Declarative Axiomatic and Provable Correct Systems

In Swift

Manuel Meyer, vikingosegundo@gmail.com, 8th of November 2022

Quiz

What is Declarative Code?

Describes what you want achieve, not how

Resembles human communication more closely (“please open the door” not “stand up, make a step, make another step,…, raise your arm,…”)

By far most codes are actually declarative

Declarative Coding offers a much higher degree of alignment between Coder and Compiler, allowing “brain-sized coding”

Simplicity vs Complexity

Demo

- Complexity leads to unpredictable behaviour
- Unpredictable behaviour leads to errors
- Most teams try to fight those errors by endlessly adding more complexity
- But: Complexity leads to unpredictable behaviour
- Unpredictable behaviour leads to errors
- …

- Avoid complexities and truly embrace simplicity
- You won’t fight complexities with more complexities and win
- “Code is where the bugs live” — excessive amounts of codes will lead to excessive amounts of defects

The only solution against complexity is simplicity

- Simple != Easy
- Immutability is simpler than mutability
- Unidirectional flow is simpler than multidirectional flow
- Classes are inherently complex and should not be found at the center of systems

Declarative Domain Paradigm

What is a Paradigm?

- An idea on how to organise code in respect to certain constrains
- The understanding that unconstraint jumps complicate things lead to the Structured Programming Paradigm (“Goto Considered Harmful”)
- OO and FP both deal with mutable global state, OO by hiding mutable state, FP by maintaining immutable global state

«Declarative Domain Paradigm»

- Uses Partial Application instead of classes to constrain the system
- Behaviour and Intent are expressed as data (“Domain Specific Languages” DSLs) and decoded by pattern matching
- All model and state are immutable

Why not rely on Classes?

- “Spiderman Principle”: With great power comes great responsibility
- Classes are too powerful, they allow for many different kinds of interaction
- They do not work well as black boxes
    - For sublassing often additional information beyond the class’s signatures, like in what order to call super implementations or which methods need to be subclassed together — information unavailable to the compiler
    - They have no notion of an “output” and there for need a whole zoo of patterns

Why not rely on Classes?

- Classes provide and promote unconstraint coding where the opposite is required: Constraints like immutability and true black boxing might seem restrictive, but they are actually helpful.
- Civil Engineering is guided by limitations imposed by physics and its laws. Software Engineers must provide their own constraints.

Why not rely on Classes?

Real world engineers have to keep physics in mind and use it in their creations

Software engineers have to define the “physics” of apps

Partial Application

```
func createAdder(x:Int) -> (Int) -> Int  {
    var value = x
    return {
        value = value + $0
        return value
    }
}

let adder = createAdder(x:1)
print(adder(4))
print(adder(1))
```

Partial Application

```swift
typealias Stack<T> = (push:(T) -> (), pop:() -> (T?))
func createStack<T>() -> Stack<T> {
    var array:[T] = [];
    return (push: { array.append($0) },
            pop: { array.count > 0 ? array.remove(at: array.count - 1) : nil })
}
let s:Stack<Int> = createStack()
s.push(1)
s.push(2)
print(String(describing:s.pop())) // 2
print(String(describing:s.pop())) // 1
print(String(describing:s.pop())) // nil
```

Partial Application

```swift
typealias Access<S> = (                      ) -> S  // get current state
typealias Change<C> = ( C...                 ) -> () // change state by applying Change C+
typealias Reset     = (                      ) -> () // reset to defaults
typealias Updated   = ( @escaping () -> () ) -> () // subscribe for updates
typealias Destroy   = (                      ) -> () // destroy persistent data

typealias Store<S,C> = ( /* S:State, C:Change */
      state: Access<S>,
     change: Change<C>,
      reset: Reset,
    updated: Updated,
    destroy: Destroy
)
// ————————
func createDiskStore(
    pathInDocs   p: String ,
    fileManager fm: FileManager = .default
) -> Store<AppState, AppState.Change>
{
    var state = loadAppStateFromStore(pathInDocuments:p,fileManager:fm) { didSet { callbacks.forEach { $0() } } }
    var callbacks: [() -> ()] = []
    return (
        state   : { state },
        change  : { state     = state.alter($0)  ; persistStore(pathInDocuments:p, state:state, fileManager:fm) },
        reset   : { state     = AppState()       ; persistStore(pathInDocuments:p, state:state, fileManager:fm) },
        updated : { callbacks = callbacks + [$0] },
        destroy : { destroyStore(pathInDocuments:p,fileManager:fm) }
    )
}
```

Immutable Datatypes

What cannot change on purpose
also cannot change by accident —
eliminating whole classes of possible bugs and defects.

```swift
struct CoffeeTruck: Identifiable {
    enum Change {
        case opening(It)
        case closing(It); enum It{ case it}
        case renaming(Renaming)
        case setting (Setting)
        enum Setting {
            case location    (_Location   ); enum _Location { case to(Location) }
            case openingTimes(OpeningTimes)
        }
        enum Renaming {
            case it(To); public enum To { case to(String) }
        }
    }
    let id      : UUID
    let name    : String
    let location: Location
    let open    : Bool
    let times   : OpeningTimes

    init(name n:String) { self.init(UUID(),n,.unknown,false, OpeningTimes())

    func alter(by changes:Change...) -> Self { changes.reduce(self) { $0.alter(by:$1) } }
    private func alter(by change :Change   ) -> Self {
       switch change {
        case     .closing(.it)              : return Self(id,name,location,false,times)
        case     .opening(.it)              : return Self(id,name,location,true ,times)
        case let .renaming(.it(.to(n)))     : return Self(id,n,   location,open ,times)
        case let .setting(.location(.to(l))): return Self(id,name,l       ,open ,times)
        case let .setting(.openingTimes(t)) : return Self(id,name,location,open ,t    )
        }
    }
}
```

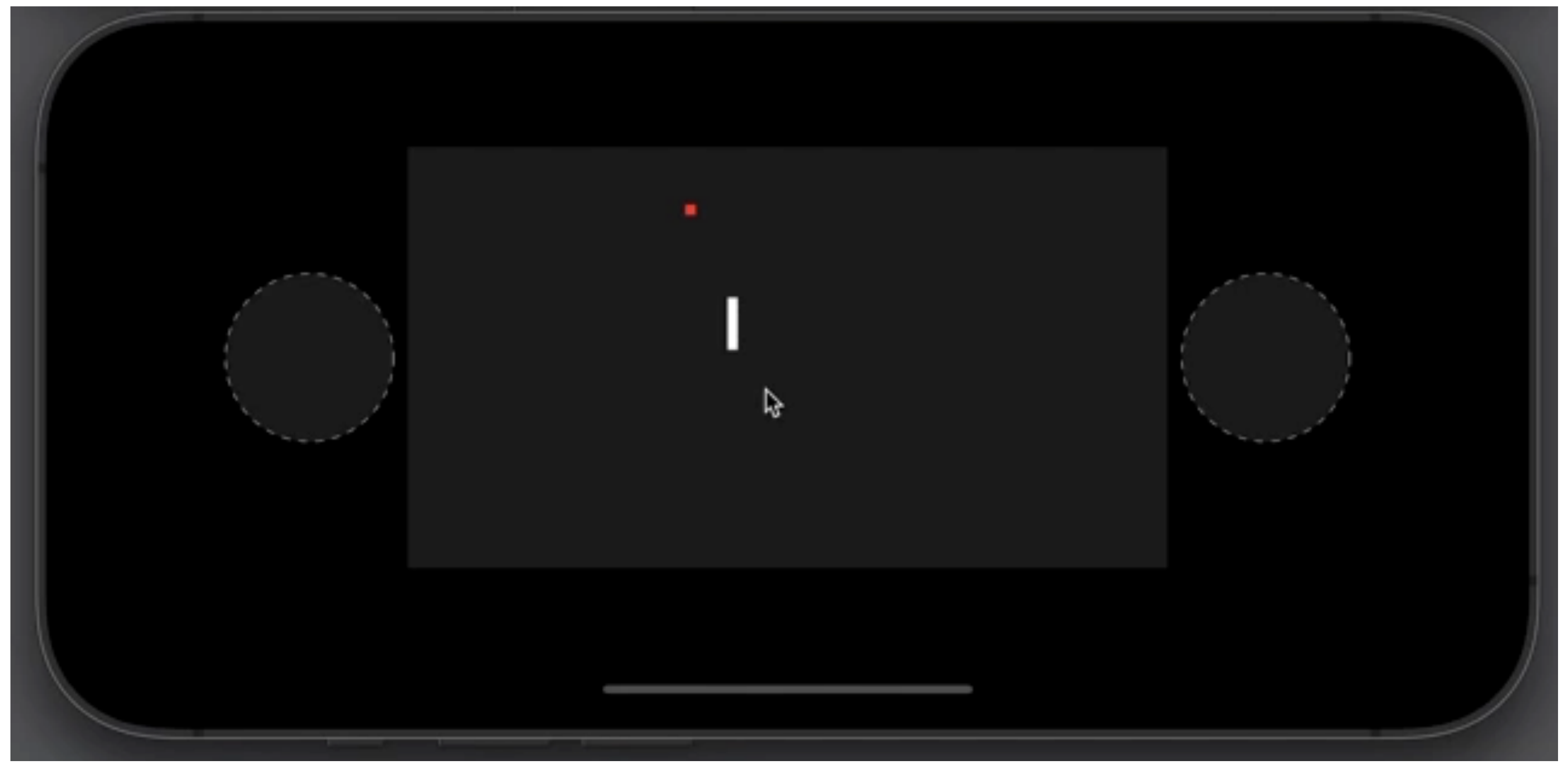
```swift
let t0 = CoffeeTruck(name:"Coffee Truck")
let t1 = t0.alter(by:.opening(.it))
let t2 = t1.alter(by:.closing(.it))
let t3 = t2.alter(by:.renaming(.it(.to("Chez Edsger"))))
           .alter(by:.setting(
                         .location(
                           .to(.coordinate(
                              .init(latitude :52.356389,
                                    longitude: 4.951944)))
```

```swift
extension CoffeeTruck {
    public enum Location {
        case unknown
        case coordinate(Coordinate)
        case address(Address)
    }
    public struct Coordinate {
        public init(latitude lat:Double,longit
            latitude = lat
            longitude = lon
        }
        public let latitude : Double
        public let longitude: Double
    }
    public struct Address {
        public init(street s:String,zipCode z:
        public let street : String
        public let zipCode: String?
        public let country: String?
    }
}
```

```swift
struct AppState:Codable {
    enum Change {
        case setting(Setting)
        enum Setting {
            case current(Truck)
            enum Truck {
                case truck(to:CoffeeTruck?)
            }
        }
    }
    let truck:CoffeeTruck?
    init() { self.init(nil) }
    func alter(by changes: Change...) -> Self { changes.reduce(self) { $0.alter(by: $1) } }
    func alter(by changes:[Change]  ) -> Self { changes.reduce(self) { $0.alter(by: $1) } }
    private
    func alter(by change : Change    ) -> Self {
        switch change {
        case .setting(.current(.truck(to:let t))): return .init(t)
        }
    }
    private init(_ t:CoffeeTruck?) { truck = t }
}
```

```swift
let appState = AppState()
let t = CoffeeTruck(name:"Chez Edsger")
let appState = appState
                   .alter(
                      by:.setting(
                           .current(
                              .truck(to:t))))
```

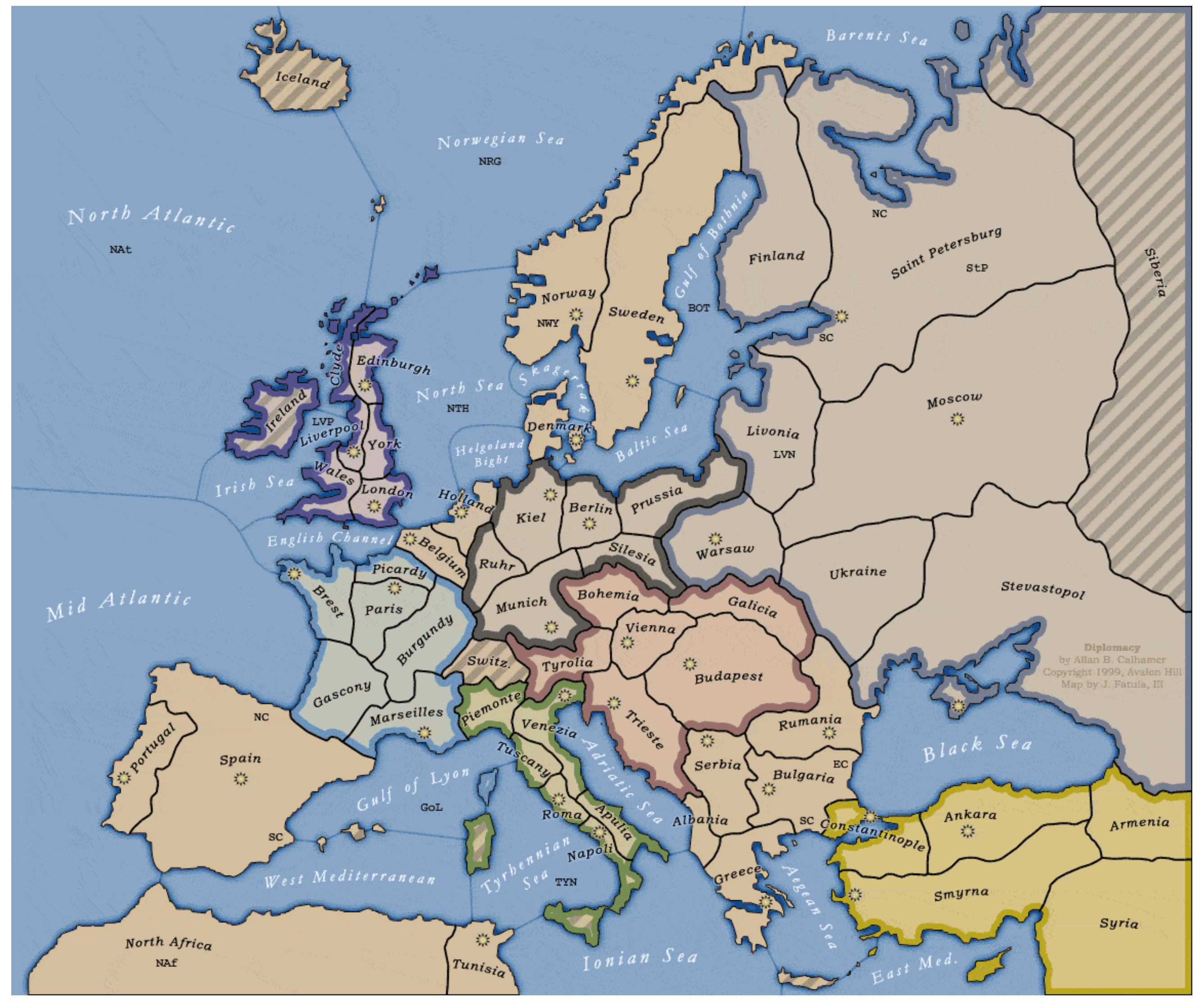
```swift
struct Snake {
    init (head:Coordinate) { self.init(head, [], .north) }

    enum Facing { case north, east, south, west }
    enum Move   { case forward, right, left      }
    enum Change { case move(Move), grow          }

    let head  : Coordinate
    let tail  : [Coordinate]
    var body  : [Coordinate] { [head] + tail }
    let facing: Facing

    func alter(_ change:Change) -> Self {
        func growTail() -> [Coordinate] { tail.last != nil ? tail + [tail.last!] : [head] } // grow by appending last element again
        switch change {
        case let .move(direction): return move (direction)
        case     .grow:            return .init(head, growTail(), facing)
        }
    }
    private init(_ h:Coordinate, _ t:[Coordinate], _ f:Facing) { head = h; tail = t; facing = f }
    private func move(_ m:Move) -> Self {
        func newSnakeTail() -> [Coordinate] { Array(([head] + tail).prefix(tail.count)) }
        switch (m, facing) {
        case (.forward, .north): return .init(.init(x:head.x,     y:head.y - 1), newSnakeTail(), .north)
        case (.forward, .east ): return .init(.init(x:head.x + 1, y:head.y    ), newSnakeTail(), .east )
        case (.forward, .south): return .init(.init(x:head.x,     y:head.y + 1), newSnakeTail(), .south)
        case (.forward, .west ): return .init(.init(x:head.x - 1, y:head.y    ), newSnakeTail(), .west )
        case (   .left, .north): return .init(.init(x:head.x - 1, y:head.y    ), newSnakeTail(), .west )
        case (   .left, .east ): return .init(.init(x:head.x,     y:head.y - 1), newSnakeTail(), .north)
        case (   .left, .south): return .init(.init(x:head.x + 1, y:head.y    ), newSnakeTail(), .east )
        case (   .left, .west ): return .init(.init(x:head.x,     y:head.y + 1), newSnakeTail(), .south)
        case (  .right, .north): return .init(.init(x:head.x + 1, y:head.y    ), newSnakeTail(), .east )
        case (  .right, .east ): return .init(.init(x:head.x,     y:head.y + 1), newSnakeTail(), .south)
        case (  .right, .south): return .init(.init(x:head.x - 1, y:head.y    ), newSnakeTail(), .west )
        case (  .right, .west ): return .init(.init(x:head.x,     y:head.y - 1), newSnakeTail(), .north)
        }
    }
}
```



```swift
enum Empire {
    case britain, france, germany, russia, ottoman, austria, italy
}
enum Unit:Equatable {
    case army (Empire)
    case fleet(Empire)
}
struct Territory {
    init(name:String, kind:Kind, occupiedBy:Unit? = nil) {
        self.name = name
        self.kind = kind
        self.occupiedBy = occupiedBy
    }
    enum Kind { case land, sea }
    enum CoastKind {
        case unified
        case fragmented([(String, [Territory])])
    }
    enum Border {
        case land   (Territory, Territory)
        case coastal(Territory, Territory, Territory.CoastKind)
        case sea    (Territory, Territory)
        case channel(Territory)
    }
    let name:String
    let kind: Kind
    let occupiedBy:Unit?
    enum Change { case occupy(Unit?) }
    func execute(_ change: Change) -> Self {
        switch change {
        case .occupy(let unit): return .init(name: name, kind: kind, occupiedBy: unit)
        }
    }
}
```

```swift
final class Board {
    var territories: [Territory]
    let borders    : [Territory.Border]
    enum Change {
        case move(Move)
        case hold(Territory, with:Unit, of:Empire)
        enum Move{
            case  army(of:Empire, from:Territory, to:Territory)
            case fleet(of:Empire, from:Territory, to:Territory)
        }
    }
    init(territories t:[Territory], borders b:[Territory.Border]) { territories = t; borders = b }
    func execute(command:Change)  {
        switch command {
        case let .move( .army(empire, from:from, to:to)): move(unit: .army(empire),from:from,to:to)
        case let .move(.fleet(empire, from:from, to:to)): move(unit:.fleet(empire),from:from,to:to)
        case let .hold(territory, with:unit,  of:empire): print("hold \(territory.name) with:\(unit) of
        }
    }
    private func move(unit:Unit, from:Territory, to:Territory)  {
        switch (unit, from.kind, to.kind) {
        case let ( .army(empire),.land,.land): connected(left:from,right:to) ? moveArmy (for:empire,from
        case let (.fleet(empire), .sea, .sea): connected(left:from,right:to) ? moveFleet(for:empire,from
        default: ()
        }
    }
}
```

```swift
board.execute(command:.move(.army(of:.germany,from:berlin,  to:silesia       )))
board.execute(command:.move(.army(of:.germany,from:silesia, to:galicia       )))
board.execute(command:.move(.army(of:.germany,from:galicia, to:rumania       )))
board.execute(command:.move(.army(of:.germany,from:rumania, to:bulgaria      )))
board.execute(command:.move(.army(of:.germany,from:bulgaria,to:constantinople)))
```



```swift
struct Light:Codable, Equatable, Identifiable {
    enum Change {
        case renaming(_RenameIt); enum _RenameIt { case it     (to:String)    }
        case turning (_TurnIt  ); enum _TurnIt   { case it     (Turn)         }
        case adding  (_Add     ); enum _Add      { case mode   (Light.Mode)   }
        case toggling(_Toggle  ); enum _Toggle   { case display(to:Interface) }
        case setting (_Set     ); enum _Set      {
            case hue        (to:Double      )
            case saturation (to:Double      )
            case brightness (to:Double      )
            case temperature(to:Temperature)
        }
    }
    init(id:String, name:String) { self.init(id, name, .off, 0, 0, 0, 0, .slider, [], .unset) }

    enum Mode:Codable { case unset, hsb, ct }
    enum Temperature { case mirek(Int) }

    let id        : String
    let name      : String
    let isOn      : Turn

    let hue       : Double
    let saturation: Double
    let brightness: Double
    let ct        : Int

    let modes       : [Mode]
    let selectedMode: Mode
    let display     : Interface

    public func alter(by changes:Change...) -> Self { changes.reduce(self) { $0.alter($1) } }

    private init(_ x:String,_ n:String,_ o:Turn,_ b:Double,_ s:Double,_ h:Double,_ ct:Int,_ d:Interface,_ m:[Mode],_ y:Mode) { id = x,name = n,isOn = o,hue = h,saturation = s,brightness =
    private func alter(_ change:Change) -> Self {
        func __(_ x: Double) -> Double { min(max(x, 0.0), 1.0) }
        func __(_ ct:Int) -> Int { min(max(ct, 153), 500)}
        switch change {
        case let .renaming(         .it(to:name      )): return .init(id, name, isOn,   brightness,   saturation,   hue,   ct,    display, modes,          selectedMode)
        case     .turning (         .it(.on          )): return .init(id, name, .on,     brightness,   saturation,   hue,   ct,    display, modes,          selectedMode)
        case     .turning (         .it(.off         )): return .init(id, name, .off,    brightness,   saturation,   hue,   ct,    display, modes,          selectedMode)
        case let .setting ( .brightness(to:brightness)): return .init(id, name, isOn,__(brightness),  saturation,   hue,   ct,    display, modes,          selectedMode)
        case let .setting ( .saturation(to:saturation)): return .init(id, name, isOn,   brightness,__(saturation),  hue,   ct,    display, modes,          .hsb        )
        case let .setting (        .hue(to:hue       )): return .init(id, name, isOn,   brightness,   saturation,__(hue),  ct,    display, modes,          .hsb        )
        case let .setting (.temperature(to:.mirek(ct))): return .init(id, name, isOn,   brightness,   saturation,   hue,__(ct),   display, modes,          .ct         )
        case     .toggling(    .display(to:.slider   )): return .init(id, name, isOn,   brightness,   saturation,   hue,   ct,    .slider, modes,          selectedMode)
        case     .toggling(    .display(to:.scrubbing)): return .init(id, name, isOn,   brightness,   saturation,   hue,   ct, .scrubbing, modes,          selectedMode)
        case let .adding  (       .mode(mode         )): return .init(id, name, isOn,   brightness,   saturation,   hue,   ct,    display, modes + [mode], selectedMode)
        }
    }
}
```

```swift
Light(id:"01", name:"01").alter(
        by:
            .renaming(.it(to:"Turingzaal 1")),
            .setting (.brightness (to:0.5   )),
            .setting (.hue        (to:0.5   )),
            .setting (.saturation (to:0.5   )),
            .setting (.temperature(to:200.mk)),
            .turning (.it         (   .on   ))
    )
```

Assignment

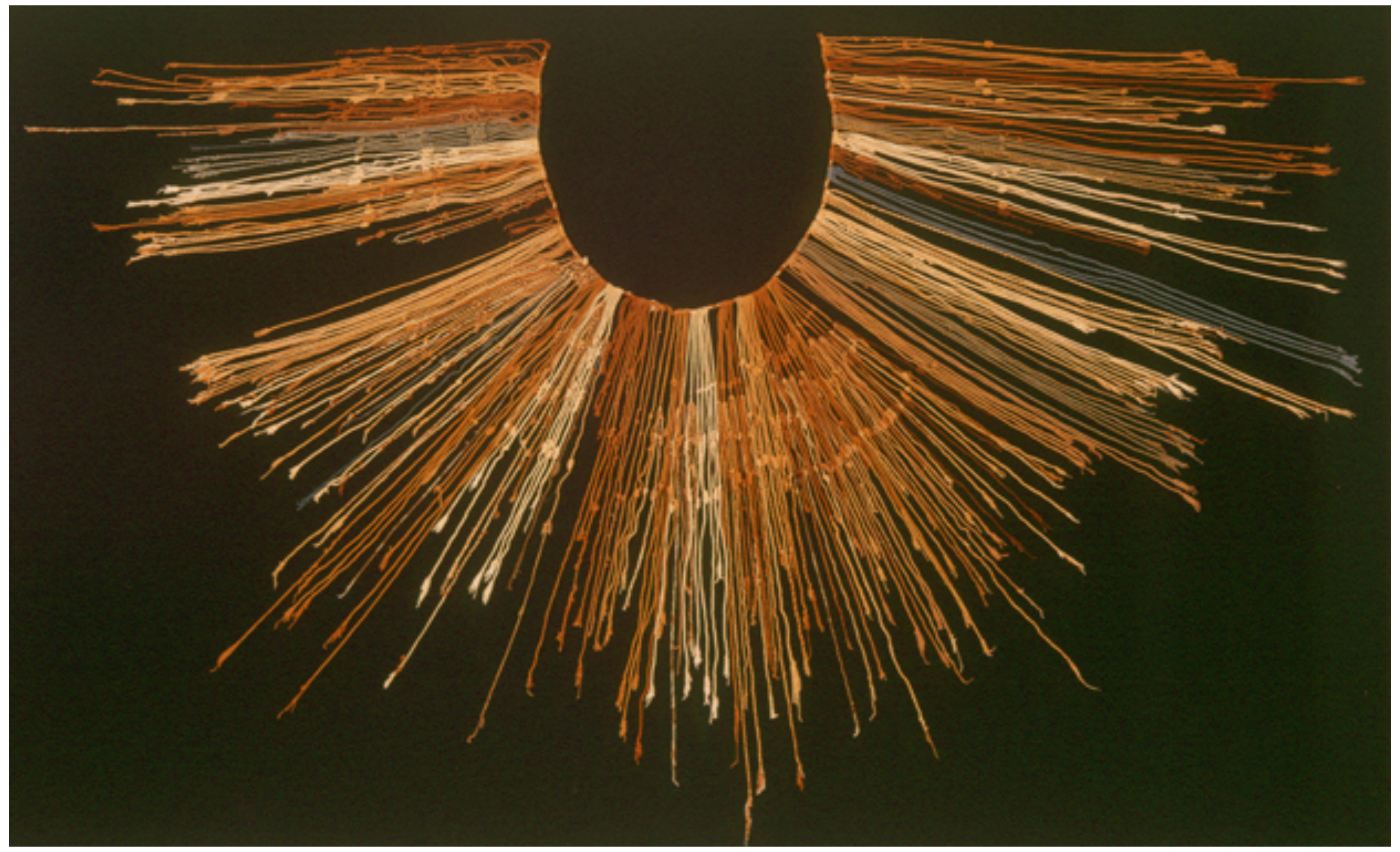
Exercise

https://bit.ly/3E3i6LP

or

https://swiftfiddle.com/jyzdziivsvfnrhmwdtmevhc37m

Khipu



Implements Robert C Martins “Clean Architecture”

It consists of UseCases, that are grouped in Features

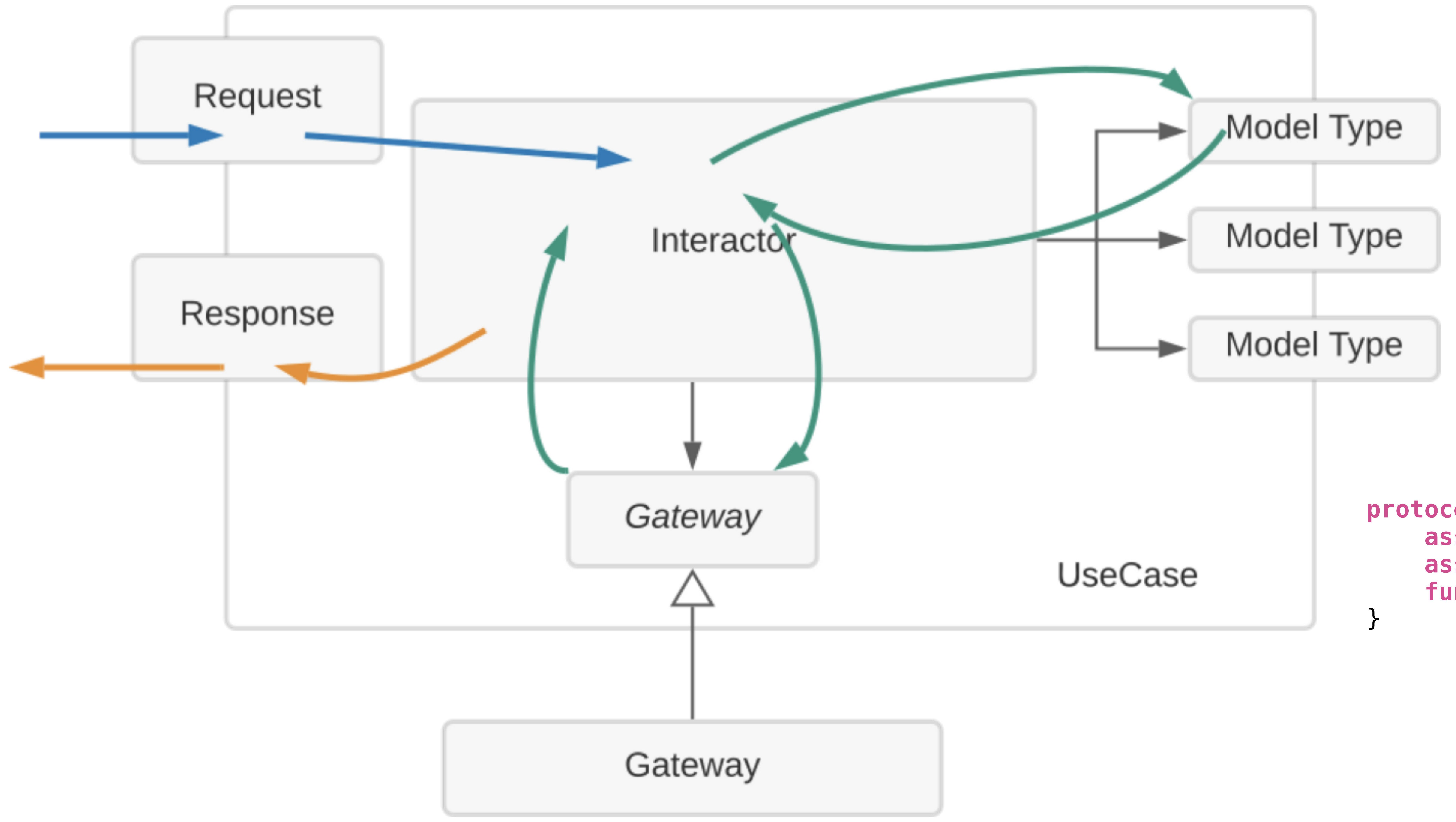
Features are grouped into Domains

UseCases have their own DSL each, ensuring separation of concerns

Features in a Domain share one Message DSL

Khipu

UseCases



```
protocol UseCase {
    associatedtype RequestType
    associatedtype ResponseType
    func request(to request:RequestType)
}
```

Khipu

UseCases

```
protocol UseCase {
    associatedtype RequestType
    associatedtype ResponseType
    func request(to request:RequestType)
}
```

```
struct ItemAdder:UseCase {
    enum Request { case add  (TodoItem) }
    enum Response{ case added(TodoItem) }
    init(store s:Store<AppState,AppState.Change>,
     responder r:@escaping (Response) -> ()) {
        store = s
        respond = r
    }
    private let store: Store<AppState,AppState.Change>
    private let respond: (Response) -> ()

    func request(to request: Request) {
        switch request {
        case let .add(t): store.change(.add(t))
                          respond(.added(t))
        }

    }
    typealias RequestType = Request
    typealias ResponseType = Response
}
```

```
struct ItemRemover:UseCase {
    enum Request { case remove (TodoItem) }
    enum Response{ case removed(TodoItem) }
    init(store s:Store<AppState,AppState.Change>,
     responder r: @escaping (Response) -> ()) {
        store = s
        respond = r
    }
    private let store: Store<AppState,AppState.Change>
    private let respond: (Response) -> ()

    func request(to request:Request) {
        switch request {
        case let .remove(t):store.change(.remove(t));
                            respond(.removed(t))
        }
    }
    typealias RequestType = Request
    typealias ResponseType = Response
}
```

Khipu

Features

```
typealias  Input = (Message) -> ()
typealias Output = (Message) -> ()

func createTodoListFeature(
    store    s: AppStore,
    output out: @escaping Output) -> Input {
    let initializer = ListInitializer(store:s,responder:process(on:out))
    let adder       = ItemAdder      (store:s,responder:process(on:out))
    let remover     = ItemRemover    (store:s,responder:process(on:out))
    let updater     = ItemUpdater    (store:s,responder:process(on:out))
    func execute(cmd:Message.Todos) {
        switch cmd {
        case     .initialize    : initializer.request(to:.initialize)
        case let .add   (item:t): adder      .request(to:.add   (t) )
        case let .remove(item:t): remover    .request(to:.remove(t) )
        case let .update(item:t): updater    .request(to:.update(t) )
        }
    }
    return { if case let .todos(c) = $0 { execute(cmd:c) } }
}
func process(on out:@escaping Output) -> (ListInitializer.Response) -> () {{ switch $0{ case .initialized   :out(.todos(.initialized))     }}}
func process(on out:@escaping Output) -> (ItemAdder      .Response) -> () {{ switch $0{ case let .added  (t):out(.todos(.added  (item:t))) }}}
func process(on out:@escaping Output) -> (ItemRemover    .Response) -> () {{ switch $0{ case let .removed(t):out(.todos(.removed(item:t))) }}}
func process(on out:@escaping Output) -> (ItemUpdater    .Response) -> () {{ switch $0{ case let .updated(t):out(.todos(.update (item:t))) }}}
```

Khipu

App Domain

```swift
typealias  Input = (Message) -> ()
typealias Output = (Message) -> ()
typealias AppStore = Store<AppState,AppState.Change>

func createAppDomain(
    store      : AppStore,
    receivers  : [Input],
    rootHandler: @escaping Output) -> Input {
    let features: [Input] = [
       createTodoListFeature(store:store,output:rootHandler),
       createLoggingFeature(store:store)
   ]
    return { msg in
        (receivers + features).forEach {
            $0(msg)
        }
    }
}
```

```swift
enum Message {
    case todos(Todos)
    case logger(Logger)
    enum Todos {
        case initialize
        case initialized
        case add    (item:TodoItem)
        case added  (item:TodoItem)
        case remove (item:TodoItem)
        case removed(item:TodoItem)
        case update (item:TodoItem)
        case updated(item:TodoItem)
    }
    enum Logger {
        case log(item:LogItem)
    }
}
```

Khipu

Assembling the app

```
@main
struct Items_App: App {
    var body: some Scene {
        WindowGroup {
            ContentView(viewState:viewState)
        }
    }
}
fileprivate var store       : Store     = createDiskStore()
fileprivate var viewState   : ViewState = ViewState(store:store,roothandler:{ rootHandler($0) })
fileprivate var rootHandler: ((Message) -> ())! = createAppDomain(
    store      : store,
    receivers  : [ viewState.handle(msg:) ],
    rootHandler: { rootHandler($0) }
)
```

Khipu

Axiomatic Codes

```
struct TodoItem:Identifiable {
    //….
    private func alter(_ c:Change    ) -> Self {
        switch c {
        case let    .title(t): return Self(id,t    ,completed,due,created,location)
        case          .finish: return Self(id,title,true     ,due,created,location)
        case        .unfinish: return Self(id,title,false    ,due,created,location)
        case let .location(l): return Self(id,title,false    ,due,created,l       )
        case let        .due(.timeSpan(.start(.from(b),.to(e)))):
            return e > b  //check for timespans if dates are in correct order
                ? Self(id,title,completed,.timeSpan(.start(.from(b),.to(e))),created,location)
                : self
        case let      .due(d): return Self(id,title,completed,d  ,created,location)
        }
    }
}
struct SimplePendulum {
    // …
    func alter(_ cmd:Change) -> Self {
        switch (paused,cmd) {
        case (false,       .tick        ): return advance()
        case ( true,    .unpause        ): return .init(angle,angleV,angleA,r,gravity,dampening,bobDiameter,hue,false )
        case (false,      .pause        ): return .init(angle,angleV,angleA,r,gravity,dampening,bobDiameter,hue,true  )
        case (_, let .set(.radius   (r))): return .init(angle,angleV,angleA,r,gravity,dampening,bobDiameter,hue,paused)
        case (_, let .set(.dampening(d))): return .init(angle,angleV,angleA,r,gravity,        d,bobDiameter,hue,paused)
        case (_, let .set(.gravity  (g))): return .init(angle,angleV,angleA,r,      g,dampening,bobDiameter,hue,paused)
        case (_, _                      ): return self
        }
    }
}
```

Khipu

Proving Correctness

Demo

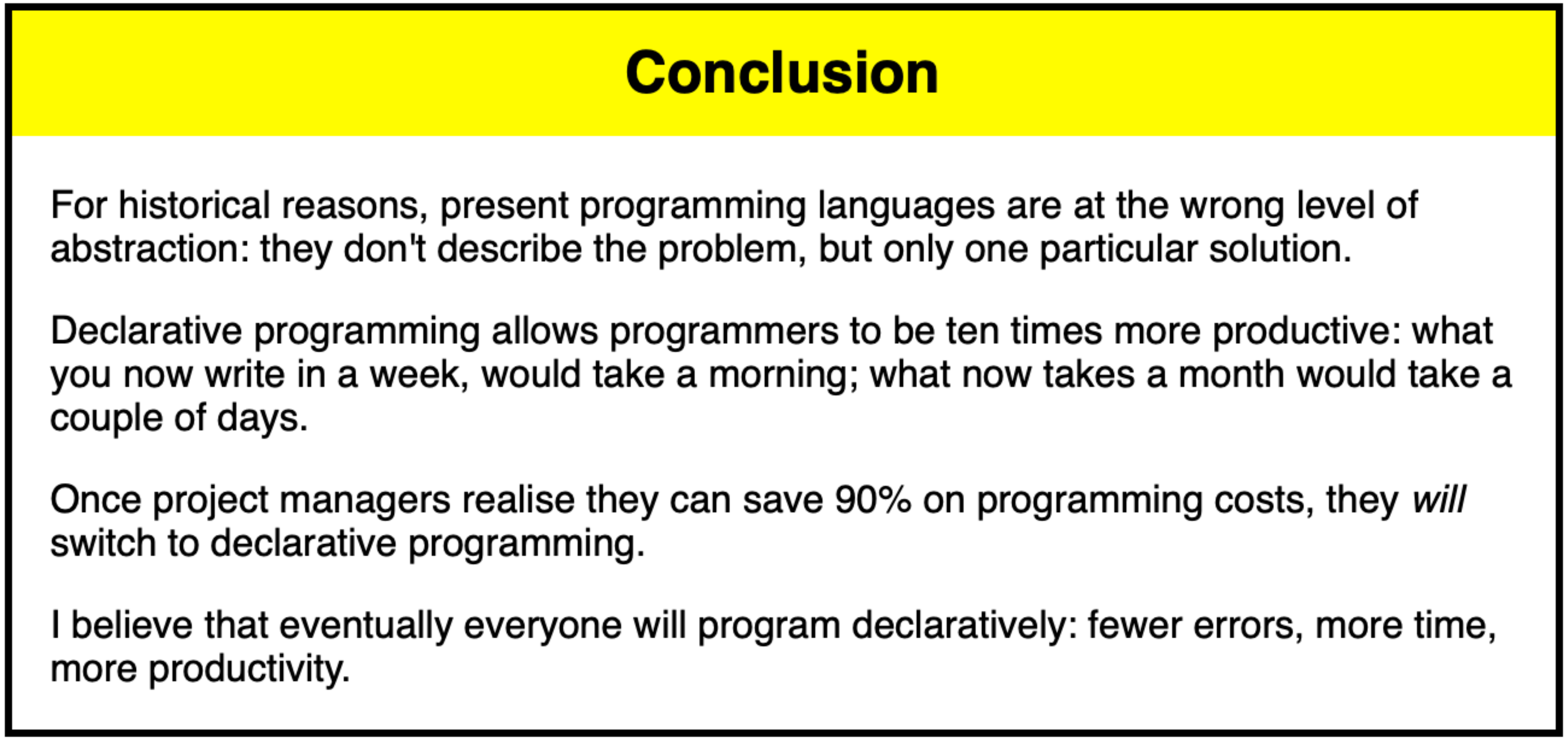
Khipu

Further Properties

- Minimal surface area: One function with one parameter
- Can be wrapped in anything — function, struct, class, tuple
- Khipu works as application architecture but also as library architecture
- Truly independent from UI, also very portable
- Easily testable with simple tests
- Allows for more agility also in designing (“form follows function”)

Declarative Coding

Why are most professional coders ignoring it?



Conclusion

For historical reasons, present programming languages are at the wrong level of abstraction: they don't describe the problem, but only one particular solution.

Declarative programming allows programmers to be ten times more productive: what you now write in a week, would take a morning; what now takes a month would take a couple of days.

Once project managers realise they can save 90% on programming costs, they *will* switch to declarative programming.

I believe that eventually everyone will program declaratively: fewer errors, more time, more productivity.

Source: “Declarative Amsterdam 2019”, Steven Pemberton “Declarative vs Procedural”

Declarative Coding

Economics of Effort

There is no relationship in science between effort and progress: It doesnt matter, how much time you put in into discovering something new.

Yet our education systems and work places incentivise effort-based thinking.

Result: obsession with tooling but unique and fresh thinking is discouraged

Procedural/imperative coding fits the world view of effort-based thinkers, while declarative coding is alien to them.

Declarative Coding

Economics of Effort

This leads to Sunk Cost Fallacy:

No matter how defect a certain code is — it will be preferred over investing any amount of time or resources into better replacements

Effectively this means, that errors, long hours and constantly burnt out developers are preferred over clean, error-free and efficiently created codes and developers mental health

Declarative Coding

When will they finally switch?

Just in the last half year the worldwide economic situation has dramatically shifted

Due to inflation money becomes more expensive and harder to get by start ups and other companies

This is our best chance in years to spread the word for Declarative Coding, as many of them will have to create more with less money — a perfect task for Declarative Coding

“Hoping for change without doing anything yourself, is like standing at the train station waiting for a ship…”

– Albert Einstein

Thanks!

vikingosegundo@gmail.com

decodemeester.medium.com

https://gitlab.com/vikingosegundo