Доменное моделирование и архитектура приложений на ReScript

Сергей Самохов

Банк Точка

hoichi

hi@hoichi.io

HOLY JS/...

точка

СЕРЕГА САМОХОВ

- В основном фронт
- Банк Точка
- FP и простота
- hoichi.io

ПРО ДОКЛАД

- Доменное моделирование
- Статические типы
- ReScript

DOMAIN-DRIVEN DESIGN

- Доменные эксперты (aka «Бизнес»)
- Единый язык

RESCRIPT

- Функциональный (без фанатизма)
- Прагматичный
- Типы
  - строгие
  - хорошо выводятся
  - выразительные

ПЛАН ДОКЛАДА

1. Исключение невозможных состояний
2. Моделирование процессов и обработка ошибок
3. Ввод-вывод
4. Общий подход к архитектуре

1. ИСКЛЮЧЕНИЕ НЕВОЗМОЖНЫХ СОСТОЯНИЙ

ЛЕТИТ ДРАКОН...

```
type dragon = {
  heads: array<dragonHead>,
  isComing: bool,
  hasCrashed: bool,
}
```

```
let dragon = {
  heads: [{...}, {...}],
  isComing: true,
  hasCrashed: true,
}
```

АЛГЕБРАИЧЕСКИЕ ТИПЫ ДАННЫХ

1. Типы-произведения

```
type flyingMonsterState = {
  isComing: bool,
  hasCrashed: bool,
}
```

```
type flyingMonsterState = {
  isComing: bool,   // true + false == 2
  hasCrashed: bool, // true + false == 2
}                   // 2 * 2 == 4
```

1. Типы-произведения

```
type flyingMonsterState = {
  isComing: bool,      // true + false == 2
  hasCrashed: bool,    // true + false == 2
  isPoisonous: bool,   // true + false == 2
}                      // 2 * 2 * 2 == 8
```

```
type flyingMonsterState = {
  isComing: bool,
  hasCrashed: bool,
}                    // 2 * 2 == 4
```

isComing	**hasCrashed**	**законно**
false	false	да
true	false	да
true	true	*НЕТ*
false	true	да

ПОПЫТКА №2: СУММАРНЫЙ ТИП

```
type flyingMonsterState = Coming | Crashed | Landed //  1+1+1

type dragon = {
  heads: array<dragonHead>,
  state: flyingMonsterState,
}
```

```
type flyingMonsterState = Coming | Crashed | Landed

type dragon = {
  heads: array<dragonHead>,
  state: flyingMonsterState,
}

let dragon = {
  heads: [],    // ???
  state: Coming
}
```

ПОПЫТКА №3

```
type dragon = {
  heads: option<array<dragonHead>>,
  state: flyingMonsterState,
}

let dragon = { heads: None, state: Coming }
```

ПОПЫТКА №3

```
let dragon2 = { heads: Some([...]), state: Crashed }
let dragon3 = { heads: None, state: Landed }
```

СУММАРНЫЙ ТИП С АРГУМЕНТАМИ

```
type dragon =
  | Coming
  | Crashed
  | Landed(array<dragonHead>)
```

```
type dragon =
  | Coming
  | Crashed
  | Landed(array<dragonHead>)

switch someDragon {
| Coming => ()
| Crashed => ()
| Landed(heads) => heads->Array.forEach(_ => ())
}
```

А ГОЛОВУ ТЫ ДОМА НЕ ЗАБЫЛ?

```
type landedDragon = { heads: array<dragonHead> }

let chopDragonHead: (landedDragon) => dragonHead

let marryThePrincess: (knight, dragonHead) => prince
```

```
let dragon = { heads: [] }
```

```
dragon.heads[0] // option<dragonHead>
```

```
let dragon = { heads: [] }
let chopDragonHead: (landedDragon) => option<dragonHead>
```

```
let chopDragonHead: (landedDragon) => option<dragonHead>

let marryThePrincess:
  (knight, option<dragonHead>) =>
    Married<prince> | Beheaded<deadKnight>
```

УМНЫЙ В ГОРУ НЕ ПОЙДЕТ

```
switch chopDragonHead(dragon) {
| Some(head) => Some(marryThePrincess(head, knight))
| None => None
}

// aka
landedDragon
  ->chopDragonHead
  ->Option.map(marryThePrincess(_, knight)) // Some(prince) |
```

НЕПУСТЫЕ СПИСКИ

```
type nonEmptyList<'a> = ('a, array<'a>) // ну или запись

type landedDragon = {
  head: nonEmptyList<head>,
}

let dragon = {
  heads: [] // error
}
```

```
let chopDragonHead: (landedDragon) => dragonHead
let marryThePrincess: (knight, dragonHead) => prince
```

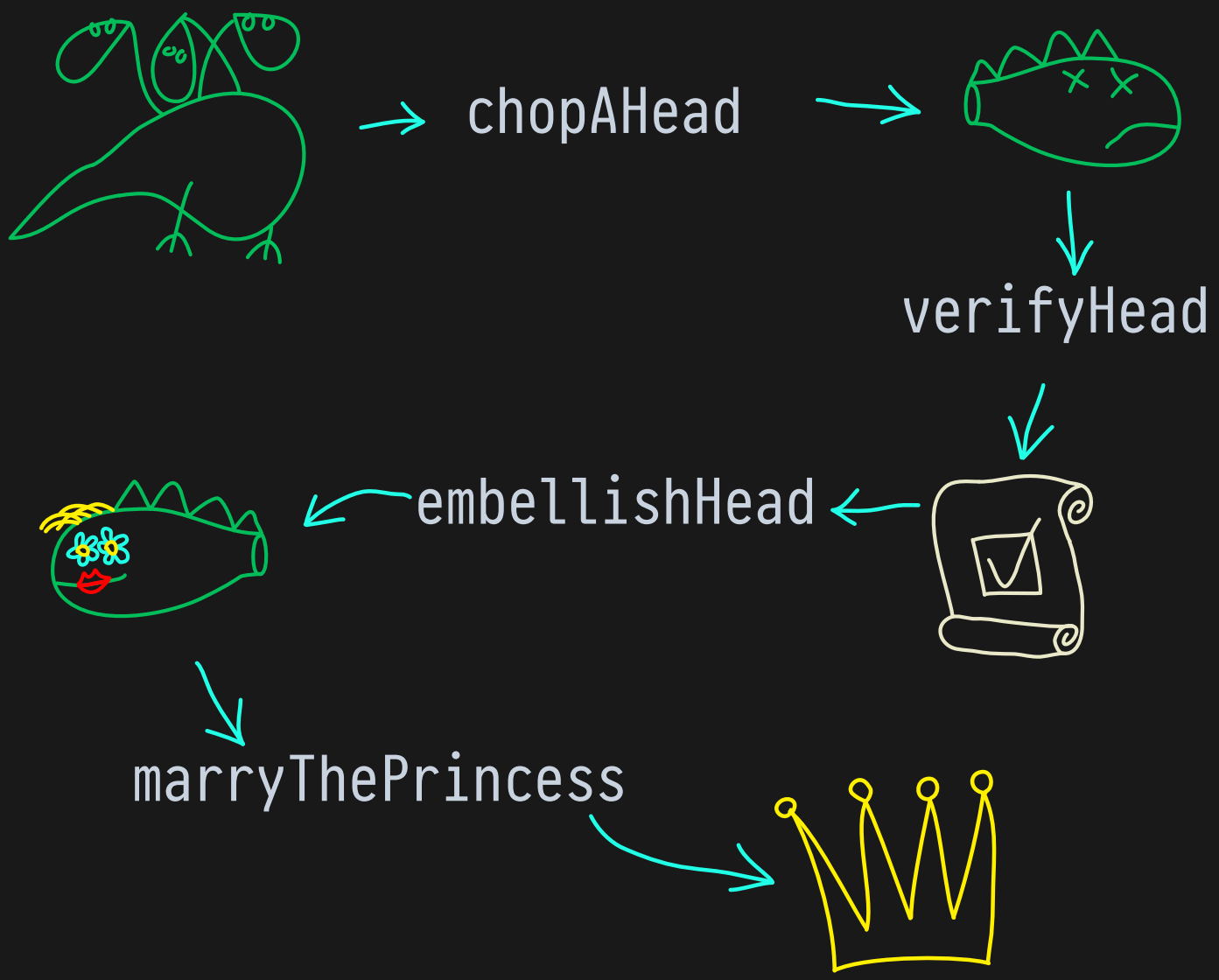
```
type landedDragon = {
  head: nonEmptyList<head>,
  health: int,
}
```

```
module Health: {
  type t
  let make: int => option<t>
  let toInt: t => int
} = {
  type t = int
  ...
}
```

```
type landedDragon = {
  heads: nonEmptyList<head>,
  health: Health.t,
  armor: Armor.t,
}
let armor: Armor.t = ...

let dragon = {
  health: armor, // error
  ...
}
```

2. ПРОЦЕССЫ (И ОШИБКИ)


chopAHead
verifyHead
embellishHead
marryThePrincess

```
dragon
  ->chopDragonHead    // ???
  ->verifyHead        // ???
  ->embellishHead     // ???
  ->marryThePrincess  // ???
```

RESULT

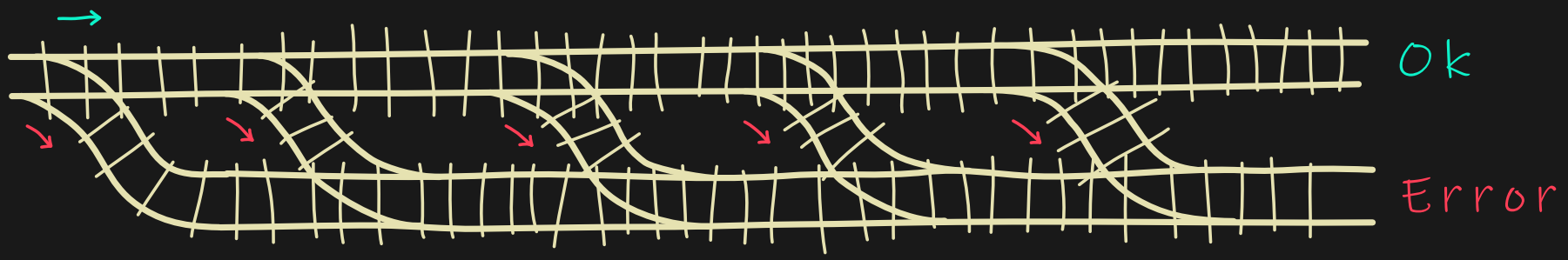
```
type result<'a, 'e> =
  | Ok('a)
  | Error('e)
```

```
switch dragon {
| Error(error) => <ErrorMessage error />
| Ok(dragon) => <DragonFight dragon />
}
```

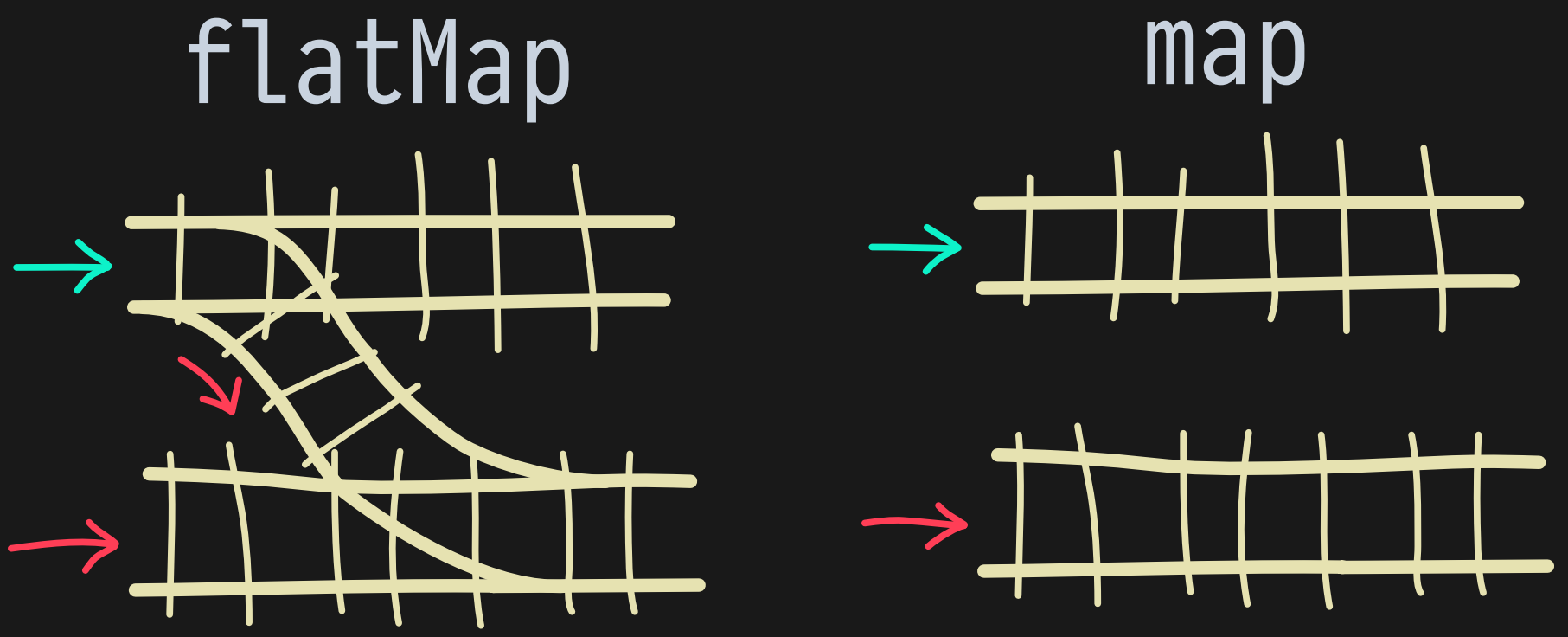
```
switch (dragon, knight) {
| (Ok(dragon), Ok(knight)) => <DragonFight dragon knight />
| (Error(error), _)
| (_, Error(error)) => <ErrorMessage error />
}
```

РЕЛЬСОВОЕ ПРОГРАММИРОВАНИЕ

(rail-oriented programming)



```
dragon
->Result.flatMap(chopDragonHead)
->Result.flatMap(Magistrate.verifyTheHead)
->Result.map(Goldsmith.embellishTheHead(gold, jewels))
->Result.map(
  embellishedHead => Church.marry(princess, embellishedHead)
)
```


flatMap
map

3. ВВОД-ВЫВОД

Hello, cruel World!

ПАРСИНГ JSON

```
type dragon = { heads: array<dragonHead> }  // 1

let dragonHeadStruct = ...                  // 2
let dragonStruct = S.object(o => {
  heads: o->S.field(
    "Heads",
    S.array(dragonHeadStruct)->S.Array.min(1)
  )
})

%raw(`{
  "Heads": [{...}, {...}, {...}],
}`)->S.parseWith(dragonStruct)      // 3: Ok(dragon)
```

```
let headsTransform = array => array->S.transform(
  ~parser=array =>
    switch array->NonEmptyList.fromArray {  // 1: из массива
    | Ok(nonEmptyList) => nonEmptyList
    | Error(_) => S.fail("The glass is too empty")
  },
  ~serializer=NonEmptyList.toArray,         // 2: в массив
  (),
)

let dragonStruct = S.object(o => {
  heads:
    o->S.field("Heads", S.array(dragonHeadStruct)->headsTransf
})
```

```
dragonJson
->S.parseWith(dragonStruct)
->Result.flatMap(chopDragonHead)
->Result.flatMap(Magistrate.verifyTheHead)
->Result.map(Goldsmith.embellishTheHead(gold, jewels))
// ...
```

«ЧТО-ТО ПОШЛО НЕ ТАК»

```
let result = try {
  Api.getDragon // эксепшон!
  ->S.parseWith(dragonStruct)
  // ...
} catch {
| Not_found => ...
| Js.Exn.Error(obj) => ...
}
```

	Result	**Исключения**
Набор значений	закрытый	скорее открытый
Поддержка системы типов	Да	Ну почти
Область применения	Бизнес-логика	Ввод-вывод

```
switch Api.getDragon {
| dragonJson => Ok(dragonJson)
| exception Js.Exn.Error(obj)
  Js.Log("что-то пошло не так")
  Error(FetchError)
}
```

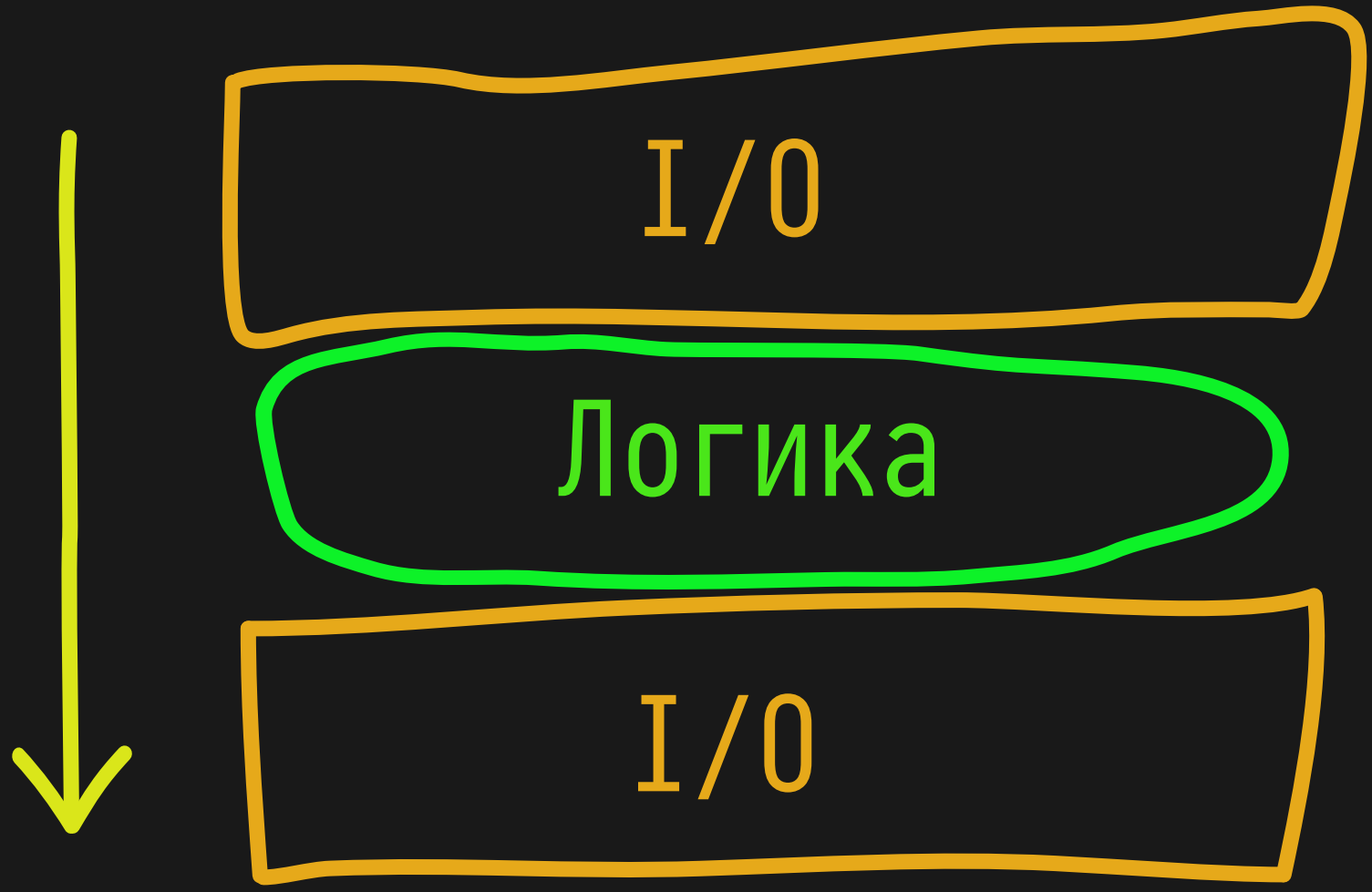
ЧТО ПО ЗАПИСИ

Да примерно то же:

- энкодинг симетричен декодингу
- с сетью/дб/файлами что-то может пойти не так

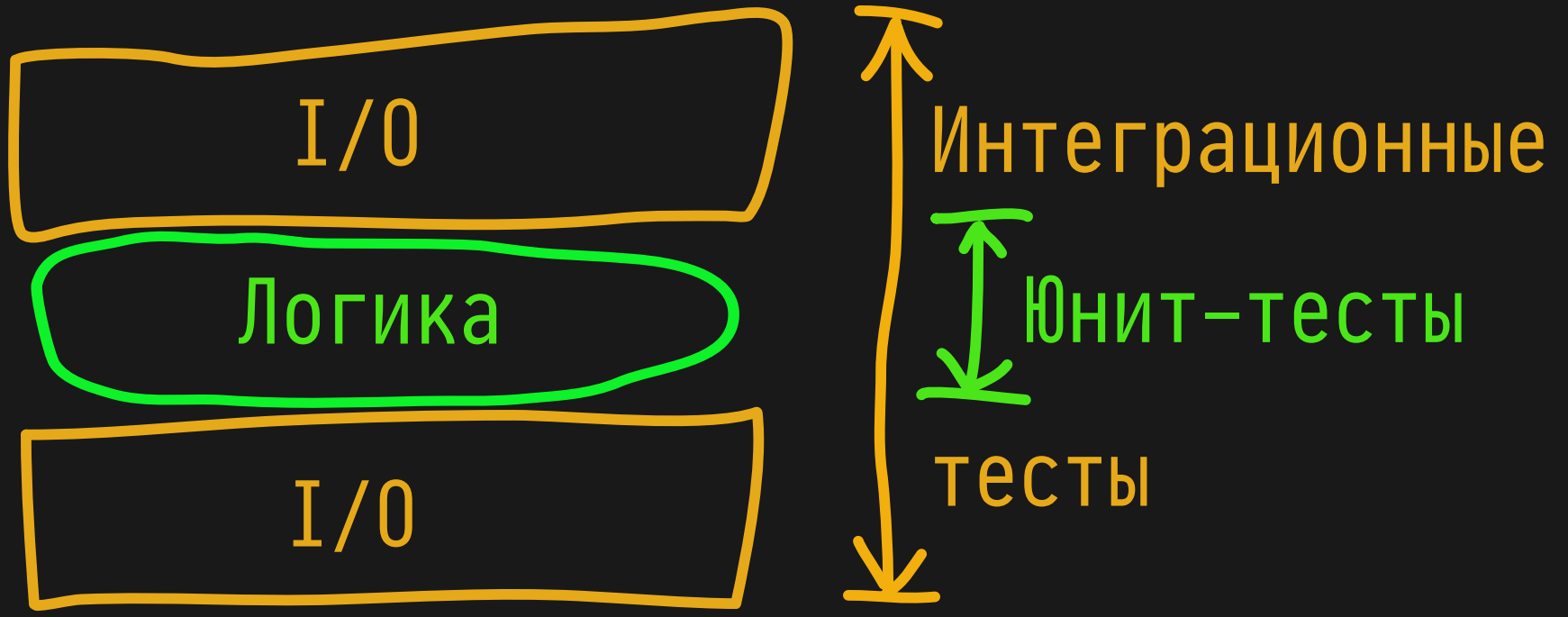
4. ТАНЦЫ ОБ АРХИТЕКТУРЕ

СЕНДВИЧ НАИЗНАНКУ


I/O
Логика
I/O

```
// грязька
Api.getDragon()
->S.parseWith(dragonStruct)
// сказька
->Result.flatMap(chopDragonHead)
->Result.flatMap(Magistrate.verifyTheHead)
->Result.map(Goldsmith.embellishTheHead(gold, jewels))
->Result.map(
    embellishedHead => Church.marry(princess, embellishedHead)
  )
// грязька
->Result.forEach(Api.liveHappilyEverAfter)
```

ТЕСТИРУЕМОСТЬ



ЛУКОВАЯ АРХИТЕКТУРА


ui
прикладные сервисы
доменные сервисы
доменная модель
db
fs
api

ФРАКТАЛЬНЫЙ ЛУК

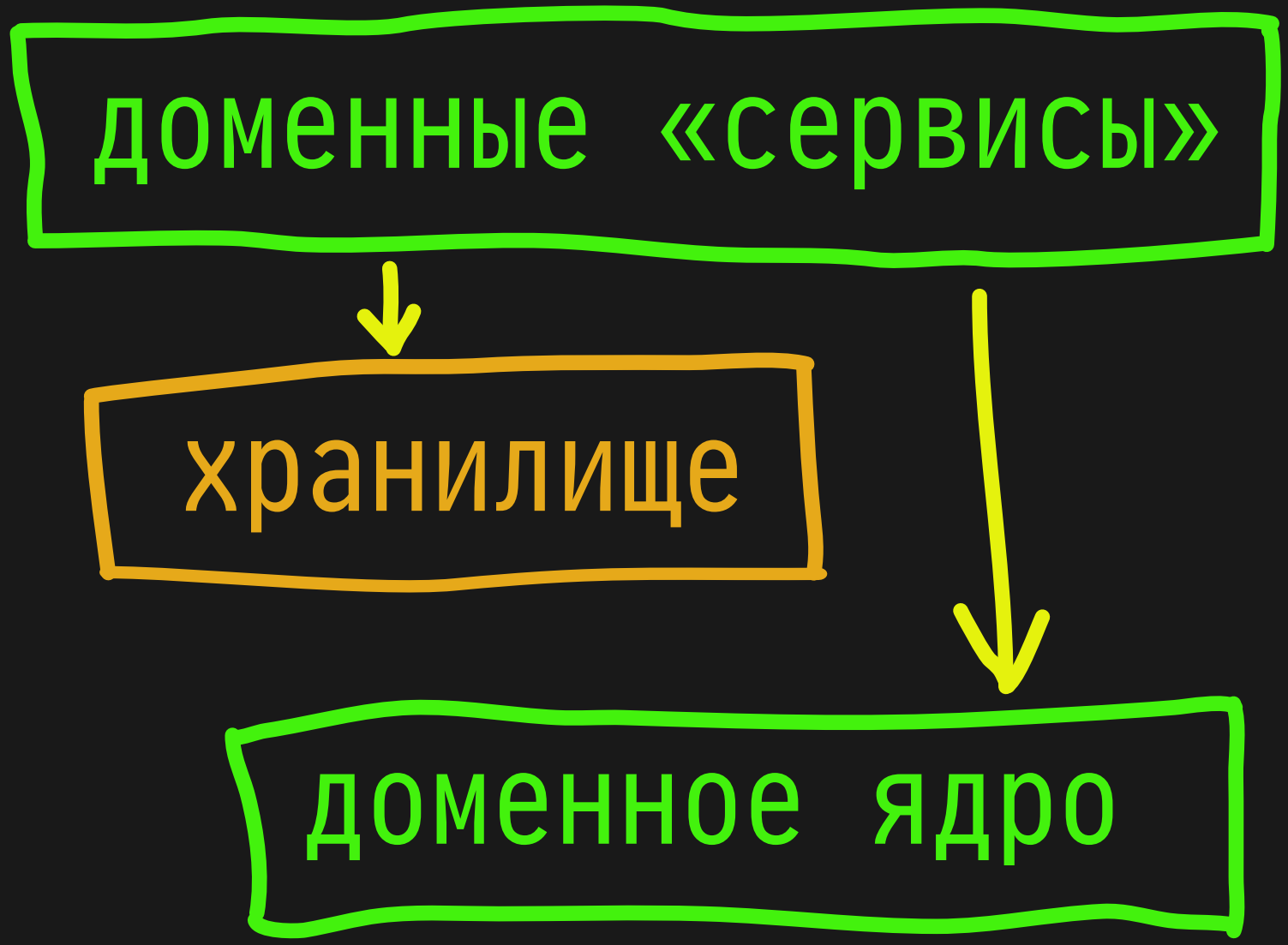
```
type remoteData<'d, 'e> = | Idle | Loading | Data('d) | Failur

switch data {
| Idle | Loading => <Loader />
| Failure(error) => <ErrorMessage error />
| Data(data) => <Page data />
}
```

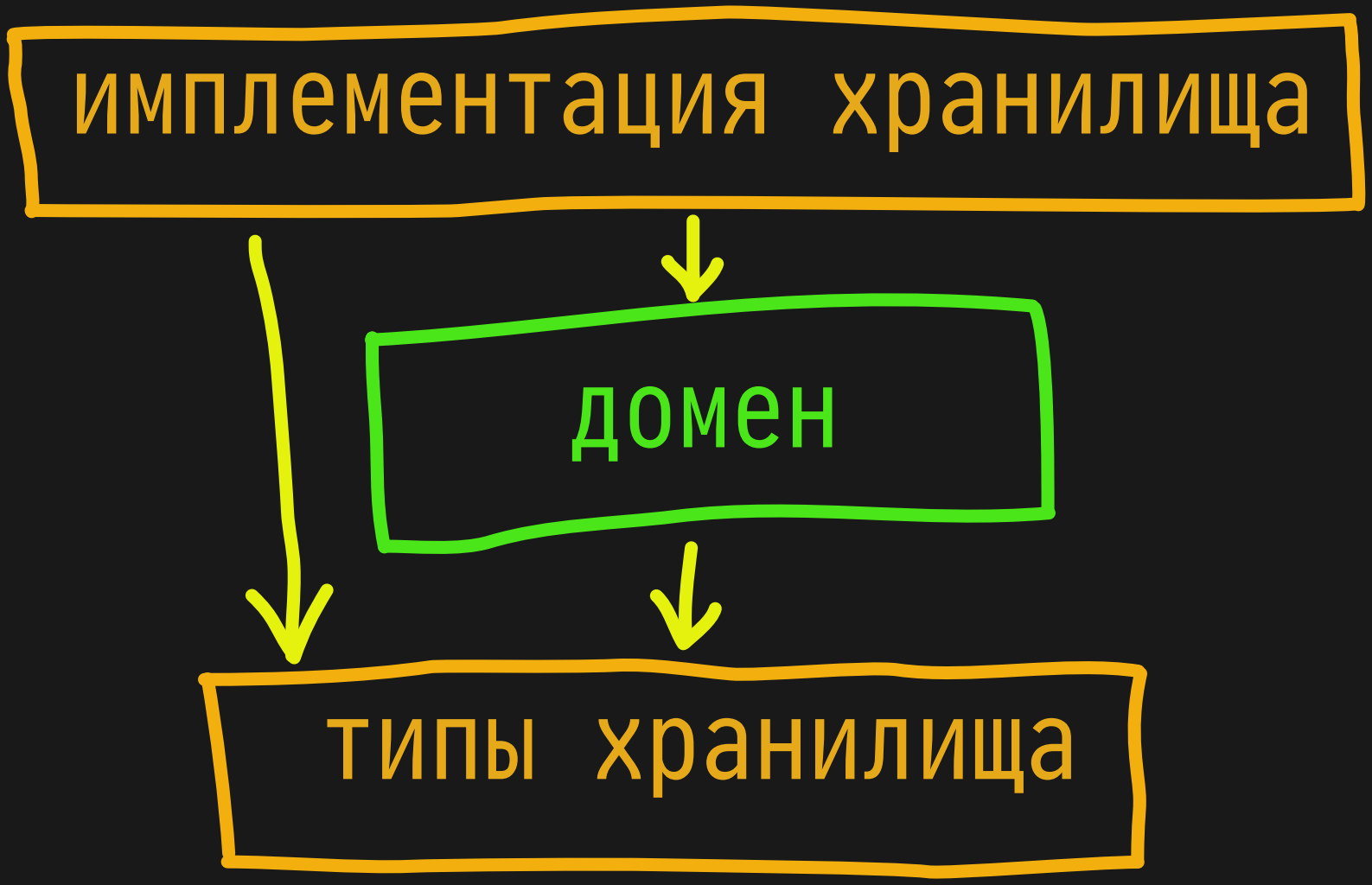

хранилище
домен

```
->Result.flatMap(chopDragonHead)
->Result.flatMap(Magistrate.verifyTheHead) // I/O->domain->I/O
->Result.map(Goldsmith.embellishTheHead(gold, jewels))
->Result.map(head => Church.marry(princess, head))
```

ВАРИАНТ 1: НЕ ЗАВИСЕТЬ ОТ ХРАНИЛИЩА


доменные «сервисы»
хранилище
доменное ядро

ВАРИАНТ 2: ЗАВИСЕТЬ ОТ ТИПОВ


имплементация хранилища
домен
типы хранилища

ВАРИАНТ 3: ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ


доменное ???
интерфейс хранилища
API
моки

СЕГОДНЯ МЫ УВИДЕЛИ, КАК

- *Моделировать/документировать* доменные сущности при помощи *типов*
- Представлять *доменные процессы* при помощи *функций*
- *Вытеснять всё сложное*/грязное/опасное из доменной области
- *Организовывать* это всё в рамках приложения/сервиса