(x => new List<int> {x, x +
1}).ToList();

SelectManyLinq(Enumerable.Range(1, 5).ToList());
// [1;2; 2;3 3;4 4;5 5;6]
```

List

```
let listBindEmpty =
    []
    |> List.bind (fun x -> List.return (x + 1)) // []

let listBindAsMap =
    List.return 1
    |> List.bind (fun x -> List.return (x + 1)) () // [2]


let listBindFunctionEmpty =
    List.return 1
    |> List.bind (fun x -> []) // []
```

Option

```
module Option =
   // ‘v -> ‘v option
   let return v = Some v
   // ('v -> 'v2 option) -> 'v option -> 'v2 option
   let bind f s = match s with
                  | Some v -> f v
                  | None -> None
```

```
let optionBindEmpty =
    None
    |> Option.bind (fun x -> Option.return (x + 1))
// None

let optionBindNone =
   Option.return 10
   |> Option.bind (fun x -> None)
// None

let optionBindMap =
    Option.return 10
    |> Option.bind (fun x -> Option.return (x + 1))
// Some 11
```

Bind vs Map

```
let optionBindMap =
    Option.return 10
    |> Option.bind (fun x -> Option.return (x + 1))
// Some 11
```

```
let optionMap =
    Option.return 10
    |> Option.map (fun x -> Option.return (x + 1))
// Some (Some 11)
```

Result

```
module Result =
   // ('v -> Result<'v, 'error>)
   let return v = Ok v
   // ('v -> Result<'v, 'error>)
   let fail err = Error err
   // ('v -> Result<'v2, 'error>) -> Result<'v1, 'error> -> Result<'v2, 'error>
   let bind f s = match s with
                  | Ok v -> f v
                  | Error err -> fail err

let resultAllEmpty = fail "error 1"
                     |> Result.bind (fun x -> fail "error 2") // Error "error 1"
```

Правила валидации пользователя

- Имя пользователя не может быть пустым
- Имя пользователя не должно быть длиннее 18 символов
- Пользователю должно быть минимум 18 лет
- Пользователь не может быть старше 122 лет

```
type UsernameErrors = UsernameEmpty | UsernameTooLong
type DateOfBirthErrors = TooYoung | TooOld
type UserErrors = UsernameError of UsernameErrors
                | DateOfBirthError of DateOfBirthErrors
```

Правила валидации пользователя

```
type Rule<'value, 'error> = 'value -> Result<'value, 'error>

let usernameCannotBeEmptyRule: Rule<string, UsernameErrors>

let usernameCannotBeLongerThan18CharsRule: Rule<string, UsernameErrors>

let ageMustBeGreaterThan18Rule: Rule<DateTime, DateOfBirthErrors>

let ageMustBeLessThan122Rule: Rule<DateTime, DateOfBirthErrors>
```

Валидация имени пользователя

```
// string -> Result<string, UsernameErrors>
let validate username =
   match  rule1 username with
   | Ok usr -> rule2 usr
   | Error err -> Result.fail err
```

```
match validUsername with
  | Ok usr -> match rule1 username with
          | Ok usr -> match rule2 usr with
                  | Ok usr -> match rule3 with
                          | Ok usr ->
                             match rule4 usr with ...
          | Error err -> Result.fail err
  | Error err -> Result.fail err
```

Решение

```
// string -> Result<string, UsernameErrors>
let validate username =
   match rule1 username with
   | Ok usr -> rule2 username
   | Error err -> Result.fail err
```

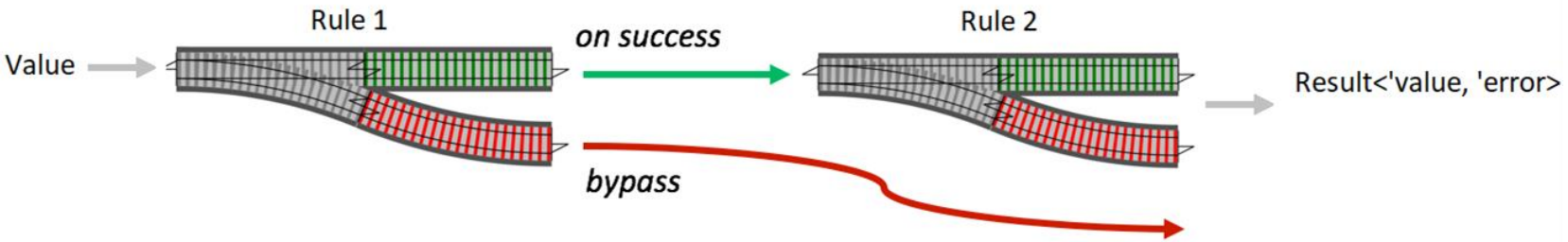
```
let validate username =
    rule1 username
    |> Result.bind rule2
```

```
let validateMultipleRules username =
     rule1 username
     |> Result.bind rule2
     |> Result.bind rule3
    ...
```

```
let (>>=) s f = bind f s
```

```
let validate username =
    rule1 username
    >>= rule2 username
```

Railway oriented programming



Композиция Monad

```
// (‘a -> Result<’a, ‘error>) list
let rulesList = [rule1;rule2;rule3;rule4;...]

let reduceExample = List.reduce (+) [1..10]
// 1 + 2 + 3 + ... + 10

let validate v = v |> List.reduce (>>) rulesList
// не скомпилируется, все функции должны иметь тип ‘a -> ‘a
```

Оператор Клеисли

```
// (‘a -> Result<’a, ‘error>) -> (‘a -> Result<’a, ‘error>) -> (‘a -> Result<’a, ‘error>)
let kleisly f2 f1 = fun v -> f1 v >>= f2
let (>=>) = f1 |> kleisly f2

// ‘a -> Result<’a, ‘error>
let validate v = v |> List.reduce (>=>) rulesList

let validateUser = rule1 >=> rule2 >=> rule3
```

Monad в Реальном Мире™

https://github.com/giraffe-fsharp/Giraffe

```
let testHandler : HttpHandler = GET >=> route "/hello" >=> text "Hello, world"
```

Computational expressions

```
type OptionBuilder() =
   member x.Bind(s, f) = Option.bind f s
   member x.Return(v) = Option.return v

let option = new OptionBuilder()
```

Пример Builder

```
// Guid -> User option
let getUserById id =
   option {
      let! userDto = getUserFromDb id //
UserDto


      return fromDtoGet userDto
   }
```

```
let getUserById id =
   let option = new OptionBuilder()
   option.Bind(getUserFromDb id,
      fun userDto ->
         option.Return(fromDtoGet userDto))
```

Async workflow builder

```
// string -> Async<string>
let google query =
  async {
      use client = new HttpClient()
      let! response = client.GetAsync(sprintf "google.com/search?q=%s" query)
                          |> Async.AwaitTask
      let! resultsPage = response.Content.ReadAsStringAsync()
                             |> Async.AwaitTask
      return resultsPage
  }
let googleHelloWorld = google "Hello, world" |> Async.RunSynchronously
```

Для чего нужен Monad?

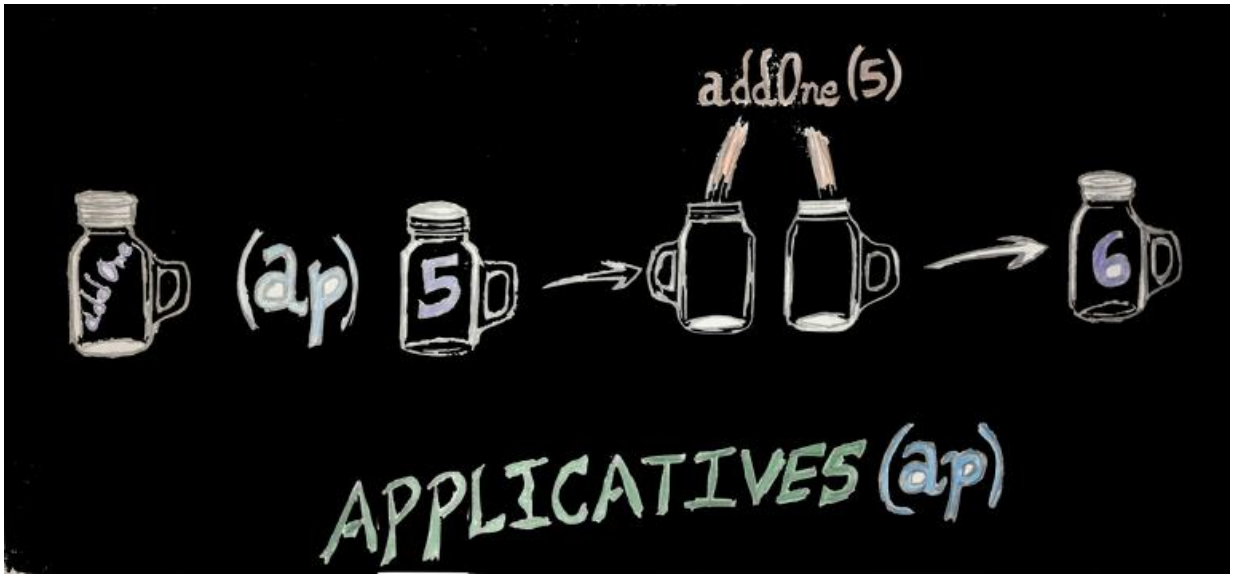
Для вычислений “по цепочке” функций,
возвращающих обёртки
(“монадических функций”)

Applicative

Определение

```
let apply: Wrapper<'a> -> Wrapper<('a -> 'b)> -> Wrapper<'b>

let pure: 'a -> Wrapper<'a>
```



Option

```
module Option =
// 'a option -> ('a -> 'b) option -> 'b option
let apply s f = match f, s with
                | Some f, Some s -> Some (f s)
                | _ -> None
let pure = Some
```

```
let applyNone =
    None
    |> Option.apply Some 10 // None

let applyNoneFunc =
    Some add1
    |> Option.apply None // None

let applySome =
    Some add1
    |> Option.apply Some 10 // Some 11
```

Option

```
let create: int -> PositiveNumber option
let div (PositiveNumber a) (PositiveNumber b)
= PositiveNumber (a / b)

// int -> int -> PositiveNumber option
let tryDiv a b =
    match create a, create b with
    | Some a, Some b -> Some (div a b)
    | _ -> None
```

```
let tryDivMonad a b =
    create a
    |> Option.bind (fun a -> create b
                             |> Option.bind (fun b ->
                                             Some (div a b)))

// int -> int -> PositiveNumber option
let tryDivApplicative a b =
    Option.pure div
    |> Option.apply (create a)
    |> Option.apply (create b)
```

Option

```
module Option =
// 'a option -> ('a -> 'b) option -> 'b option
let apply s f =
    match f, s with
     | Some f, Some s -> Some (f s)
     | _ -> None
let pure = Some
```

```
// int -> int -> PositiveNumber option
let tryDivApplicative a b =
    Option.pure div
    |> Option.apply (create a)
    |> Option.apply (create b)
```

```
let tryDivApplicative a b =
   let pureDiv = pure div // Option (int -> int -> PositiveNumber)
   let apply1 = pureDiv |> Option.apply (create a) // Option (int -> PositiveNumber)
   let apply2 = apply1 |> Option.apply (create b) // Option (PositiveNumber)
   apply2
```

Result

```
module Result =
  // Result<'a, 'error list> -> Result<('a -> 'b), 'error list> -> Result<'c, 'error list>
  let apply s f = match f, s with
                  | Ok f, Ok s -> Ok (f s)
                  | Ok f, Error s -> Error s
                  | Error f, Ok s -> Error f
                  | Error f, Error s -> Error (f @ s)
  let pure = Ok
```

Функция fromDto

```
let fromDto userDto =
  let create' username dob = { Id = userDto.Id
                               Username = username
                               DateOfBirth = dob }
  match validateUsername userDto.Username,
        validateDateOfBirth userDto.DateOfBirth with
  | Ok username, Ok dob -> create' username dob |>  Ok
  | Error username, Ok dob -> Error [UsernameError username]
  | Ok username, Error dob -> Error [DateOfBirthError dob]
  | Error username, Error dob -> Error [UsernameError username; DateOfBirthError dob]
```

Функция fromDto

```
let fromDtoApplicative userDto =
  let create' username dob = { Id = userDto.Id
                               Username = username
                               DateOfBirth = dob }
  Result.pure create'
  |> Result.apply (validateUsername userDto.Username
                   |> Result.mapError (UsernameError >> List.singleton)
  |> Result.apply (validateDateOfBirth userDto.DateOfBirth
                   |> Result.mapError (DateOfBirthError >> List.singleton))
```

Для чего нужен Applicative?

Для вычислений “монадических функций” с возможностью аккумуляции результатов вычислений

Functional dependency injection

Проблема

```
let getUserById v = v |> getUserFromDb |> Option.map fromDtoGet
let fromDtoApplicative userDto =
  let create' username dob = { Id = userDto.Id
                               Username = username
                               DateOfBirth = dob }
  Result.pure create'
  |> Result.apply (validateUsername userDto.Username
                   |> Result.mapError (UsernameError >> List.singleton))
  |> Result.apply (validateDateOfBirth userDto.DateOfBirth
                   |> Result.mapError (DateOfBirthError >> List.singleton))
```

Функции как интерфейсы

```
let getUserFromDb: Guid -> UserDto option
let fromDtoGet: UserDto -> User
let validateUsername: Rule<string, UsernameErrors>
let validateDateOfBirth: Rule<DateTime, DateOfBirthErrors>


public interface IGetUserFromDb
{
   Option<UserDto> GetUser(Guid guid);
}
```

Шаг 1: Параметризация

```
let getUserById fromDtoGet getUserFromDb v = v
                                             |> getUserFromDb
                                             |> Option.map fromDtoGet
let fromDto validateUsername validateDateOfBirth userDto =
  let create' username dob = { Id = userDto.Id
                               Username = username
                               DateOfBirth = dob }
  Result.pure create'
  |> Result.apply (validateUsername userDto.Username
                   |> Result.mapError (UsernameError >> List.singleton))
  |> Result.apply (validateDateOfBirth userDto.DateOfBirth
                   |> Result.mapError (DateOfBirthError >> List.singleton))
```

Шаг 2: Частичное применение

```
let getUserById: (UserDto -> User) ->
                 (Guid -> UserDto option) ->
                 Guid ->
                 User option
let fromDto: (string -> Result<string, UsernameErrors>) ->
             (DateTime -> Result<DateTime, DateOfBirthErrors>) ->
              UserDto ->
              Result<User, UserErrors>

// Guid -> User option
let getUserByIdDefault = getUserById fromDtoGetImpl getUserFromDataSourceImpl

// UserDto -> Result<User, UserErrors>
let fromDtoDefault = fromDto validateUsernameImpl validateDateOfBirthImpl
```

Итоги

- Рассмотрели основные паттерны
- Затронули архитектурные паттерны (разделение домена и данных, DI)
- Рассмотрели computational expressions

Источники

- F# for fun and profit (https://fsharpforfunandprofit.com)
- Giraffe (https://github.com/giraffe-fsharp/giraffe)
- Learn You a Haskell for Great Good (http://learnyouahaskell.com)
- Computational expressions (https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions)

Вопросы?

Email: shekhmametyev.rustam@gmail.com
Skype: wild.mantis