関数型デザインとモデリングの手引き

Modeling Forum 2024
2024-12-04

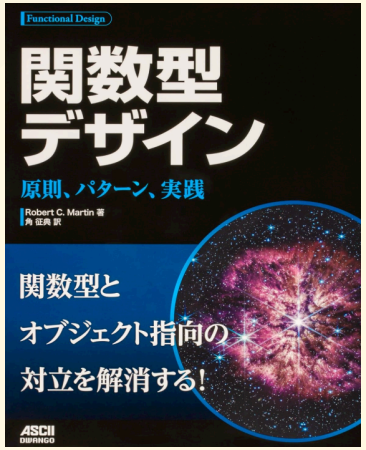
ワイクル株式会社 代表取締役
角 征典（カド マサノリ）@kdmsnr
kado.masanori@waicrew.com

スライドの場所

https://kdmsnr.com/slides

前提（1/2）

- 今回は書籍 **『関数型デザイン』** の内容を紹介
  - 著者（アンクルボブ）は関数型の専門家ではない
  - 翻訳者である私も難しい話はしません（できません💦）
    - なので、気楽に講演を聞いてください ♨️



前提（2/2）

- モデリングの話と何か関係があるのか？
  - （話題の領域が違うのよ……😿）
  - そもそもどういう状況なのか？（次ページ）

関数型とモデリングの関係性［要出典］

- 近年、オブジェクト指向では複雑さを扱えなくなってきた
- 関数型（と界隈）がそんなに怖くないことがわかってきた
- データの活用がますます重要になってきた
- 関数型的に考えたほうが便利なのかもしれない？
- ……ただぁ！
- 従来のオブジェクト指向的な考え方も役に立つと思う

今日お話すること

1. 3つのプログラミングパラダイム
2. オブジェクト指向と関数型の比較
3. オブジェクト指向と関数型の融合

1 3つのプログラミングパラダイム

3つのプログラミングパラダイム

1. 構造化プログラミング
2. オブジェクト指向プログラミング
3. 関数型プログラミング

注）実際には他にもいろいろとある。ref. Wikipedia: プログラミングパラダイム

これらはプログラマに能力を与えるのではなく、能力を**制限**している
（何をやるかではなく、**何をやらないか**を決めている）

出典：Robert C. Martin『Clean Architecture』

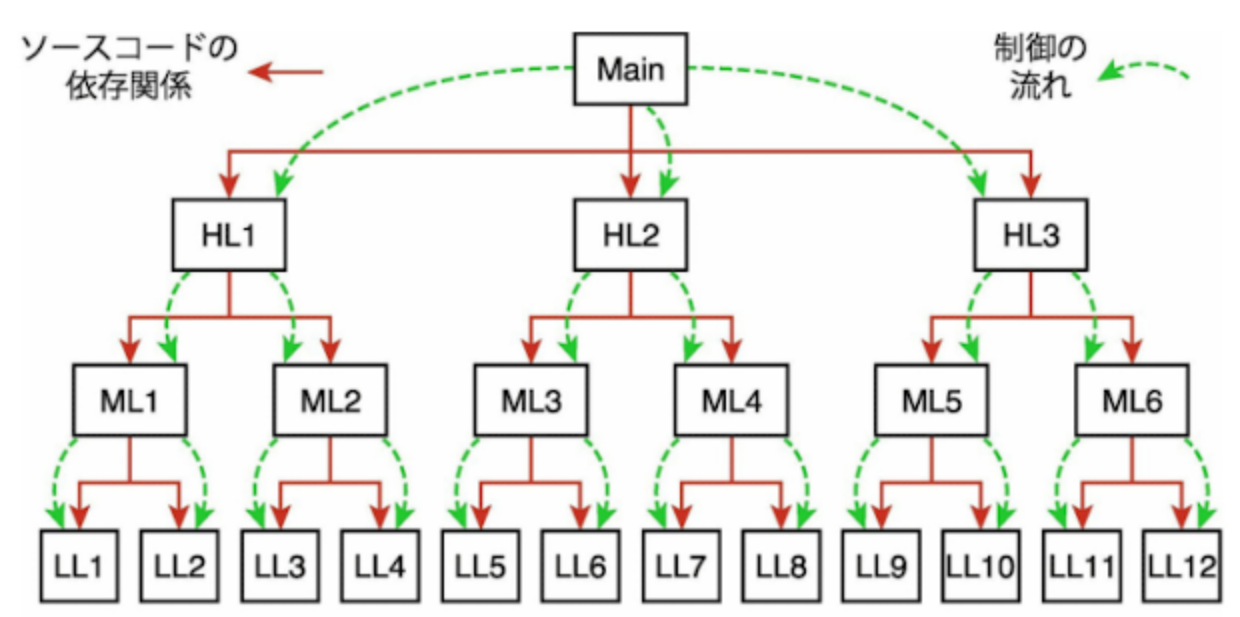
1. 構造化プログラミング

- Dijkstra「GoTo文は有害（1968）」
  - このままではプログラムをうまく分割できない
  - 順次、選択、反復だけで「構造化」すべき
- 構造化プログラミングの影響
  - プログラムをモジュールに分解して**テスト**が可能になった
  - その後、構造化設計、構造化分析へとつながる

2. オブジェクト指向プログラミング

- "SIMULA: an ALGOL-based simulation language"（1966）
- オブジェクト指向についてよく言われること：
  - カプセル化、継承、ポリモーフィズム
  - 現実世界をいい感じにモデリングできる（OOAD）
  - そもそもアラン・ケイは……（ry
    - 「`sumim + オブジェクト指向`」で検索🔎
- アンクルボブ「**ポリモーフィズムこそが重要**」と定義
  - んーどういうこと？🤔

制御の流れとソースコードの依存関係



制御の流れ（緑）は仕方ないが、依存関係（赤）は好きに決めたい！

ポリモーフィズムによる依存関係の制御

ポリモーフィズム（a.k.a. プラグインアーキテクチャ）を使えば、
ソースコードの依存関係（赤）を逆転させられる！ そうすれば、

- 依存する側［詳細］を待たずに、される側［方針］を実装できる
- それぞれを独立して開発、テスト、デプロイできる

「クリーンアーキテクチャ」も基本的にはこれだけ

3. 関数型プログラミング

- Alonzo Churchのラムダ計算（1930s）がベース
  - 変数と関数ですべてを表現する計算モデル（チューリング完全）
    - `M ::= x | (λx.M) | (M M)`
  - 「型なしラムダ計算」と「型付きラムダ計算」がある
    - おおまかに「Lisp系」と「ML系」に対応している
- アンクルボブ「**代入のないプログラミング**」と定義
  - ちょっと補足が必要……なぜ代入をなくしたいのか？

代入をなくしたい理由

- プログラムが複雑になっているのは**変数が可変**になっているから
  - 代入可能な変数 → 状態が変化 → 予測が困難
  - したがって、**変数の不変性**（単一代入）を目指せばいい
- とはいえ、すべてを不変にするのは大変すぎる 💦
  - 本書では括弧付きの **「関数型」** でヨシ！👉😸としている
  - 目の前のプログラムを動かすことを優先させたい

ref.［括弧付き］(括弧で囲われて他と区別されるところから)その語が特別の意味合いを持つこと。本来の意味とは違うこと。（精選版 日本国語大辞典）

3つのプログラミングパラダイムのまとめ

1. **構造化**：うまくモジュール化してテスト可能にすべき
2. **オブジェクト指向**：依存関係を望ましい方向へ向けるべき
3. **関数型**：システムを（できるだけ）不変にすべき

2000年代の流れ［要出典］

1️⃣ 並行処理とスケーラビリティの時代

- 2004年：CPUのデュアル化、MapReduceの論文発表
- 2005年：『ハッカーと画家』→ 日本でLispに注目が集まる
- 2006年：『入門Haskell』と『ふつうのHaskellプログラミング』
- 2007年：『Programming Erlang』
- 2009年：『Programming Scala』→ TwitterがScalaを採用開始
- 2009年：Clojureリリース

関数型でうまいことやりたいが、まだまだ難しい時代

2010年代以降の流れ［要出典］

2️⃣ マルチパラダイムの普及

- 2012年：「DSが21世紀で最もセクシーな職業」、TypeScript
- 2010年代：Python + Pandas によるデータ分析（Rも再注目）
- 2013年：Reactリリース
- 2014年：Java 8がラムダ式とStream APIを導入、Swiftリリース
- 2017年：KotlinがAndroid公式言語に
- 2020年代：Rustの台頭、関数型UI、ストリームデータ処理

他の分野でも関数型が求められる（当たり前になってきた）時代

2 オブジェクト指向と関数型の比較

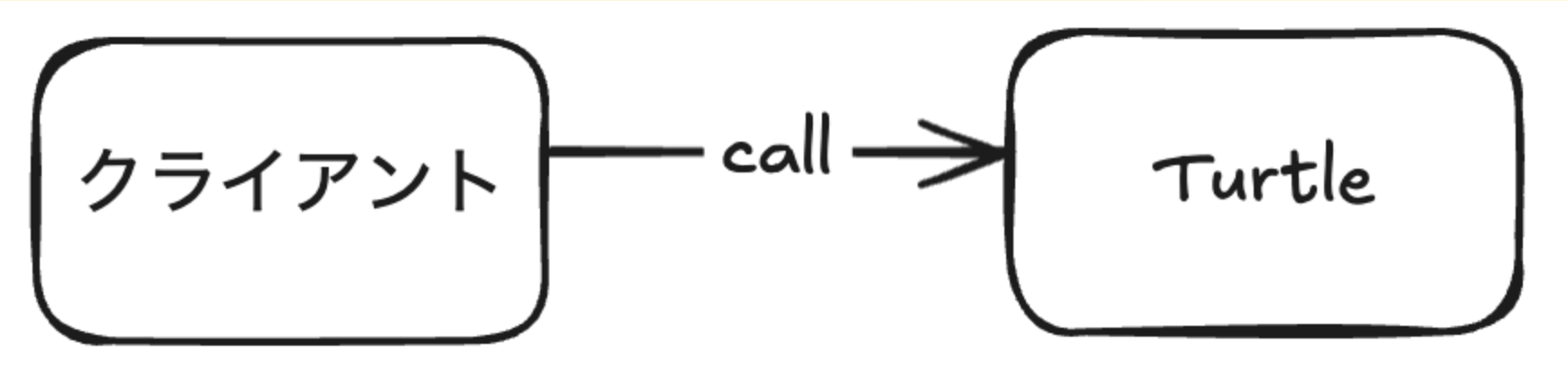
例題1「タートルグラフィックス」

タートル（亀）の仕様：

- 現在の方向へ移動する。 `Move(distance)`
- 時計回りまたは反時計回りに回転する。 `Turn(angle)`
- ペンを上げ下げする。 `PenDown` / `PenUp`
  - なお、ペンが下にあるときは、カメを移動すると線を引く。

ref. 『関数型デザイン』第14章

オブジェクト指向の場合



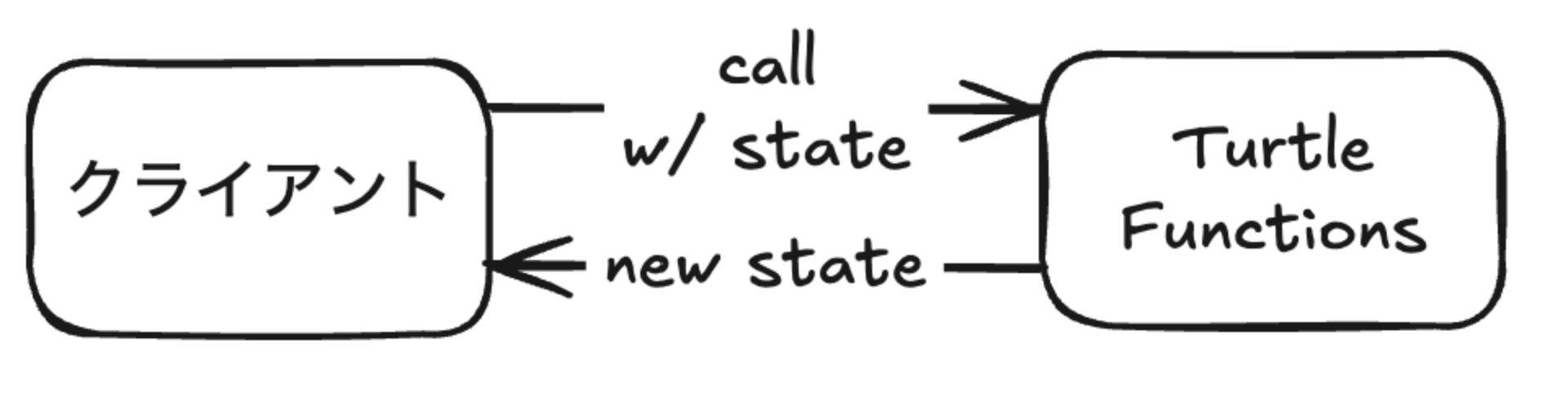
- Turtleのオブジェクトが状態を持つ（位置、角度、ペン）
- クライアントがメソッド呼び出しで状態を操作する

オブジェクト指向のサンプルコード（Ruby）

```ruby
class Turtle
  attr_accessor :position, :angle, :pen_state
  def move(distance); end
  def turn(angle); end
  def pen_down; end
end
```

```ruby
turtle = Turtle.new
turtle.pen_down
turtle.turn(180.0)
turtle.move(10.0);
```

関数型の場合



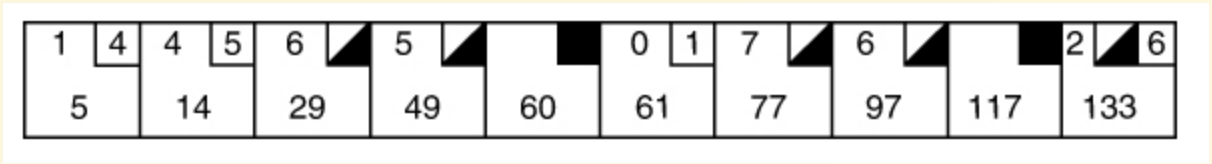
- Turtleを状態を持たない関数にする
- 関数を呼び出すときには、状態をデータとして渡す
- 状態Aを渡したときは、常に新しい状態Bが戻るようにする

関数型のサンプルコード（Ruby）

```ruby
TurtleState = Struct.new(:position, :angle, :pen_down) # データ構造
module TurtleFunctions  # 各関数は新しい State を戻す
  def self.move(state, distance); end
  def self.turn(state, angle);end
  def self.pen_down(state); end
end
```

```ruby
state0 = TurtleState.new([0, 0], 0, false)
state1 = TurtleFunctions.pen_down(state0)
state2 = TurtleFunctions.turn(state1, 180.0)
state3 = TurtleFunctions.move(state2, 10.0)
```

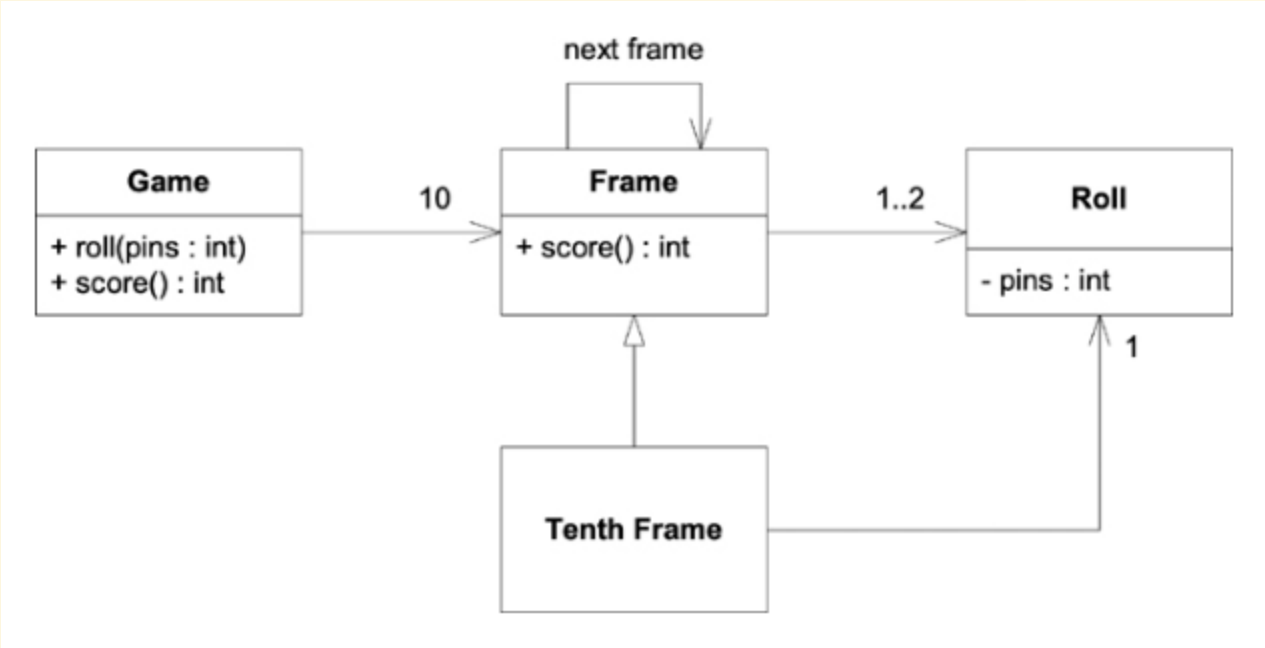
例題2「ボウリングゲーム」



- そのフレームがストライクならば、
  スコアは10点と次の投球2回分の得点である
- そのフレームがスペアならば、
  スコアは10点と次の投球1回分の得点である
- それ以外ならば、
  スコアはそのフレームの投球2回分の得点である

出典：Robert C. Martin『Clean Craftsmanship』

オブジェクト指向の場合



※ 10フレーム目は2投目までに10本になったときだけ3投目がある

※ 書籍ではTDDで開発しているのでUMLとは違った実装になっている

出典：Robert C. Martin『Clean Craftsmanship』

｜関数型の場合（1）

投球データをリストにして、計算すればいいんじゃないの？

```
[10 7 3 5 4 10 9 0]
```

※ 長くなるので、5フレーム分だけ用意した

参考：Robert C. Martin『関数型デザイン』

｜関数型の場合（2）

フレーム単位に整形してみよう。

```
[[10] [7 3] [5 4] [10] [9 0]]
```

各フレームにボーナスを追加しよう。

```
[[10 7 3]  ;; フレーム1: ストライク + ボーナス(7, 3)
 [7 3 5]   ;; フレーム2: スペア + ボーナス(5)
 [5 4]     ;; フレーム3: 通常フレーム
 [10 9 0]  ;; フレーム4: ストライク + ボーナス(9, 0)
 [9 0]]    ;; フレーム5: 通常フレーム
```

｜関数型の場合（3）

で、最後に合計すればよさそう

```
(reduce + 0 (flatten frames)) ;; framesが先程のデータ
```

ポイント：

- 関数型のバージョンにはクラスがない（当たり前）→ データ重視
- 途中の「整形」の処理は大変かもしれない
- でも、各処理はうまく分離されていて、見通しが良さそう

例題3「ゴシップ好きのバスの運転手」

もう少し難しそうな例を考えてみる：

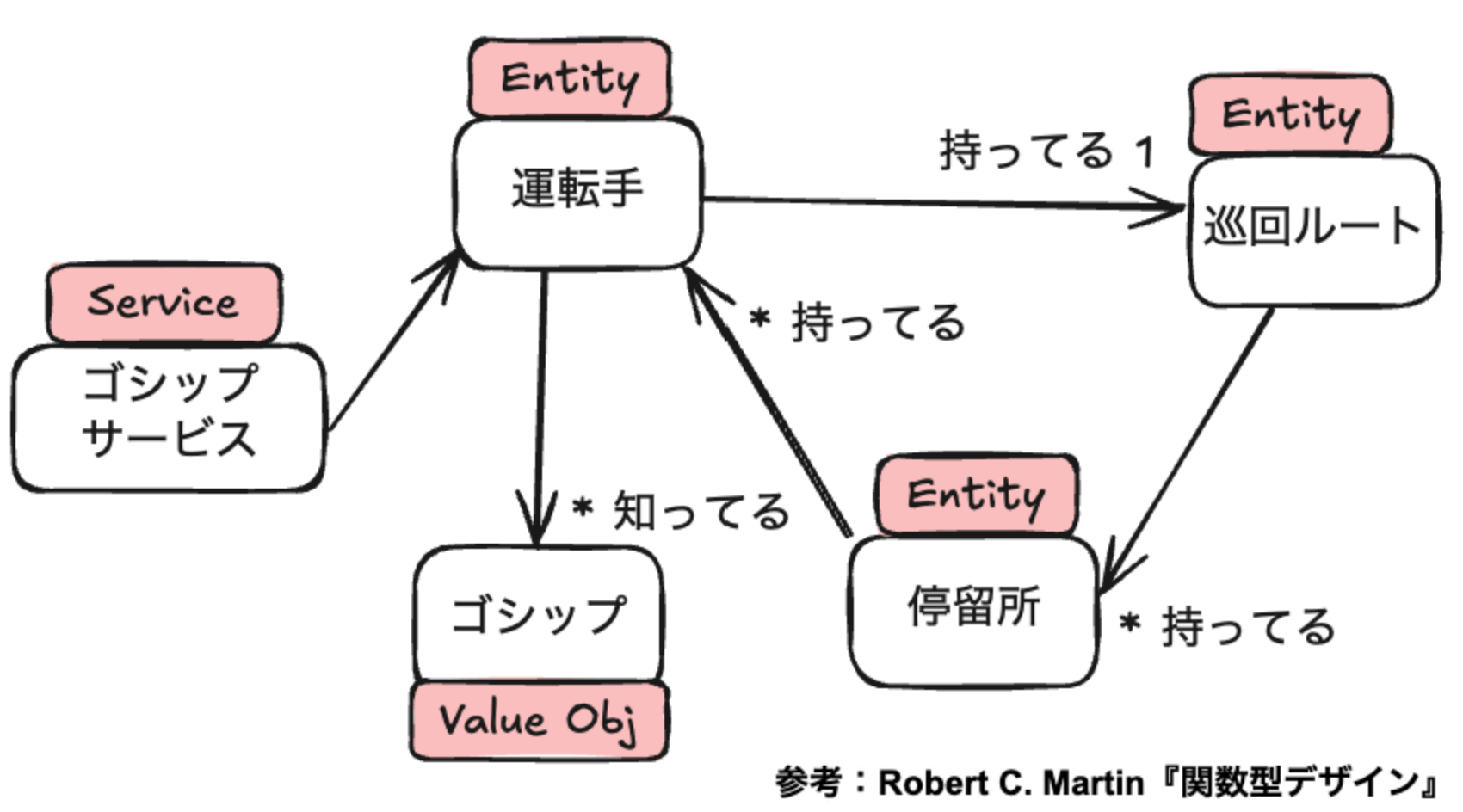
- バスの**運転手**は別々の**巡回ルート**を担当している。
- 巡回ルートは複数の**停留所**を巡回する。
- バスの運転手はいくつかの**ゴシップ**を知っている。
- 運転手は1分ごとに停留所を移動する。1日の労働時間は8時間。
- 同じ停留所に停車したら、他の運転手とゴシップを共有する。
- **Q. すべての運転手がすべてのゴシップを知るまでの分数は？**

ref. 『関数型デザイン』第8章 and https://kata-log.rocks/gossiping-bus-drivers-kata

オブジェクト指向の場合

- 問題文からクラス図を作る
- クラス図をコードに置き換える
- 使い方を考えながら、各クラスのメソッドの配置を調整する
  - このあたりからTDDでやることが多いかも

問題文からクラス図（っぽいもの）を作る



クラス図をコードに置き換える（Ruby）

```ruby
class Driver; attr_accessor :route, :gossips; end
class Route; attr_accessor :stops; end
class Stop; attr_accessor :drivers; end
class Gossip; end

class GossipService
  def run
    ...
  end
end
```

｜使い方を考える（サービスクラス）

- サービスクラスは太りがちなので気を付ける

```ruby
class GossipService
  def run
    (60 * 8).times do |min|
      @drivers.each{|d| d.drive} # 次の停留所に進める(&:driveでも可)
      stops = []; @drivers.each{|d| stops << d.current_stop} # 本来は.mapを使ってもいい
      stops.each{|st| st.exchange_gossips} # 停留所でゴシップを共有する
      return min + 1 if all_gossips_shared? # 全て共有されたら終了
    end
    "not shared gossips"
  end
```

停留所クラス

```ruby
class Stop
  attr_accessor :drivers

  def exchange_gossips # みんなでゴシップを共有する
    gossips = @drivers.flat_map(&:gossips).uniq
    @drivers.each {|d| d.add_gossips(gossips)}
  end

  def leave(driver); @drivers.delete(driver); end
  def arrive(driver); @drivers << driver; end
```

運転手クラス

```ruby
class Driver
  attr_accessor :route, :gossips, :stop_number

  def drive # 次の停留所に移動する
    current_stop.leave(self)
    @stop_number = route.next_stop_number(@stop_number)
    current_stop.arrive(self)
  end

  def current_stop; @route.get_stop(@stop_number); end
  def add_gossips(gossips); (@gossips += gossips).uniq!; end
```

巡回ルートクラス

```ruby
class Route # 重要なロジックは特にない
  attr_accessor :stops

  def get_stop(stop_number)
    @stops[stop_number]
  end

  def next_stop_number(current_stop_number)
    (current_stop_number + 1) % @stops.length
  end
```

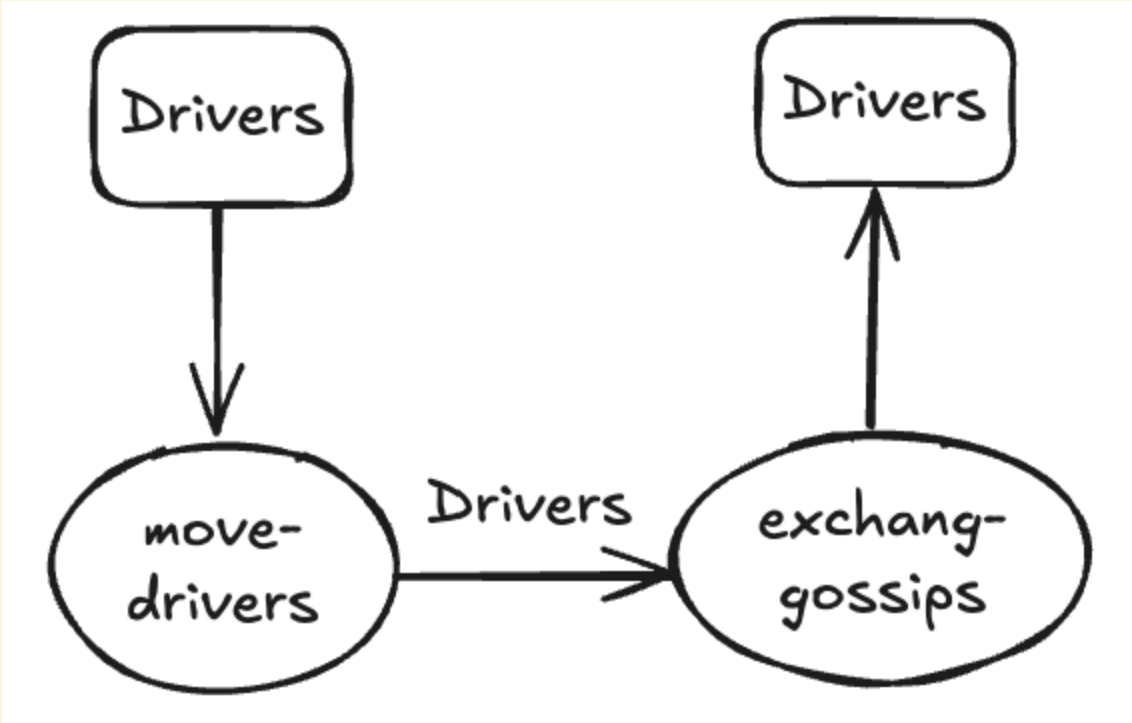
関数型の場合

- モデルを描く
- 小さな関数をたくさん作る
- 関数をうまく組み合わせる
- データ構造はシンプルにする（できれば標準データ構造）
- データは不変にする（常に新しいデータを生成する）
- 反復の代わりに再帰を使う
- できるだけ標準の関数でデータを操作する

※順不同。思いついた順番で書き連ねた。

｜モデル（の代わりにDFDっぽいもの）を描く

- **運転手**を「運転手の移動」と「ゴシップの共有」に渡せばいい？



｜小さな関数をたくさん作る（Clojure）

- 「運転手の移動」関数

```
(defn move-driver [driver])

(defn move-drivers [drivers]
  (map move-driver drivers))
```

- 「ゴシップの共有」関数

```
(defn exchange-gossips [drivers])
```

関数をうまく組み合わせる

- 「運転手の移動」と「ゴシップの共有」の関数をまとめる

```
(defn drive [drivers]
  (-> drivers                ;; スレッディングマクロ。UNIXのパイプ処理みたいな感じ。
      move-drivers
      exchange-gossips))
```

データ構造はシンプルにする

- 運転手はマップでいいや
  - 名前は文字列、ルートはベクター、ゴシップはセット（集合）

```
{:name "Bob" :route [:p :q :r] :gossips #{:X}}
```

- 生成用の小さな関数も用意する

```
(defn make-driver [name route gossips]
  (assoc {} :name name :route (cycle route) :gossips gossips))
```

データは不変にする

- たとえば「運転手の移動」では、新しい運転手を作って戻す

```
(defn move-driver [driver]
  (update driver :route rest))
```

データ構造はシンプルにする（2）

- 次に「ゴシップの共有」を実装したい
- その前に、**同じ停留所にいる運転手のデータ**が欲しい
  - たとえば、こんな感じにシンプルだと嬉しい

```
{
 :s1 [driver1 driver3],   ;; 停留所s1に driver1 と driver3 がいる
 :s2 [driver2]            ;; 停留所s2に driver2 がいる
}
```

反復の代わりに再帰を使う

- 先ほどのデータを戻す小さな関数を作る

```clojure
(defn get-stops [drivers]
  (loop [drivers drivers
         stops {}]
    (if (empty? drivers)
      stops
      (let [driver (first drivers)
            stop (first (:route driver))  ;; 運転手の現在の停留所を取得
            stops (update stops stop conj driver)]  ;; 停留所に運転手を追加
        (recur (rest drivers) stops)))))  ;; 残りの運転手でループを続行
```

できるだけ標準の関数でデータを操作する

- 停留所と運転手のデータができたので「ゴシップの共有」を実装
- 標準的な関数（`map`, `flatten`）を活用した関数にする

```
(defn exchange-gossips [drivers]
  (let [stops-with-drivers (get-stops drivers)
        drivers-by-stop (vals stops-with-drivers)]
    (flatten (map (fn [drivers]
                    (let [gossips (map :gossips drivers)
                          all-gossips (apply set/union gossips)]
                      (map #(assoc % :gossips all-gossips) drivers)))
                  drivers-by-stop))))
```

最後にサービス用の関数を作る

- すべての運転手が同じゴシップを持つまで再帰する

```
(defn run [drivers]
  (loop [drivers (drive drivers)
         time 1]
    (cond
      (> time 480) :never   ;; 480分を超えたら終了
      (apply = (map :gossips drivers)) time   ;; 全運転手が同じゴシップを持ったら終了
      :else (recur (drive drivers) (inc time)))))   ;; 繰り返し
```

オブジェクト指向と関数型の比較のまとめ

オブジェクト指向：

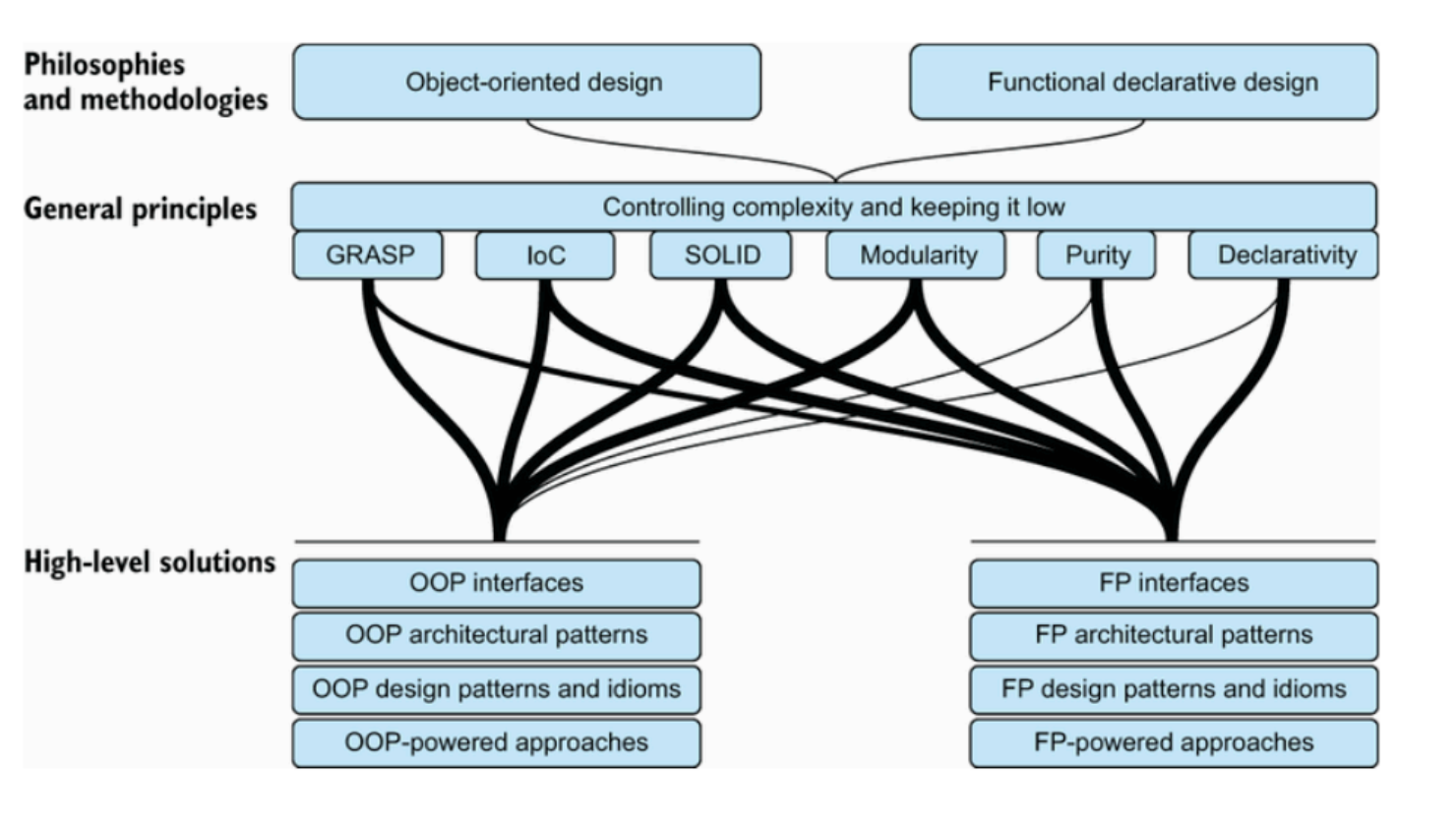
- モジュール分割は**ドメインの用語**に基づく
- 状態は**可変のオブジェクト**が保持し、オブジェクト内部で更新する

関数型：

- モジュール分割は**振る舞い**に基づく
- 状態は**不変のデータ構造**が保持し、各関数で新しい状態を生成する

3 オブジェクト指向と関数型の融合

目指すところは同じ → 複雑性に対応



Source: Alexander Granin『Functional Design and Architecture』Manning

1 SOLID

- SOLID: アンクルボブが提唱する設計原則
- オブジェクト指向の原則と思われていたが、実は普遍的な原則
  - これまでのやり方を適用できる！

SRP: 単一責任の原則

- モジュールはひとつの責任を持つ……わけではない（名前が悪い）
- モジュールは**特定のアクターに責任を持つ**
  - 同じ理由で同じ時期に変更されるものはまとめておく
  - 違う理由で違う時期に変更されるものは分離しておく
- モジュールの種類は異なるが（OOPはクラス、関数型は関数）、モジュールに分割する原則は同じものが使える

OCP: オープン・クローズドの原則

- 拡張にはオープンで、修正にはクローズである
  - 言い換えれば、**既存のコードを修正せずに機能を変更したい**
- そのためには、抽象化の仕組みを使って変更を制御する
  - OOP: **ポリモーフィズム**でクラスを入れ替える
  - 関数型: いくつかの方法があるが、いずれも**関数**を使用する
- 拡張する部分（or 修正しない部分）を決めるのはどちらも同じ

LSP: リスコフの置換原則

- OCPをサポートする原則
- モジュールの置き換えを可能にする
  - 型、引数、戻り値、例外処理などを等価にする
- **OOPも関数型もまったく同じ**

ISP: インターフェイス分離の原則

- 利用者が必要としないものに依存させない
  - OOP：インターフェイスを小さく分割する
  - 関数型：小さな関数を作る
    - なお、**部分適用**や**関数合成**で必要な機能を構成できる
- どちらも考え方は同じだが、関数型のほうが柔軟に対応可能

DIP: 依存関係逆転の原則

- 上位レベル（抽象）は下位レベル（具象）に依存しない
  - OOP: ポリモーフィズムやDIを使用する
  - 関数型: 高階関数などで振る舞いを抽象化する

SOLIDのまとめ

- SOLIDは普遍的な原則
  - SRPは構成の原動力
  - OCPは道徳心
  - LSPとISPは不注意で生じた穴を囲む警告サイン
  - DIPはその他の原則の基盤
- 具体的なやり方は違っても考え方は転用できる

2️⃣ Purity（純粋性）

- 純粋性……とは？
  - 同じインプットから同じアウトプット（参照透過性）
  - 外部の状態が変化しない（副作用がない）
- 純粋性のメリット
  - 予測しやすい、テストしやすい、並行処理しやすい
- したがって、純粋なものとそうじゃないものを分けるべき

『脳に収まるコードの書き方』

(cont.)『脳に収まるコードの書き方』

まえがきをアンクルボブが書いている（彼のシリーズなので）。

> **マーク・シーマン**のことは何年も前から知っています。(snip) 彼とはいろいろなところで**意見が合いません**。(snip) でも、マークと違う意見を持つのは、とても慎重にならなければいけません。というのも、彼の議論の論理は非の打ち所がないからです。
>
> *出典：『脳に収まるコードの書き方』（まえがき）*

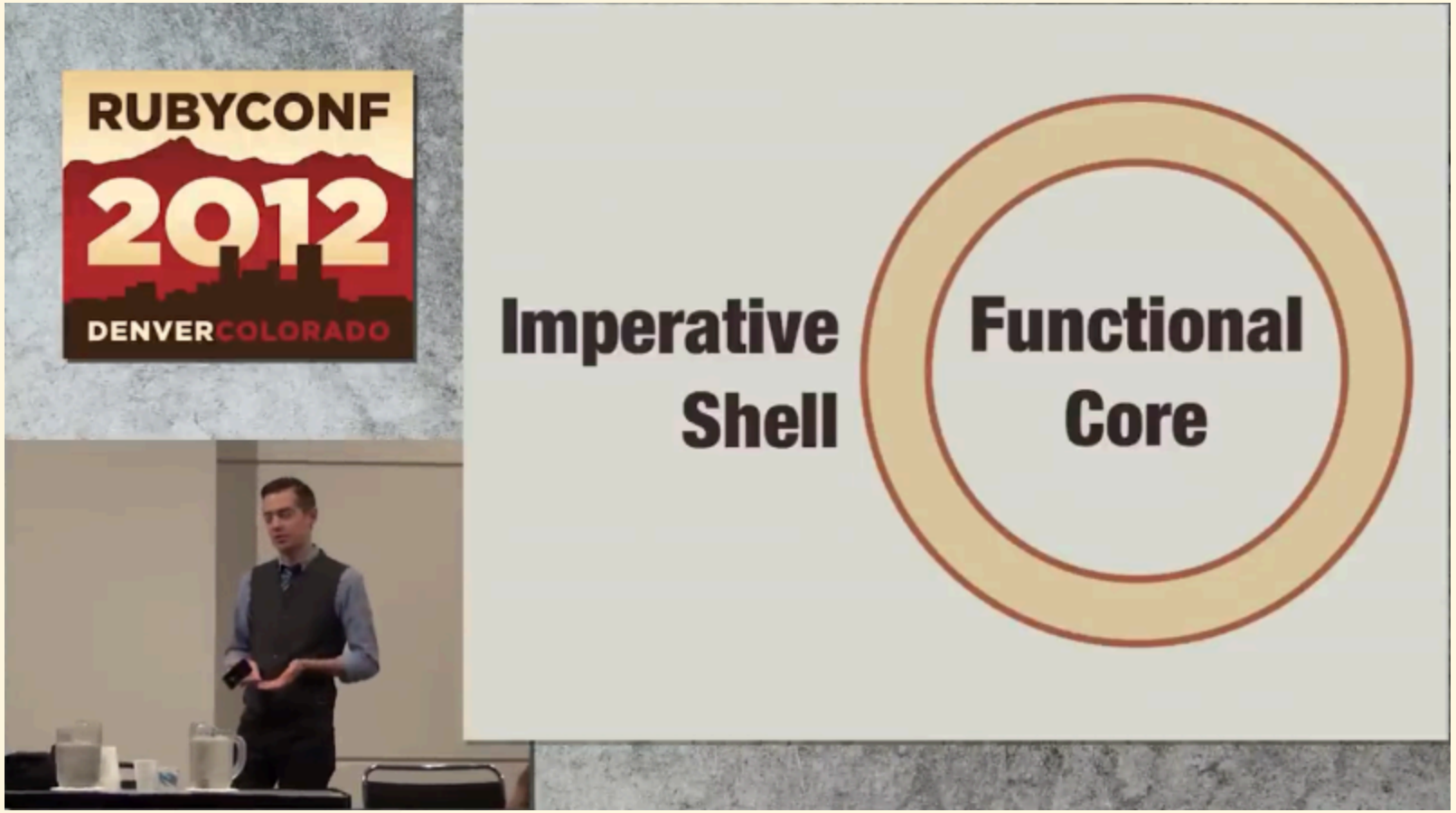
関数型コア・命令型シェル

> 私は、いわゆるオブジェクト指向のコードベースを捨ててHaskellに移行することを組織に期待しているわけではありません。**関数型コア・命令型シェルへの移行を推奨しています**。関数型コア・命令型シェルによって、コードベースで純粋でない操作を実装している箇所を簡単に独立させられます。
>
> *出典：『脳に収まるコードの書き方』（13章 関心事の分離）*

関数型コア・命令型シェルとは

- 関数型と命令型を組み合わせた設計アプローチ
  - by Gary Bernhardt at Ruby Conf 2012
    - https://www.youtube.com/watch?v=yTkzNHF6rMs
- 基本的な考え方：
  1. **関数型コア**: 重要な部分（ビジネスルールなど）を表現
  2. **命令型シェル**: 関数型を囲み、外部とのやりとりを処理

これが「純粋なものとそうじゃないものを分ける方法」になる


RUBYCONF
2012
DENVERCOLORADO
Imperative
Shell
Functional
Core

「タートルグラフィックス🐢」の場合

操作は「関数型コア」にまとめる。状態は持たない。

```ruby
module TurtleCore
  def self.move(state, distance)
    new_position = change_position(state, distance)
    state.merge(position: new_position)
  end

  def self.turn(state, angle)
    state.merge(angle: state[:angle] + angle)
  end
end
```

(cont.)「タートルグラフィックス🐢」の場合

状態を持つ部分は「命令型シェル」のオブジェクトにする。

```
class TurtleShell
  def initialize
    @state = { position: [0, 0], angle: 0, pen_state: false }
  end
  def move(distance)
    @state = TurtleCore.move(@state, distance)
  end
  def turn(angle)
    @state = TurtleCore.turn(@state, angle)
  end
```

「ゴシップ好きのバスの運転手🚌」の場合

関数型コアの部分

- `move-driver` 関数：運転手を次の停留所に移動させる
- `exchange-gossips` 関数：停留所でゴシップを交換する

命令型シェルの部分

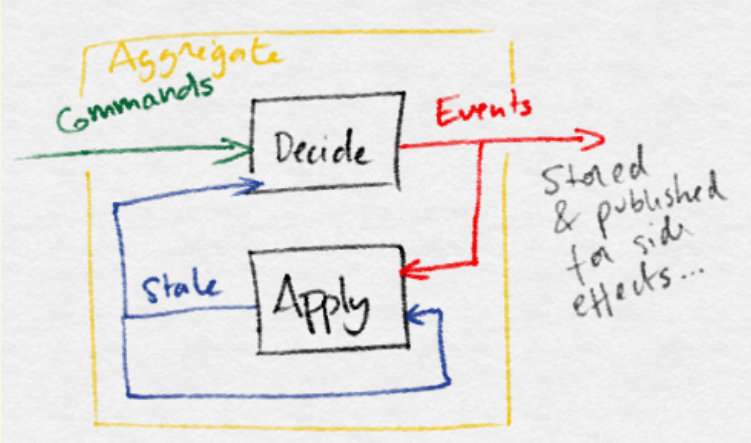
- `run` 関数：時間の経過を計算して、何らかのUI等に出力する
  - （サンプルコードにはその部分は書いていないけど）

前に見た図に（なんとなく）似ている

- オブジェクト指向と同じものを関数型でも目指している！
- 中央部分は**関数型コア**、両サイドは**命令型シェル**に対応する
  - 中央部分は関数型なので、テストが容易になる
  - 副作用がある部分は両サイドに寄せる

EventSourcingとDeciderパターン

- EventSourcingはイベントの履歴から状態を作る手法
- Deciderはコマンドと状態からイベントを作る（関数コア）
- 状態の計算は大変なのでApply（or Evolve）関数に任せる



Source: https://thinkbeforecoding.com/post/2014/01/04/Event-Sourcing.-Draw-it

3 型（Type）

『関数型デザイン』はあまり型については触れていないが、同時期に同じ出版社から出た『関数型ドメインモデリング』があるので**省略**

全体のまとめ

1. 3つのプログラミングパラダイム
    - 構造化、オブジェクト指向、関数型
2. オブジェクト指向と関数型の比較
    - オブジェクト指向：データ（ドメイン）中心、可変状態
    - 関数型：振る舞い中心、不変データ
3. オブジェクト指向と関数型の融合
    - SOLID原則は普遍的に使える
    - 関数型コア・命令型シェルがちょうどいい妥協案
    - 型については他の書籍を参考に