
THIS IS THE WAY
imgflip.com

Building Software Architecture from First Principles

Dejan Miličić
Head of DevRel @ RavenDB

BDD#20

What are First Principles?

- most basic, foundational truths or assumptions
- cannot be deduced from any other propositions
- breaking down complex systems into their most fundamental parts
- understanding them thoroughly before building up new ideas

- Fundamental Building Blocks
- Innovation and Creativity
- Questioning every assumption and belief about a problem
- Decisions based on fundamental truths

Our goal today

- apply First Principles reasoning to Software Architecture
- examine various Architectural Styles and Patterns
- identify common parts and principles
- decompose into Fundamental Blocks
- recompose into Something New

Common architecture styles today…
(some are architectural patterns, actually)

- Layered (N-Tier) Architecture
- Microservices
- Event-Driven Architecture – EDA
- Service-Oriented Architecture - SOA
- Monolith
- Modular Monolith
- Clean Code
- Hexagonal (Ports & Adapters)
- Event Sourcing
- Functional Core, Imperative Shell
- CQRS

…concentrating on different aspects

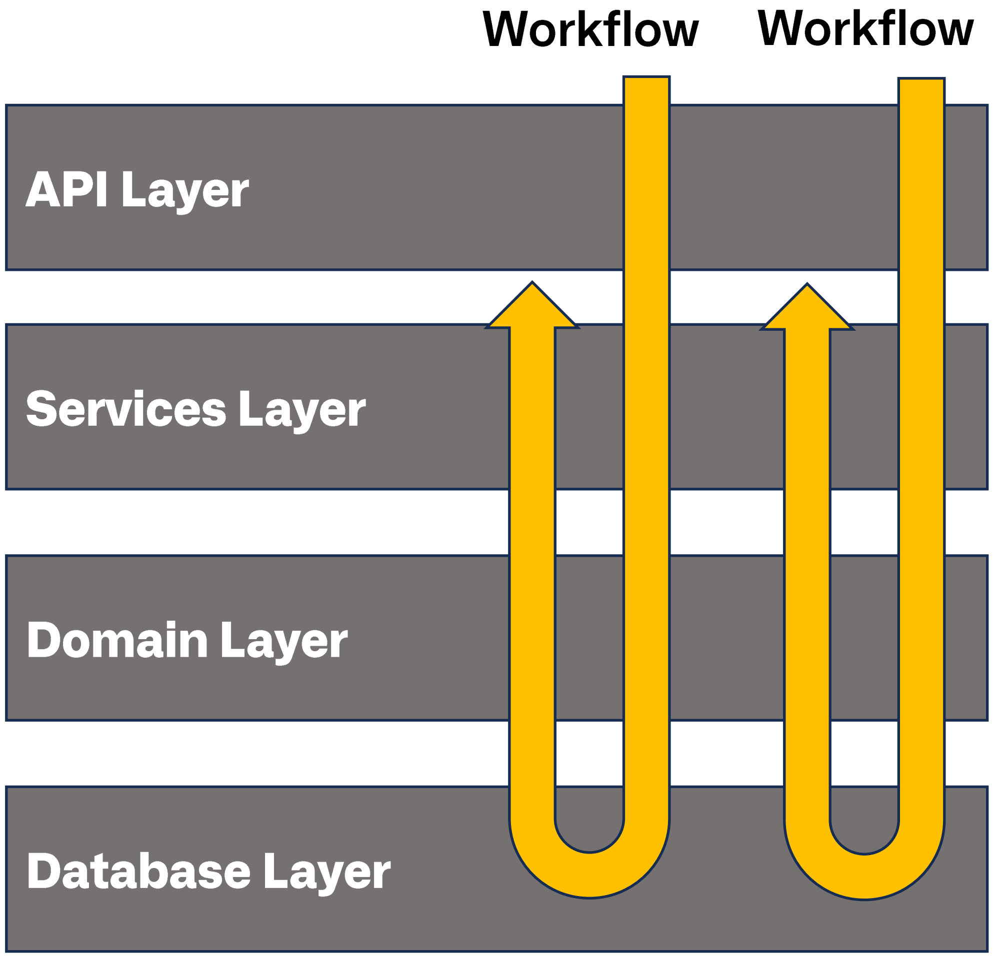
Structure

- Layered
- Component-based
- Pipes & Filters
- MVC
- Object-oriented

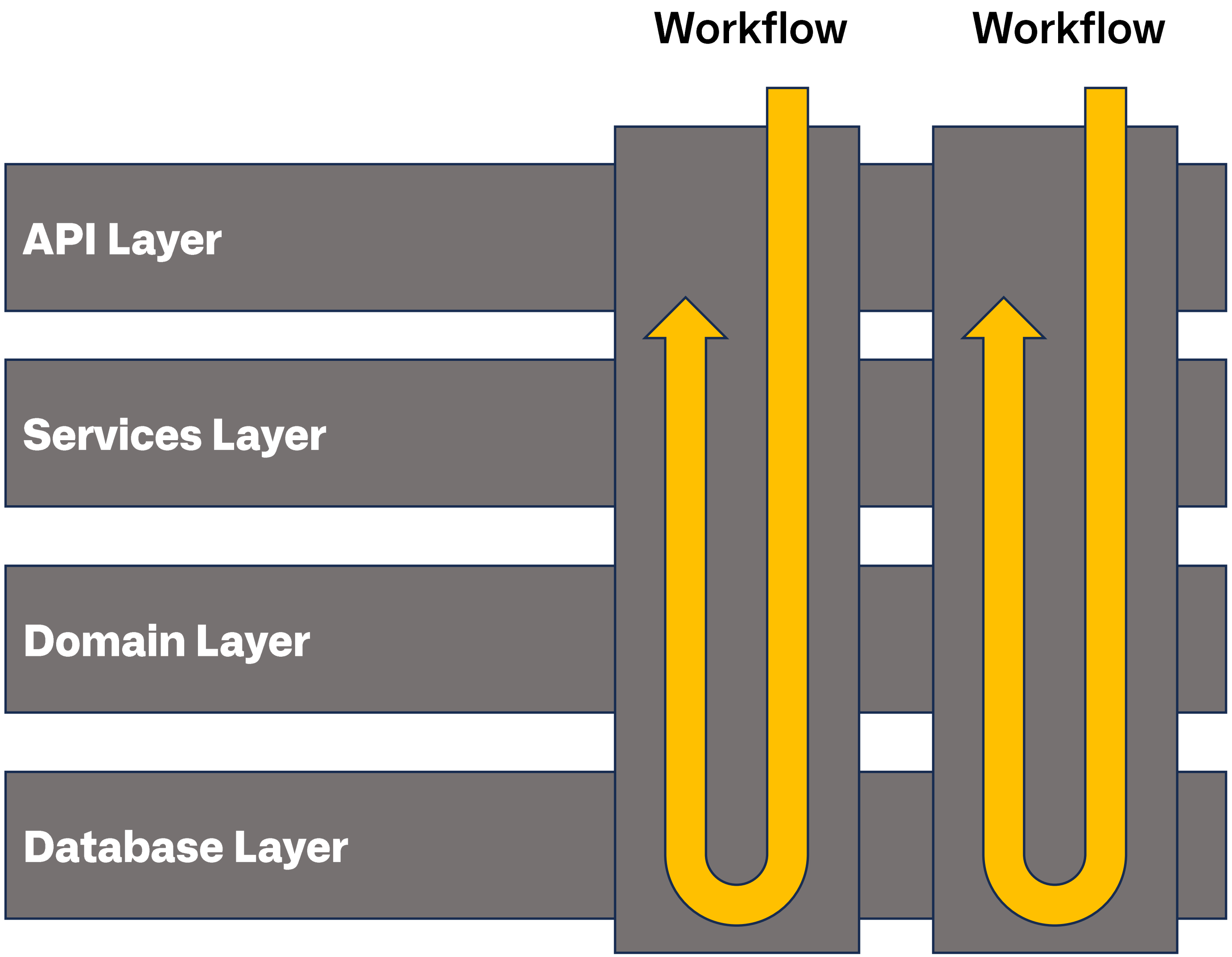
Communication

- SOA
- Message Bus
- EDA
- Publish-Subscribe

Layered (N-Tier) Architecture

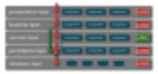
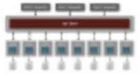
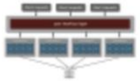


Layered Architecture with Vertical Slices



Constraints

1. Open and Closed layers must be respected
2. Only Database layer can talk to a database
3. All database logic must reside in Database layer

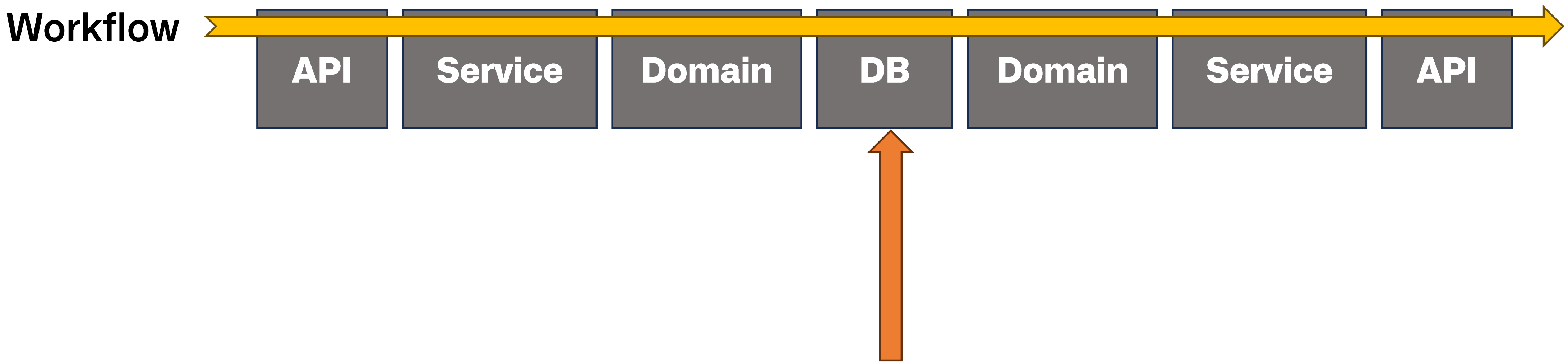
	layered 	modular monolith 	microkernel 	microservices 	service-based 	service-oriented 	event-driven 	space-based 
agility	★	★★	★★★	★★★★★	★★★★	★	★★★	★★
abstraction	★	★	★★★	★	★	★★★★★	★★★★	★
configurability	★	★	★★★★	★★★	★★	★	★★	★★
cost	★★★★★	★★★★★	★★★★★	★	★★★★	★	★★★	★★
deployability	★	★★	★★★	★★★★★	★★★★	★	★★★	★★★
domain part.	★	★★★★★	★★★★★	★★★★★	★★★★★	★	★	★★★★★
elasticity	★	★	★	★★★★★	★★	★★★	★★★★	★★★★★
evolvability	★	★	★★★	★★★★★	★★★	★	★★★★★	★★★
fault-tolerance	★	★	★	★★★★★	★★★★	★★★	★★★★★	★★★
integration	★	★	★★★	★★★	★★	★★★★★	★★★	★★
interoperability	★	★	★★★	★★★	★★	★★★★★	★★★	★★
performance	★★	★★★	★★★	★★	★★★	★★	★★★★★	★★★★★
scalability	★	★	★	★★★★★	★★★	★★★	★★★★★	★★★★★
simplicity	★★★★★	★★★★★	★★★★	★	★★★	★	★	★
testability	★★	★★	★★★	★★★★★	★★★★	★	★★	★
workflow	★	★	★★	★	★	★★★★★	★★★★★	★

Fundamentals of Software Architecture: An Engineering Approach
by Mark Richards & Neal Ford, 2020

Vertical Slice stretched out

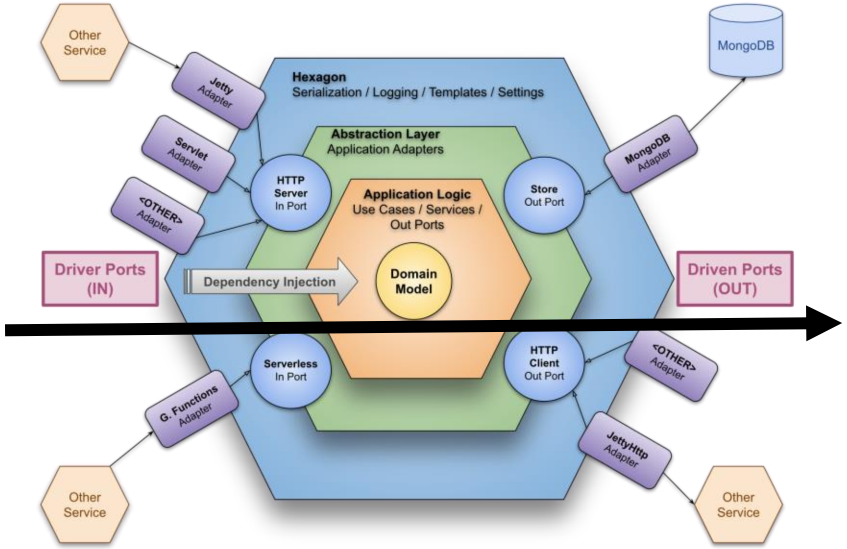
a.k.a “there is no gravity in software architecture”

Database-centric architecture

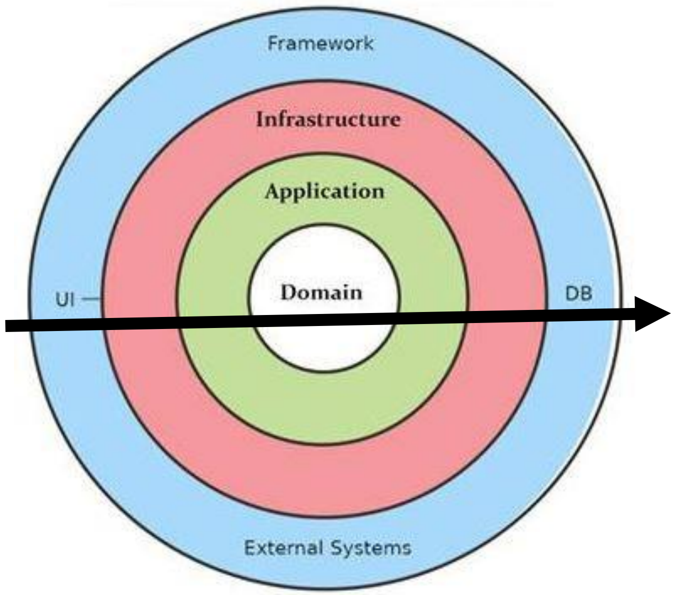


Domain-centric architectures

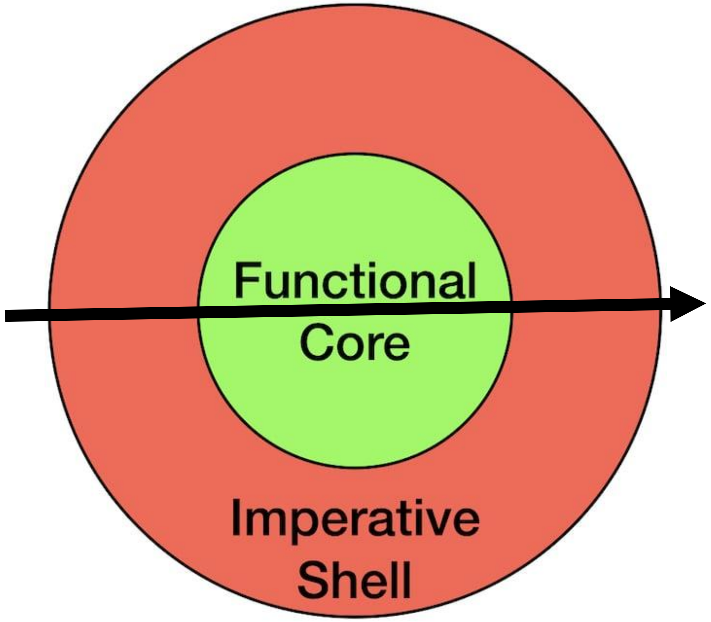
Hexagonal
Ports & Adapters



Clean
Onion



Functional Core / Imperative Shell

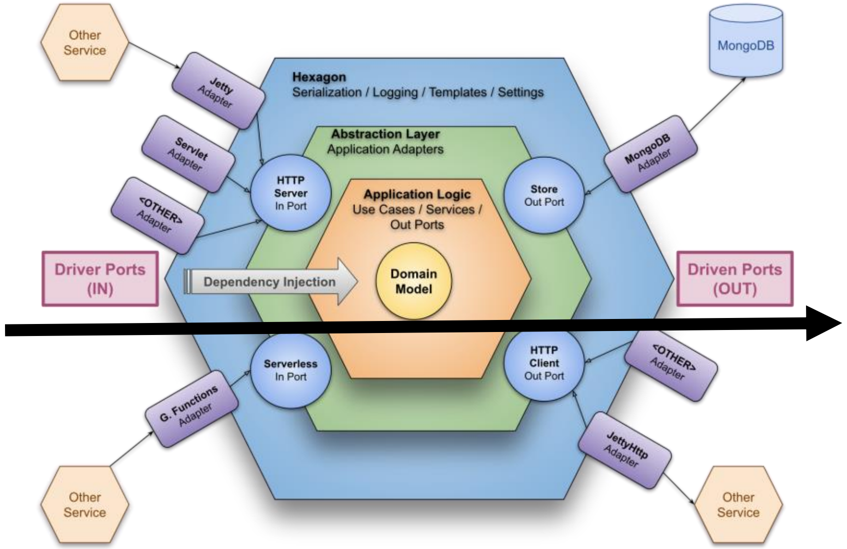


First Principles in Architecture

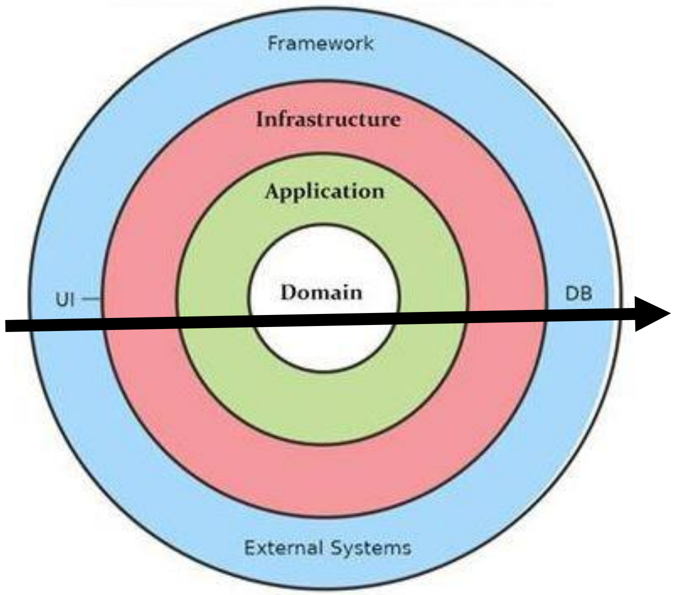
- #1 – Domaincentricity

Domain-centric architectures

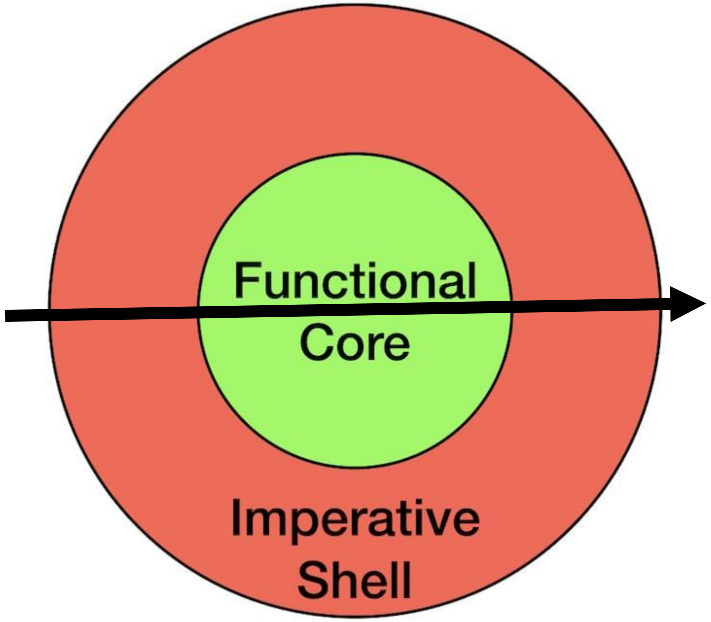
Hexagonal
Ports & Adapters



Clean
Onion



Functional Core / Imperative Shell



Domain-centric Workflow

Composition

over inheritance

Composition

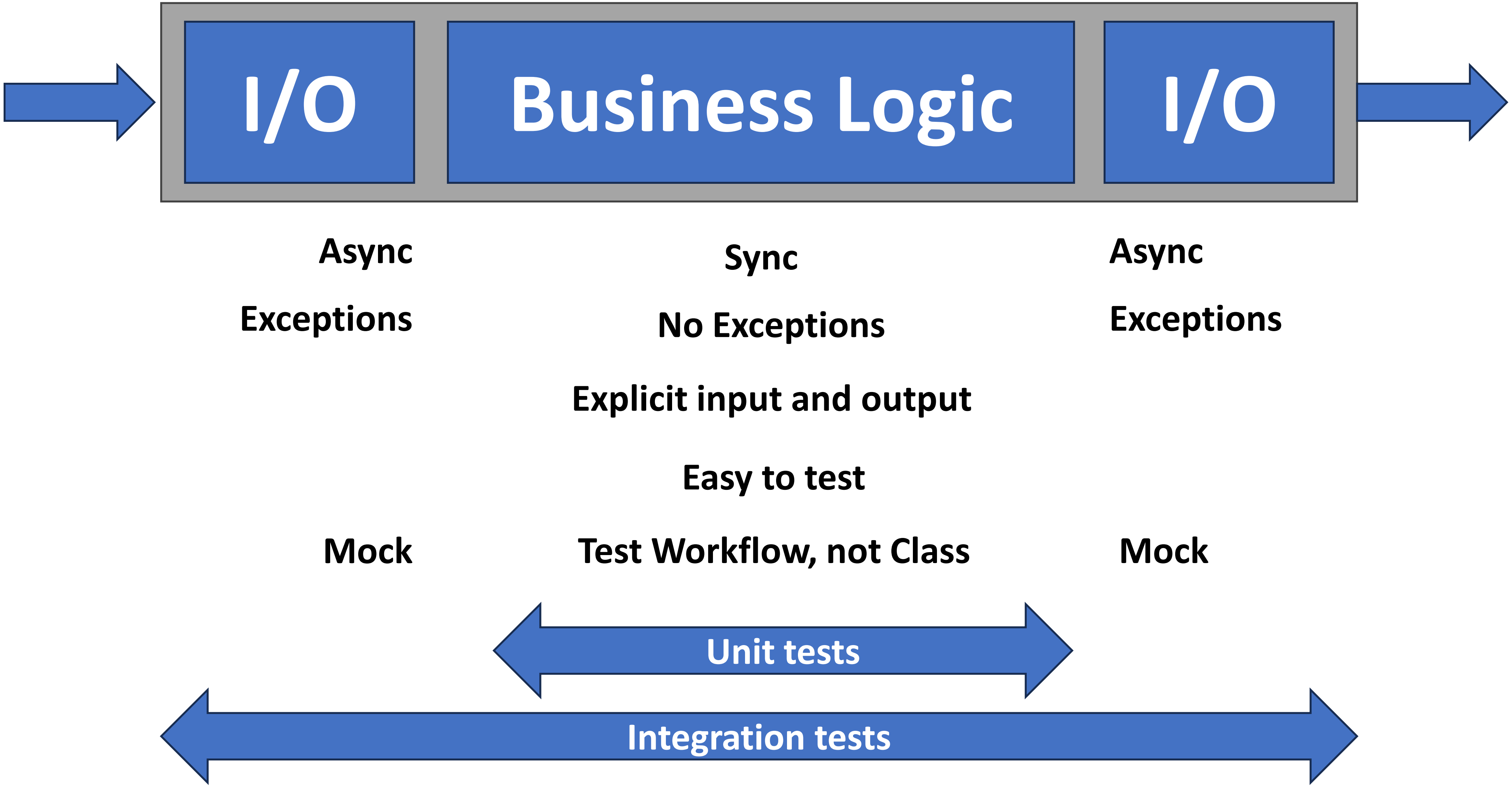
Load data from DB
Get data from input

Process data

Save data to DB
Display data

Imperative shell	Functional core	Imperative shell
Impure shell	Pure core	Impure shell
Non-deterministic	Deterministic	Non-deterministic

Domain-centric Workflow



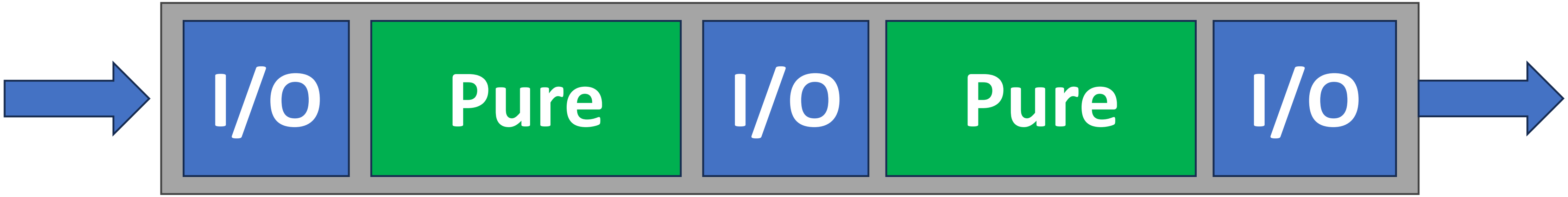
Pure functions!

- = Deterministic + No Side Effects
- referential transparency
- honest functions
- easier testing
- lower cognitive load, easier reasoning
- composability
- reuse
- parallelization (thread safe)

First Principles in Architecture

- #1 – Domaincentricity
- #2 – Pure Functional Core

I/O in the middle of the Workflow?



Heavy ORMs might not be a good fit…

```
void AddEntry(int listId, string desc, IDocumentStore store)
{
    using var dbSession = store.OpenSession();

    var entry = new TodoListEntry();
    entry.ListId = listId;
    entry.Desc = desc;
    session.Store(entry);

    var sendEmailCommand = new SendEmailCommand();
    sendEmailCommand.EntryId = entry.Id;
    session.Store(sendEmailCommand);

    session.SaveChanges();
}
```

… but refactoring is possible (sometimes)

```
void AddEntry(int listId, string desc, IDocumentStore store)
{
    var entry = new TodoListEntry();
    entry.ListId = listId;
    entry.Desc = desc;

    var sendEmailCommand = new SendEmailCommand();
    sendEmailCommand.EntryId = entry.Id;

    using var dbSession = store.OpenSession();
    session.Store(entry);
    session.Store(sendEmailCommand);
    session.SaveChanges();
}
```

Validation on the edge

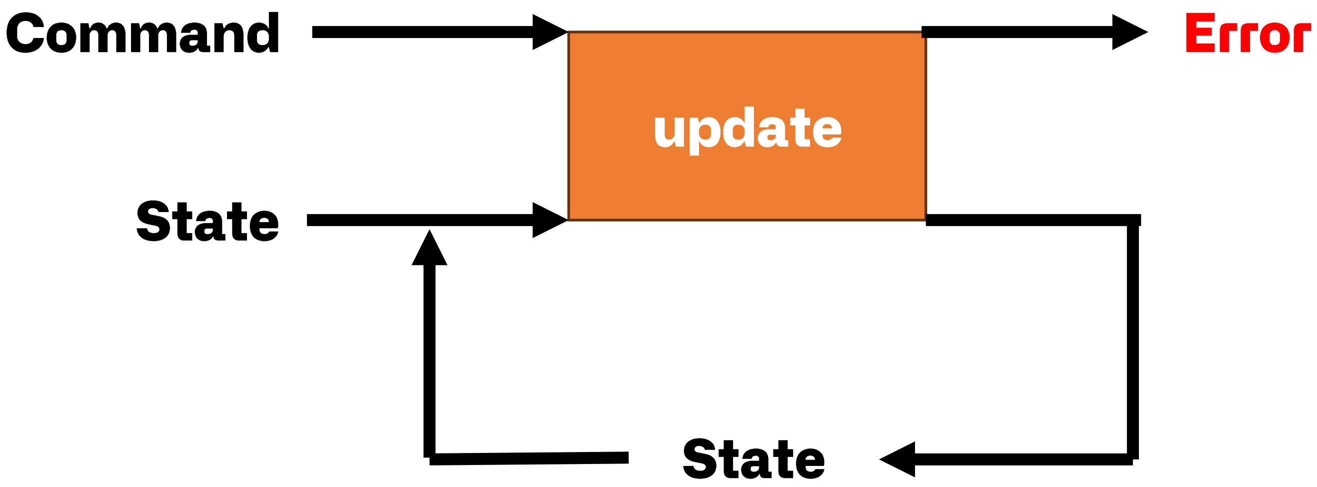
- Validation on the edge
- No need for validation in the core domain
- Parse, don't validate!
- Parsing == Types

First Principles in Architecture

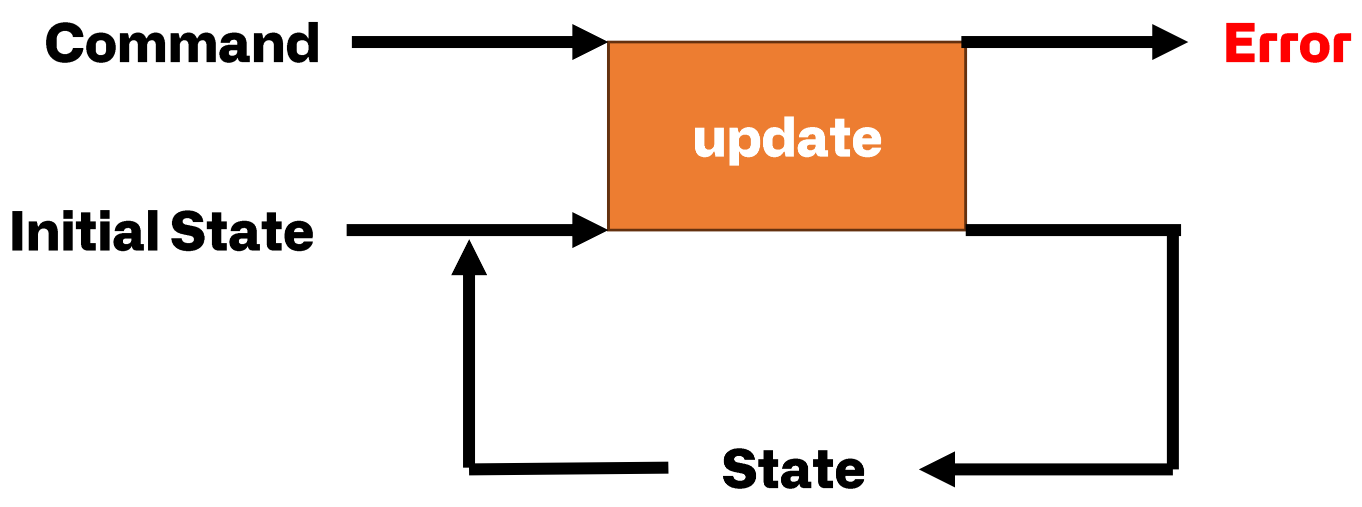
- #1 – Domaincentricity
- #2 – Pure Functional Core
- #3 – Type-Driven Design


```
let update(State, Command) -> State’
```

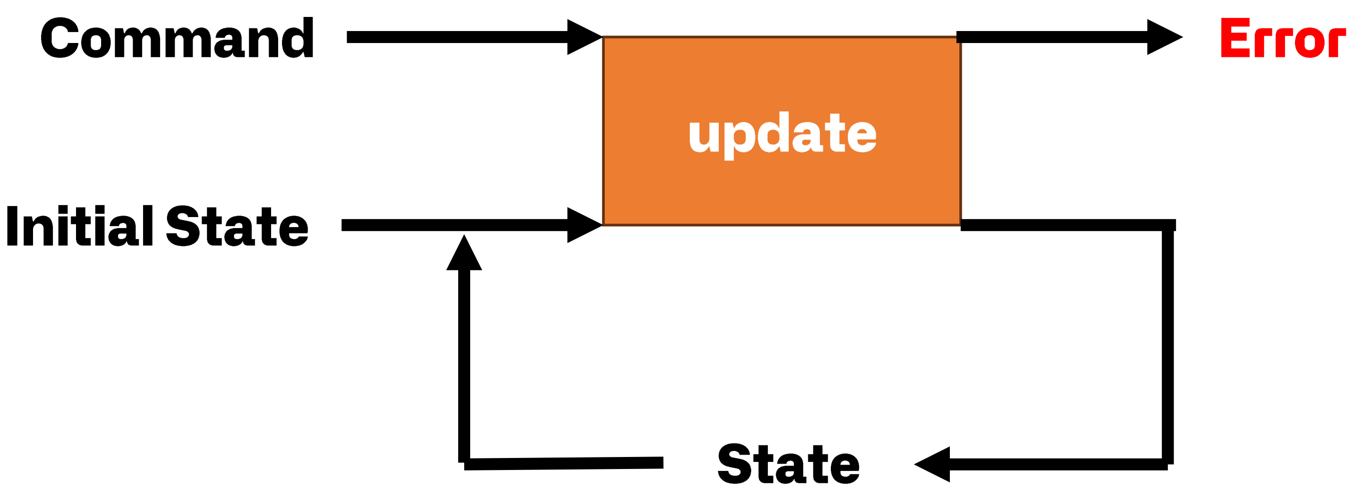

```
let update(State, Command) -> Result<State, Error>
```




```
let update(State, Command) -> Result<State, Error>
```




```
let update(State, Command) -> Result<State, Error>
let init -> State
```




```
type Object =
  { update : (State, Command) -> Result<State, Error>
    init   : State }
```

Transaction script design pattern

- pattern for complete handling of individual requests
- top-down linear flow
- Advantages
  - simplicity
  - rapid development
- Disadvantages
  - code duplication
  - limited scalability

Transaction Script [impl:update]

```
let handle msg id object db : Result<unit, Error> =
    let state =
        match db.load id with
        | Ok(Some state) -> Ok state
        | Ok(None) -> Ok object.init
        | Error err -> Error err

    match state with
    | Ok state ->
        match object.update state msg with
        | Ok newState -> db.save id newState
        | Error err -> Error err
    | Error err -> Error err
```

```
type Object =
  { update : (State, Msg) -> Result<State, Error>
    init   : State }

type KeyValueStore =
 { load: Key -> Result<State option, Error>
   save: (Key, State) -> Result<unit, Error> }
```

Todo List [impl:update]

```fsharp
type Msg =
    | CreateEntry of string
    | CompleteEntry of string
    | DeleteList

type State =
    { Entries: TodoEntry list
      IsDeleted: bool }

and TodoEntry = { Description: string; IsDone: bool }

type Error =
    | ListIsDeleted
    | UncompletedEntries

type Object =
    { update: State → Msg → Result<State, Error>
      init: State }
```

```fsharp
let init = { Entries = []; IsDeleted = false }

let update (state: State) (msg: Msg) : Result<State, Error> =
    match msg, state with
    | CreateEntry _, state when state.IsDeleted = true → Error ListIsDeleted
    | CreateEntry desc, state →
        Ok
            { state with
                Entries = state.Entries @ [ { Description = desc; IsDone = false } ] }

    | CompleteEntry _, state when state.IsDeleted = true → Error ListIsDeleted
    | CompleteEntry desc, state →
        let updateEntry entry =
            if entry.Description = desc then
                { entry with IsDone = true }
            else
                entry

        Ok
            { state with
                Entries = state.Entries ▷ List.map updateEntry }

    | DeleteList, state when state.IsDeleted → Ok state
    | DeleteList, state →
        if state.Entries ▷ List.exists (fun e → not e.IsDone) then
            Error UncompletedEntries
        else
            Ok { state with IsDeleted = true }

let todoList = { update = update; init = init }
```

```fsharp
CreateEntry "Buy milk"
▷ todoList.update todoList.init
▷ verify (
    Ok
        { Entries =
            [ { Description = "Buy milk"
                IsDone = false } ]
          IsDeleted = false }
)
```

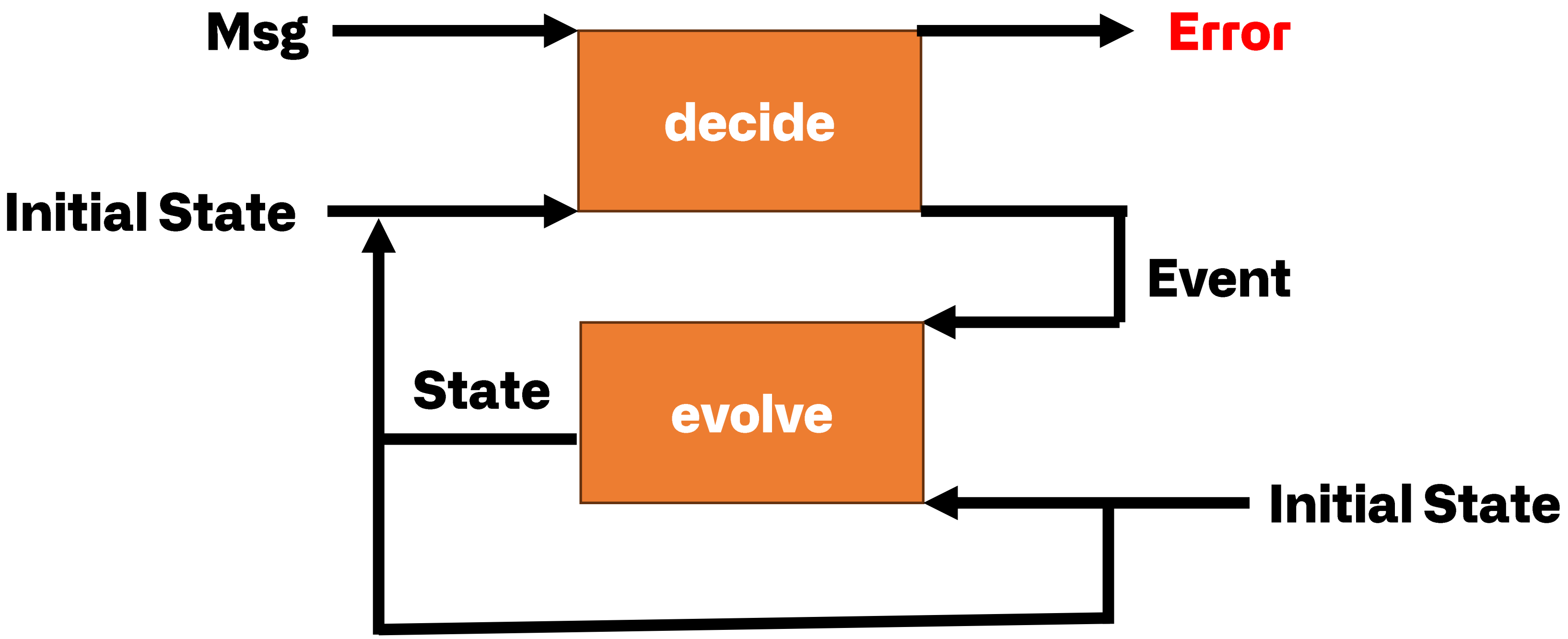
Event Sourcing

- recording sequence of immutable events
- events represent ordered facts about the past

- you can reconstruct state at any point in time
- you are not forced to model – projections are easily changed
- no loss of information about the system
- audability
- scalability
- flexibility


```
let decide (State, Msg) -> Event list
```


```
let decide (State, Msg) -> Result<Event list, Error>
```




```
let decide (State, Msg) -> Result<Event list, Error>
let evolve (State, Event) -> State
let init -> State
```

Event Sourcing [impl:decide + evolve]

```
let handle msg id object es : Result<unit, Error> =
    let state =
        match es.loadEvents id with
        | Ok [] -> Ok object.init
        | Ok events -> Ok(List.fold object.evolve object.init events)
        | Error err -> Error err

    match state with
    | Ok currState ->
        match object.decide currState msg with
        | Ok newEvents -> es.appendEvents id newEvents
        | Error err -> Error err
    | Error err -> Error err
```

```
type Object =
  { decide : (State, Command) -> Result<Event list, Error>
    evolve : (State, Event) -> State
    init   : State }

type EventStore =
 { loadEvents: Key -> Result<Event list, Error>
   appendEvents: (Key, Event list) -> Result<unit, Error>}
```

Todo List [impl:decide + evolve]

```
type Msg =
    | CreateEntry of string
    | CompleteEntry of string
    | DeleteList

type Event =
    | EntryCreated of string
    | EntryCompleted of string
    | ListDeleted

type State =
    { Entries: TodoEntry list
      IsDeleted: bool }

and TodoEntry = { Description: string; IsDone: bool }

type Error =
    | ListIsDeleted
    | UncompletedEntries

type Object =
    { decide: State → Msg → Result<Event list, Error>
      evolve: State → Event → State
      init: State }
```

```
CreateEntry "Buy milk"
▷ todoList.decide todoList.init
▷ verify (Ok [ EntryCreated "Buy milk" ])


[ EntryCreated "Buy milk" ]
▷ List.fold todoList.evolve todoList.init
▷ verify (
    { Entries =
        [ { Description = "Buy milk"
            IsDone = false } ]
      IsDeleted = false }
)
```

```
let decide (state: State) (msg: Msg) : Result<Event list, Error> =
    match msg with
    | CreateEntry _ when state.IsDeleted → Error ListIsDeleted
    | CreateEntry entry → Ok [ EntryCreated entry ]

    | CompleteEntry _ when state.IsDeleted → Error ListIsDeleted
    | CompleteEntry entry → Ok [ EntryCompleted entry ]

    | DeleteList when state.IsDeleted → Ok []
    | DeleteList →
        if state.Entries ▷ List.exists (fun e → not e.IsDone) then
            Error UncompletedEntries
        else
            Ok [ ListDeleted ]

let evolve (state: State) (event: Event) : State =
    match event with
    | EntryCreated desc →
        { state with
            Entries = state.Entries @ [ { Description = desc; IsDone = false } ] }

    | EntryCompleted desc →
        let updateEntry entry =
            if entry.Description = desc then
                { entry with IsDone = true }
            else
                entry

        { state with
            Entries = state.Entries ▷ List.map updateEntry }

    | ListDeleted → { state with IsDeleted = true }

let init = { Entries = []; IsDeleted = false }

let todoList =
    { decide = decide
      evolve = evolve
      init = init }
```

Let’s compare two approaches

```
type Object =
  { update : (State, Command) -> Result<State, Error>
    init   : State }

type Object =
  { decide : (State, Command) -> Result<Event list, Error>
    evolve : (State, Event) -> state
    init   : State }
```

We can unify them

```
type Object =
  { update : (State, Command) -> Result<State, Error>
    init   : State }
```

```
type Object =
  { decide : (State, Command) -> Result<Event list, Error>
    evolve : (State, Event) -> state
    init   : State }
```

```
let update state command = // (Command, State) -> Result<State, Error>
  match decide command state with
  | OK events ->
        let newState = List.fold evolve state events
       newState
  | Error err -> Error err
```

Transaction Script [impl:decide + evolve]

```
let handle msg id object db : Result<unit, Error> =
    let state =
        match db.load id with
        | Ok None -> Ok object.init
        | Ok(Some state) -> Ok state
        | Error err -> Error err

    match state with
    | Ok currState ->
        let events = object.decide currState msg
        match events with
        | Ok newEvents ->
            let newState = List.fold object.evolve currState newEvents
            db.save id newState
        | Error err -> Error err
    | Error err -> Error err
```

```
type Object =
  { decide : (State, Command) -> Result<Event list, Error>
    evolve : (State, Event) -> State
    init   : State }
```

Event Sourcing | Transaction Script

```
type Object =
  { decide : (State, Msg) -> Result<Event list, Error>
    evolve : (State, Event) -> State
    init   : State }
```

```
let handle msg id object es : Result<unit, Error> =
    let state =
        match es.loadEvents id with
        | Ok [] -> Ok object.init
        | Ok events ->
            Ok(List.fold object.evolve object.init events)
        | Error err -> Error err

    match state with
    | Ok currState ->
        match object.decide currState msg with
        | Ok newEvents -> es.appendEvents id newEvents
        | Error err -> Error err
    | Error err -> Error err
```

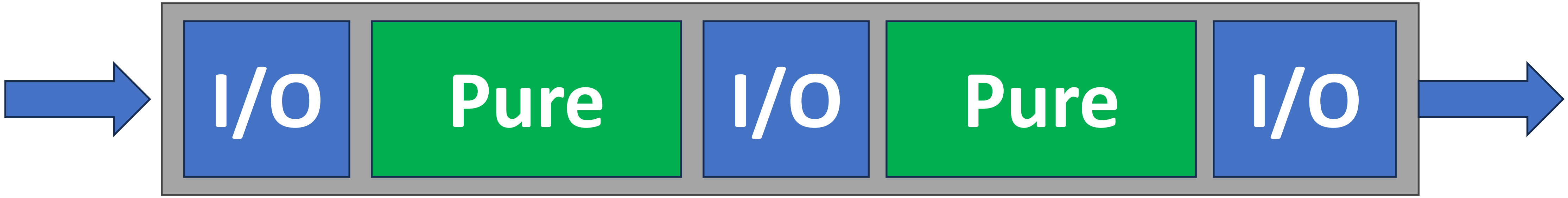
```
let handle msg id object db : Result<unit, Error> =
    let state =
        match db.load id with
        | Ok None -> Ok object.init
        | Ok(Some state) -> Ok state
        | Error err -> Error err

    match state with
    | Ok currState ->
        let events = object.decide currState msg
        match events with
        | Ok newEvents ->
            let newState =
              List.fold object.evolve currState newEvents
            db.save id newState
        | Error err -> Error err
    | Error err -> Error err
```

```
type EventStore =
 { loadEvents: Key -> Result<Event list, Error>
   appendEvents: (Key, Event list) -> Result<unit, Error>}
```

```
type KeyValueStore =
 { load: Key -> Result<State option, Error>
   save: (Key, State) -> Result<unit, Error> }
```

Side Effects?

- how to send an email after creating todo list entry?
- how to load or save something to the database?
- how to mix pure functions and I/O when you need?



Effects (Algebraic Effects)

- A way to model side effects as first-class types within the type system, separating them from pure logic
- Encapsulate side effects without directly executing them in functions
- Decouple pure logic from side effects

Examples

  - I/O operations: network requests, file I/O, database access
  - Error handling – exceptions, retries, fallback behaviors

Example: modeling side effects as Effects

```
type Effect = SendEmail of string


let perform (effect: Effect) : Async<Result<Msg list, Error>> =
    match effect with
    | SendEmail desc ->
        async {
            // call to the email server
            // use try/catch, process errors…
            return Ok []
        }
```

Event Sourcing | Transaction Script

```
type Object =
    { decide: State -> Msg -> Result<Event list * Effect list, Error>
      evolve: State -> Event -> State
      perform: Effect -> Async<Result<Msg list, Error>>
      init: State * Effect list }
```

```
let handle msg id object es : Result<Effect list, Error> =
    match es.loadEvents id with
    | Error err -> Error err
    | Ok events ->
        let (initState, initEffects) =
            match events with
            | [] -> object.init
            | _ -> List.fold object.evolve
                      (fst object.init) events, []

        match object.decide initState msg with
        | Error err -> Error err
        | Ok(newEvents, newEffects) ->
            match es.appendEvents id newEvents with
            | Error err -> Error err
            | Ok() -> initEffects @ newEffects |> Ok
```

```
let handle msg id object db : Result<Effect list, Error> =
    match db.load id with
    | Error err -> Error err
    | Ok state ->
        let (initState, initEffects) =
            match state with
            | None -> object.init
            | Some state -> state, []

        match object.decide initState msg with
        | Error err -> Error err
        | Ok(newEvents, newEffects) ->
            let newState =
               List.fold object.evolve initState newEvents

            match db.save id newState with
            | Error err -> Error err
            | Ok() -> initEffects @ newEffects |> Ok
```

```
type Msg =
    | CreateEntry of string
    | CompleteEntry of string
    | DeleteList

type Event =
    | EntryCreated of string
    | EntryCompleted of string
    | ListDeleted

type Effect = SendEmail of string

type State =
    { Entries: TodoEntry list
      IsDeleted: bool }

and TodoEntry = { Description: string; IsDone: bool }

type Error =
    | ListIsDeleted
    | UncompletedEntries
```

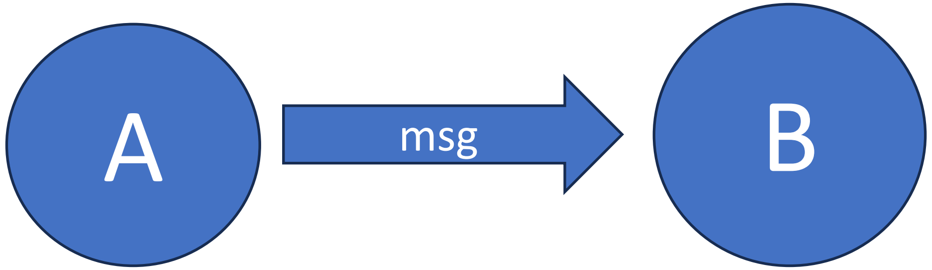
```
let decide state msg : Result<Event list * Effect list, Error> =
    match msg with
    | CreateEntry _ when state.IsDeleted -> Error ListIsDeleted
    | CreateEntry entry -> Ok([ EntryCreated entry ], [ SendEmail entry ])

    | CompleteEntry _ when state.IsDeleted -> Error ListIsDeleted
    | CompleteEntry entry -> Ok([ EntryCompleted entry ], [])

    | DeleteList when state.IsDeleted -> Ok([], [])
    | DeleteList ->
        if state.Entries |> List.exists (fun e -> not e.IsDone) then
            Error UncompletedEntries
        else
            Ok([ ListDeleted ], [])
```

```
type Object =
    { decide: State -> Msg -> Result<Event list * Effect list, Error>
      evolve: State -> Event -> State
      perform: Effect -> Async<Result<Msg list, Error>>
      init: State * Effect list }
```

```
let perform (effect: Effect) : Async<Result<Msg list, Error>> =
    match effect with
    | SendEmail desc ->
        async {
            do! Async.Sleep 30
            printfn "Sending email: %s" desc
            return Ok []
        }
```

Todo List [decide + evolve + effect]

Message passing in OOP



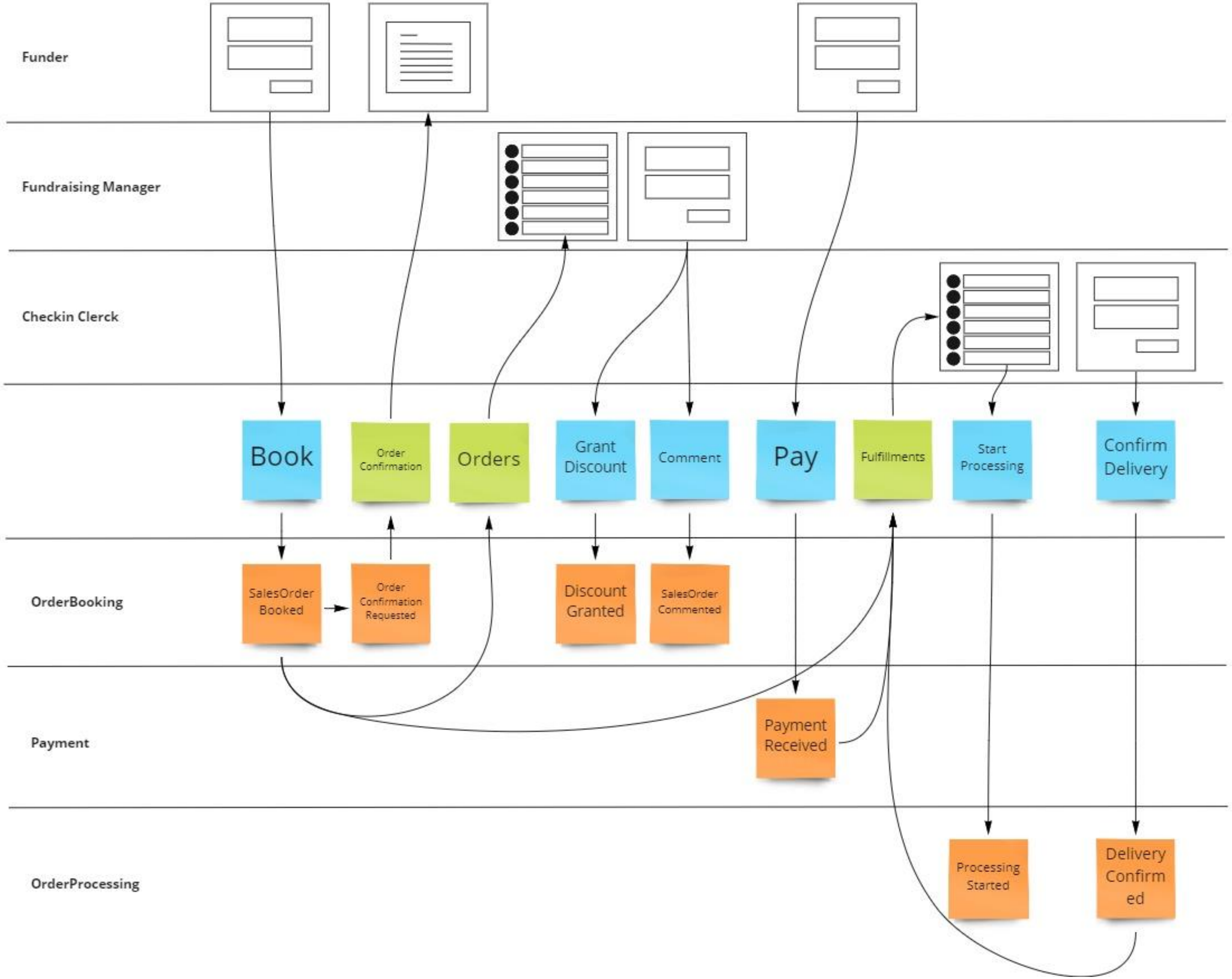
- typical OO language: `class A { … b.msg() }`, routing via vtables
- not a syntax sugar anymore
- not Alan Kay’s message passing anymore
- synchronous, in-process call that demands certain guarantees
- coupled
- performance: latency(A -> B -> C) = latency(A) + latency(B) + latency(C)

Message passing

- Asynchronous
- Decoupling
- Multiple receivers
- Horizontal scaling
- EDA – compatible
- Backpressure
- Concurrency control at code level
- Lower latency / Higher Throughput
- Routing
- Buffering
- Resilience / Fault Tolerance

First Principles in Architecture

- #1 – Domaincentricity
- #2 – Pure Functional Core
- #3 – Type-Driven Design
- #4 – Async Messaging



Event Modeling

- Event Modeling
  Adam Dymitruk, 2018
- Event Storming
  Alberto Brandolini, 2013
- CQRS
  Greg Young, 2007
- Event Sourcing
  Martin Fowler, 2005
- Domain-Driven Design
  Eric Evans, 2003

Credit: Yves Goeleven

Fantastic 9 – Yves Goeleven, 2023

In\Out	Command	Event	State
Command	Delegation	Aggregate Root	Downstream Activity
Event	Reaction	Event Stream Processing	Projection
State	Task processing	Event Generator	State Transformation

First Principles in Architecture

- #1 – Domaincentricity
- #2 – Pure Functional Core
- #3 – Type-Driven Design
- #4 – Async Messaging

```
type Object =
    { decide: State -> Msg -> Result<Event list * Effect list, Error>
      evolve: State -> Event -> State
      perform: Effect -> Async<Result<Msg list, Error>>
      init: State * Effect list }
```


THIS IS THE WAY
imgflip.com