



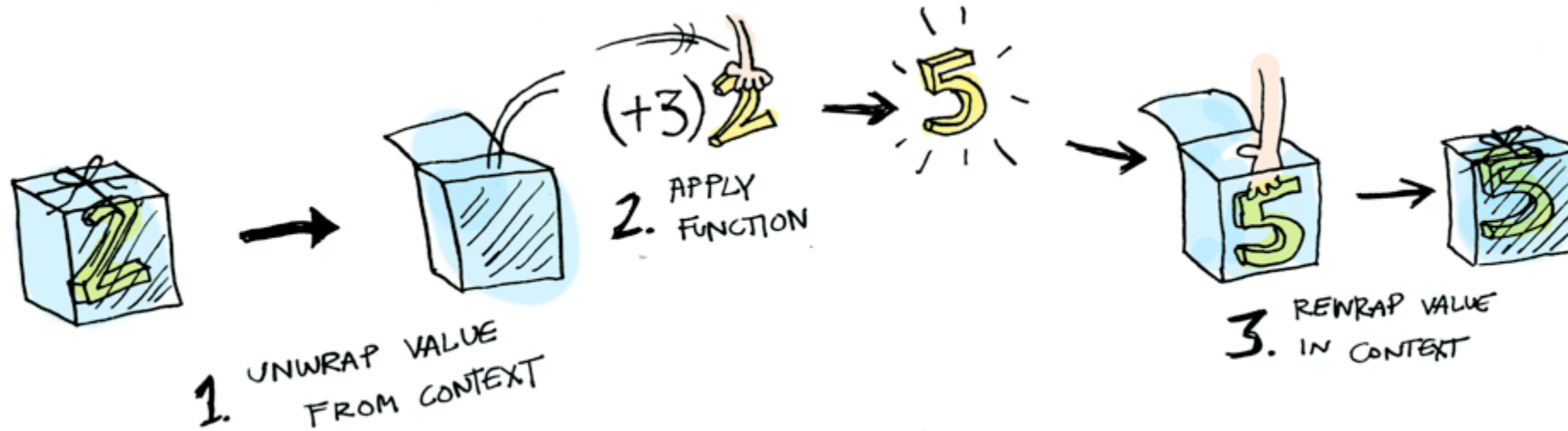
AARHUS
UNIVERSITY

ORAL EXAM: SWAFP – LWSHARP



MONADS AND FUNCTORS

WHAT IS A FUNCTOR?



WHAT IS A FUNCTOR?

Mapping over a
wrapped value
while preserving
structure

Core function:
map

Example types:
List<'a>,
Option<'a>, etc.

FUNCTOR LAWS

Identity law

- $\text{Map id} = \text{id}$
- Identity function (id)

Composition law

- $\text{Map } (f \gg g) = \text{map } f \gg \text{map } g$

A BASIC FUNCTOR

```
let f x = x + 1

let value1 = Some 3
let value2 = None

let result1 = Option.map f value1
// Some 4

let result2 = Option.map f value2
// None
```



FUNCTOR IN LWSHARP

- Will show it directly in Rider

```
let mapForShowCase f m : Computation<'b> =  
    fun store ->  
        match m store with  
        | Error err -> Error err  
        | Ok (value, store') -> Ok (f value, store')
```

WHAT IS A MONAD?

Design pattern which can combine code fragments

Apply a function that returns a wrapped value to a wrapped value

Functions needed (return and bind)

Look for function that performs some side-effects or can fail

WHAT IS A MONAD?

```
return : 'a -> M<'a>
```

$(\gg=) :: ma \rightarrow (a \rightarrow mb) \rightarrow mb$

1. $\gg=$ TAKES
A MONAD
(LIKE **Just 3**)

2. AND A
FUNCTION THAT
RETURNS A MONAD
(LIKE **half**)

3. AND IT
RETURNS
A MONAD

BASIC MONAD

```
// return : 'a -> Option<'a>
let returnValue x = Some x

// bind : Option<'a> -> ('a -> Option<'b>) -> Option<'b>
let bind opt f =
    match opt with
    | Some x -> f x
    | None -> None

let safeDivide x y =
    if y = 0 then None
    else Some (x / y)

let addOne x = Some (x + 1)

let result =
    Some 10 >=> (fun x -> safeDivide x 2) >=> addOne
```



MONADS IN LWSHARP

```
type Computation<'a> = Store -> Result<'a * Store, RuntimeError>

let run f store : Result<'a * Store, RuntimeError> =
    f store

let returnValue (x: 'a) : Computation<'a> = fun store -> Ok (x, store)

let bind (m: Computation<'a>) (f: 'a -> Computation<'b>) : Computation<'b> =
    fun store ->
        match m store with
        | Error err -> Error err
        | Ok (value, newStore) -> f value newStore
```

TESTING AND RECURSIVE FUNCTIONS

WHAT IS PROPERTY BASED TESTING?

Problem with Unit tests

- Test for a **specific** example

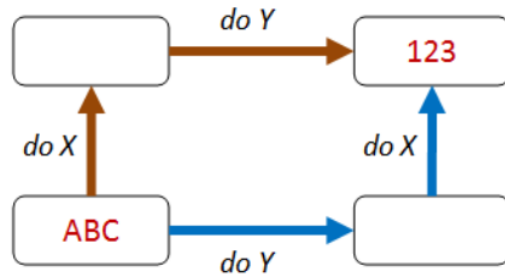
Property based testing

- Tests general **properties** that should always hold
- Works for many automatically generated (valid) inputs
- Uses random input (generation) and must meet specific requirements

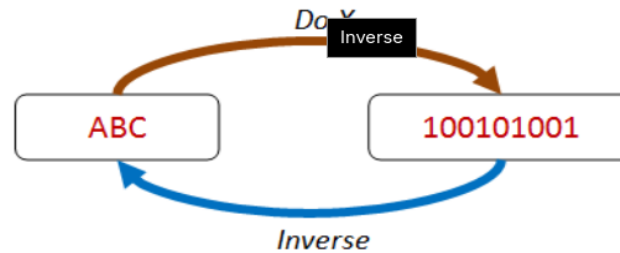
WHAT IS A PROPERTY?

-
- **Something** that should always be true about a function, no matter which valid input
 - A **rule** or **law** about behavior of the function

HOW DO WE FIND PROPERTIES?



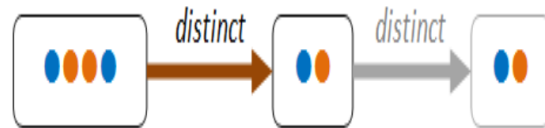
NEGATION



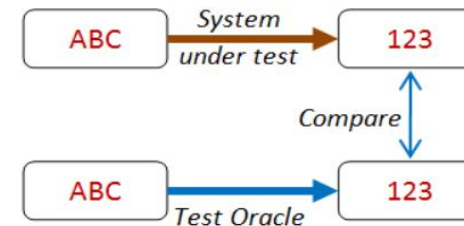
NON-CHANGING



DISTINCT



TEST ORACLE



HOW TO IMPLEMENT?



```
let sort (xs: list<int>) : list<int> =  
    List.sort xs
```



OLD STYLE (UNIT TESTING)

```
open Xunit

[<Fact>]
let ``Sorting [3; 1; 2] gives [1; 2; 3]`` () =
    let input = [3; 1; 2]
    let expected = [1; 2; 3]

    let result = List.sort input

    Assert.Equal(expected, result)
```



PROPERTY-BASED TESTING

```
open FsCheck
open FsCheck.Xunit

[<Property>]
let ``Sorting a list does not change its length`` (xs: list<int>) =
    List.sort xs |> List.length = List.length xs

[<Property>]
let ``Sorting a list twice gives the same result as sorting once`` (xs: list<int>) =

    List.sort (List.sort xs) = List.sort xs
```

WHY PROPERTY BASED TESTING?

Test **general**
rules not specific
examples

Randomized
generation finds
edge cases

Properties help
defining what
must be true

WHAT IS A RECURSIVE FUNCTION?

- A function that **calls itself** to solve a smaller instance of the problem
- There is a **Base case** that ensures recursion stops

HOW TO IMPLEMENT

```
let rec factorial n =  
  if n <= 1 then 1  
  else n * factorial (n - 1)  
  
let rec fib n =  
  if n <= 1 then n  
  else fib (n-1) + fib (n-2)
```



TYPES OF RECURSION

1. Simple recursion – example from before
2. Mutual recursion – functions call each other recursively
3. Tail Recursion – last operation is the recursive call (optimized)

TAIL RECURSION

```
let factorialTR n =  
  let rec loop acc n =  
    if n <= 1 then acc  
    else loop (acc * n) (n - 1)  
  loop 1 n
```

WHY RECURSION?

- Expresses **divide and conquer**

REACTIVE PROGRAMMING

WHAT IS REACTIVE PROGRAMMING?

Core idea

- Instead of manually updating the state, you define relationships between values.

Analogy

- A Spreadsheet cell in Excel, if you change one cell all dependent formulas update automatically

Imperative vs. Reactive

- Imperative: Do this first and then update
- Define relationships, the system ensures updates happen automatically

WHAT IS AN EVENT STREAM?

Definition

- Continuous sequence of events/data arriving over time

Stream vs. Static

- Streams are ongoing and dynamic

Push Model

- Data is pushed to the subscribers (Observers)
- Responds to changes automatically rather than polling

HOW TO IMPLEMENT?

```
open System
open System.Timers

let createSensorAndObservable timerInterval =
    let random = Random()
    let timer = new Timer(float timerInterval)
    timer.AutoReset <- true

    let observable =
        timer.Elapsed
        |> Observable.map (fun _ -> 50.0 + random.NextDouble() * 50.0)

    let task = async {
        timer.Start()
        do! Async.Sleep 5000
        timer.Stop()
    }

    (task, observable)

let sensorTask, sensorStream = createSensorAndObservable 1000

sensorStream
|> Observable.filter (fun value -> value > 80.0)
|> Observable.subscribe (fun value -> printfn "High sensor reading: %f" value)
|> ignore

Async.RunSynchronously sensorTask
```



WHY USE REACTIVE PROGRAMMING?

Key Benefits

- Automatic event reaction
- No polling
- Possibility to compose multiple asynchronous events
- Simplifies concurrency



WHAT IS AN AKKA STREAM?

Design

- Processing data streams of data asynchronously and non-blocking

Core concepts

- Source produces data (sensor data, timer etc.)
- Flow processes or transforms data (e.g filter)
- Sink consumes the data (logs, DB, UI)
- Actor based, can hold states
- Can be distributed

Difference to reactive programming

- **Reactive Programming:** reacts to discrete events; simple, push-based, declarative
- **Akka Streams:** structured pipelines (Source → Flow → Sink) with built-in backpressure
- **Key Difference:** Akka Streams handle **continuous, high-throughput data flows** reliably, while reactive programming handles **individual events**



MONOIDS AND MODEL/TYPE-BASED PROGRAMMING



WHAT IS A MONOID?

Definition

- A "type" with a way to combine elements safely

Rules

- **Closure:** Combining two elements of the type produces another element of the same type
- **Associativity:** The grouping order of operations does not matter
- **Identity:** There is an element such that combining it with any element does nothing.

SIMPLE MONOID

```
let add x y = x + y
let zero = 0

// Associativity
let example1 = add (add 1 2) 3
let example2 = add 1 (add 2 3)

printfn "Associativity: %b" (example1 = example2)
printfn "Identity: %b" (add 5 zero = 5)
```



CUSTOM MONOID

```
type Config = { MaxConnections: int; Timeout: int }

let combineConfig c1 c2 =
    { MaxConnections = c1.MaxConnections + c2.MaxConnections
      Timeout = max c1.Timeout c2.Timeout }

let emptyConfig = { MaxConnections = 0; Timeout = 0 }

let config1 = { MaxConnections = 10; Timeout = 30 }
let config2 = { MaxConnections = 5; Timeout = 20 }

let combinedConfig = combineConfig config1 config2
printfn "Combined Config: %A" combinedConfig // { MaxConnections = 15; Timeout = 30 }
```



WHAT IS A MODEL/TYPE-BASED PROGRAMMING?

Core idea

- Model- and type-based programming means using the type system to model the domain and its rules, so that invalid states and illegal operations are impossible to represent.

Key points

- Types are the domain model not just data containers
- Business rule are directly encoded in the types
- Program states are explicit
- The compiler enforces correctness at compile time

ACTIVE PATTERNS

```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd
```

```
let TestNumber input =  
  match input with  
  | Even -> printfn "%d is even" input  
  | Odd -> printfn "%d is odd" input
```

```
TestNumber 7  
TestNumber 11  
TestNumber 32
```

```
//7 is odd  
//11 is odd  
//32 is even
```



DIFFERENT TYPES

Abbrev: Type
alias

Product Type:
Tuples

Sum:
Discriminated
Unions

Enumeration
(not type safe,
because just
named integers)

Records:
Aggregates of
named values



HOW TO IMPLEMENT

```
type ProgramResult =  
  { FilePath: string  
    Success: bool  
    Store: State.Store  
    Error: PipelineError option }
```

```
type Var = string  
type Value = int  
  
type Expr =  
  | Const of Value  
  | Var of Var  
  | Add of Expr * Expr  
  | Sub of Expr * Expr  
  | Mul of Expr * Expr  
  | Div of Expr * Expr
```

ENFORCING DOMAIN RULES

```
module starp.Domain.User
open starp.Domain.PrimitiveTypes
open starp.Domain.FitnessProfile
open System

type User = {
    Id: UserId
    Name: Name
    Email: EmailAddress
    Profile: FitnessProfile option
}

module User =
    let createUser (name: Name) =
        { Id = UserId(Guid.NewGuid())
          Name = name
          Email = EmailAddress ""
          Profile = None }

    let updateUserProfile user profile =
        { user with Profile = Some profile }
```

```
type EmailAddress = EmailAddress of string
module Email =
    let isValidEmail (EmailAddress email) =
        let emailRegex = System.Text.RegularExpressions.Regex(@"^[^\s]+@[^\s]+\.[^\s]+$")
        emailRegex.IsMatch(email)

    let createEmailAddress email =
        if isValidEmail (EmailAddress email) then Ok (EmailAddress email)
        else Error InvalidEmail
```



WHY?

Types

- Help us understand the domain
- Domain rules are enforced
- Easier to refactor
- An explicit TODO

Documentation

- Readable by non-programmers
- Defines a common language we can use to talk about the code

AKKA

WHAT IS AKKA?

Core idea

- The Akka framework allows for building a system out of many tiny, independent workers that talk only by sending messages

Functionalities

- Actor creation and lifecycle management
- Supervision and fault recovery
- Scalability across threads, processes and machines

WHAT IS THE ACTOR MODEL ?

Core idea

- The actor model replaces shared-memory concurrency with messages and actors for concurrent computing

Principles

- Concurrency: Actors run independently
- Isolation: Actors encapsulate their own state
- Asynchronous communication

WHAT IS AN ACTOR?

Core idea

- An actor is a tiny autonomous program with a mailbox, private state and behavior

Functionalities

- Receives messages asynchronously
- Processes messages sequentially
- Sends messages to other actors
- Create new Actors
- Change own behavior or state

BASIC EXAMPLE

```
open Akka.Actor
open Akka.FSharp

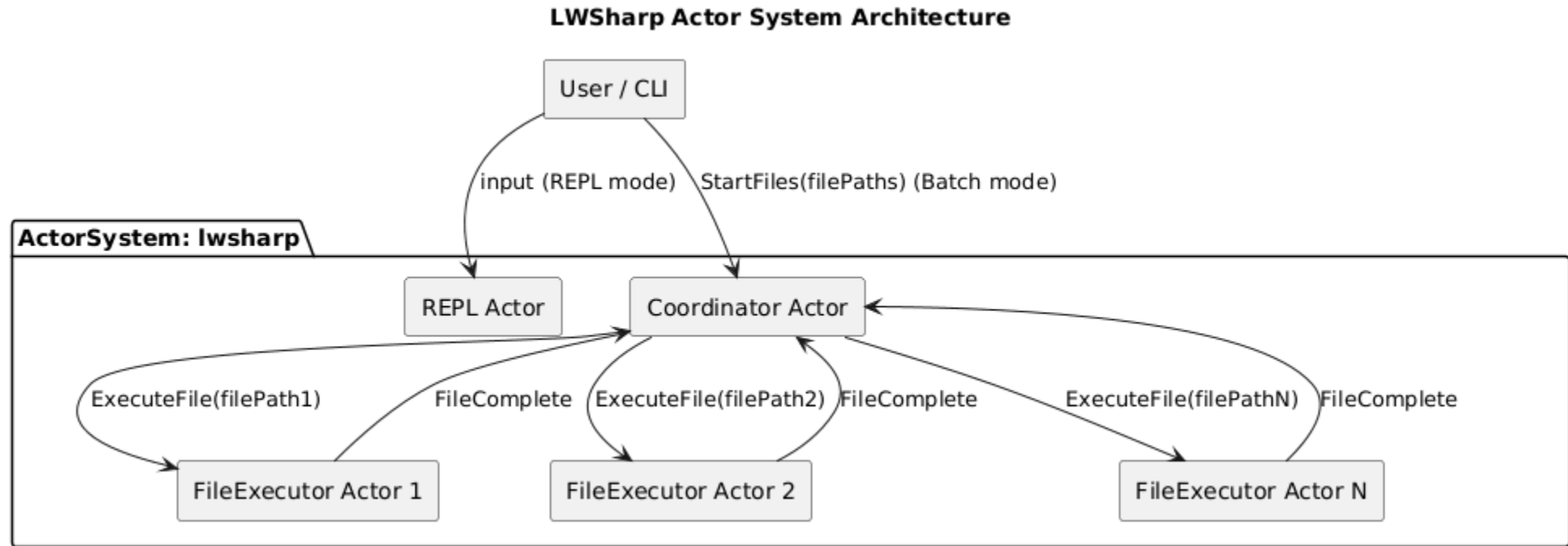
let greeter (mailbox: Actor<_>) =
    actor {
        let! msg = mailbox.Receive()
        printfn "Hello %s" msg
    }

let system = System.create "system" (Configuration.defaultConfig())
let greeterRef = spawn system "greeter" greeter

greeterRef <! "World"
```



HOW TO IMPLEMENT?



WHY?

Simpler concurrency

- No locks
- No race conditions
- Deterministic message handling

Scalability

- Actors are lightweight
- Millions of actors possible
- Automatic scheduling on thread pools
- Easy horizontal scaling

Fault tolerance

- Actors fail independently
- Supervisor restart, stop and resume actors

Location Transparency

- Actors work locally, across processes and across machines



FUNCTIONAL ARCHITECTURE

WHAT IS A FUNCTIONAL ARCHITECTURE?

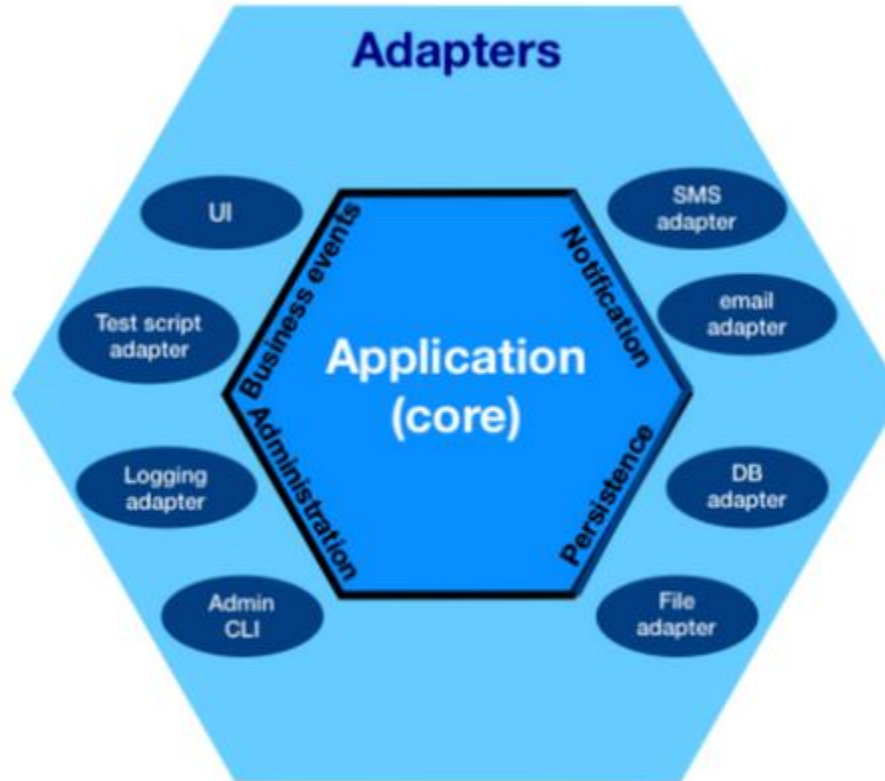
Problem

- Layered Architecture violates the Dependency Inversion Principle (DIP)
- BLL depends on the DAL
- BLL calls impure code

Core idea

- Functional architecture is about separating pure domain logic from impure infrastructure

WHAT IS THE HEXAGONAL ARCHITECTURE?



WHAT IS THE HEXAGONAL ARCHITECTURE?

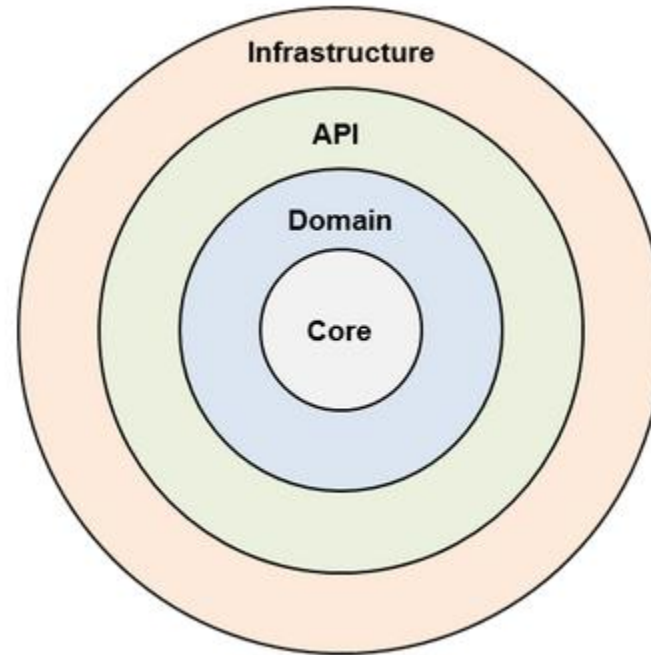
Ports

- Interfaces or abstraction defining what the domain needs or offers without the concrete implementation
- Express capabilities and not implementation details

Adapter

- A concrete implementation of a port
- Implement code with side effects

WHAT IS THE ONION ARCHITECTURE?



WHAT IS THE HEXAGONAL ARCHITECTURE?

Definition

- The system is layers where dependencies always point inward toward the domain
- The inner circles have no knowledge about the outer circles

Pros and Cons

- It has fewer artifacts than the Ports and Adapters
- The database and UI are in the same layer meaning they have access to each other

ERROR HANDLING

TRADITIONAL WAY

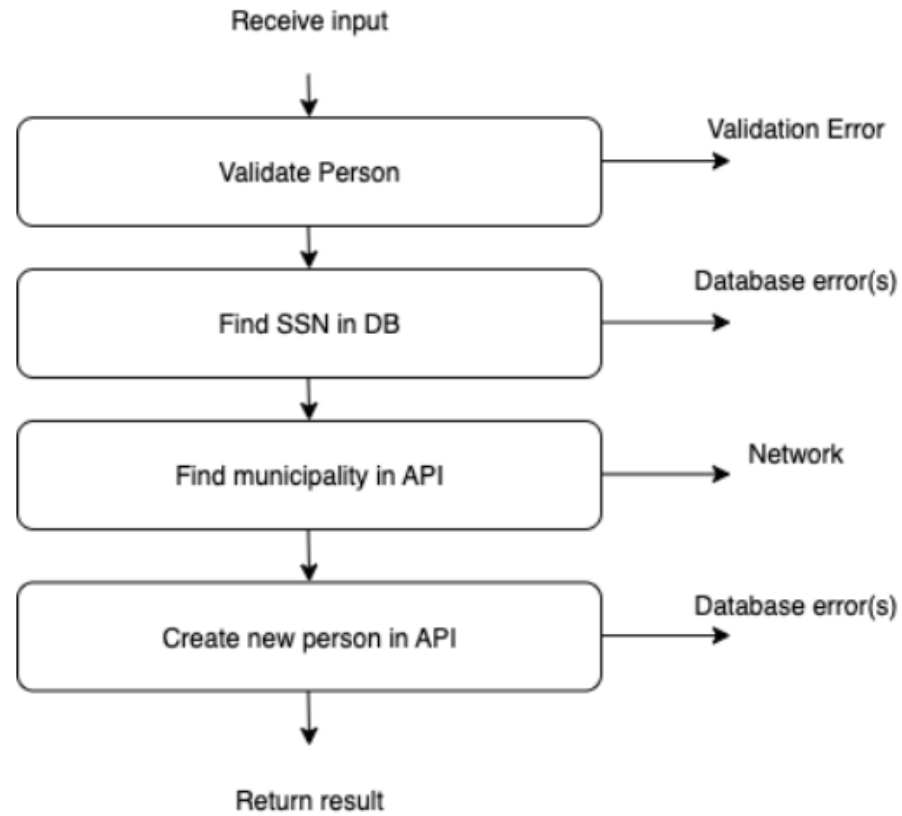
Exceptions
everywhere

Nested if/
try-catch

Hard to
compose

Hard to test

PROBLEMS

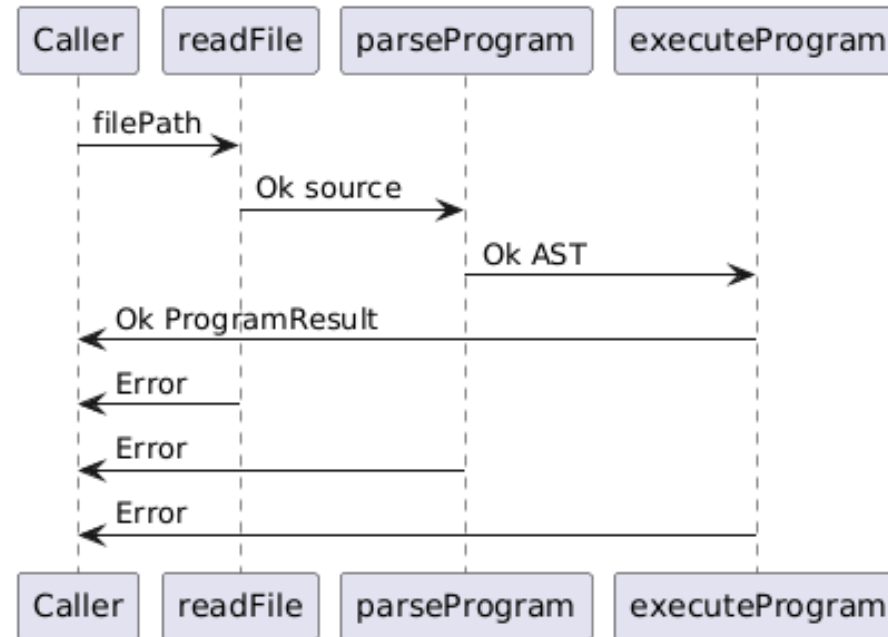


NESTED PATTERN MATCHING

```
let executeProgramFile (fileReader: IFileReader) (parser: IParser) (filePath: string) :
Result<ProgramResult, string> =
  match readFile fileReader filePath with
  | Ok source ->
    match parseProgram parser source with
    | Ok ast ->
      match executeProgram ast with
      | Ok store ->
        Ok { FilePath = filePath
              Success = true
              Store = store
              Error = None }
      | Error execErr ->
        Error execErr
    | Error parseErr ->
        Error parseErr
    | Error readErr ->
        Error readErr
  return res ? go(f, res[1], acc.concat([res[0]])) : acc
}
```



RAILWAY ORIENTED PROGRAMMING



BENEFITS OF ROP

- ROP makes error handling explicit, composable and impossible to ignore
- All following steps are skipped if one fails
- Removes the need for nested pattern matching
- **Explicit control flow:**
 - The Failure is visible, not like exception
 - One must handle the errors
 - The compiler helps you

PERSISTENT DATA STRUCTURES

WHAT ARE PDS?

Key ideas

- No mutation
- Older versions remain available
- Structural sharing instead of copying the whole structure

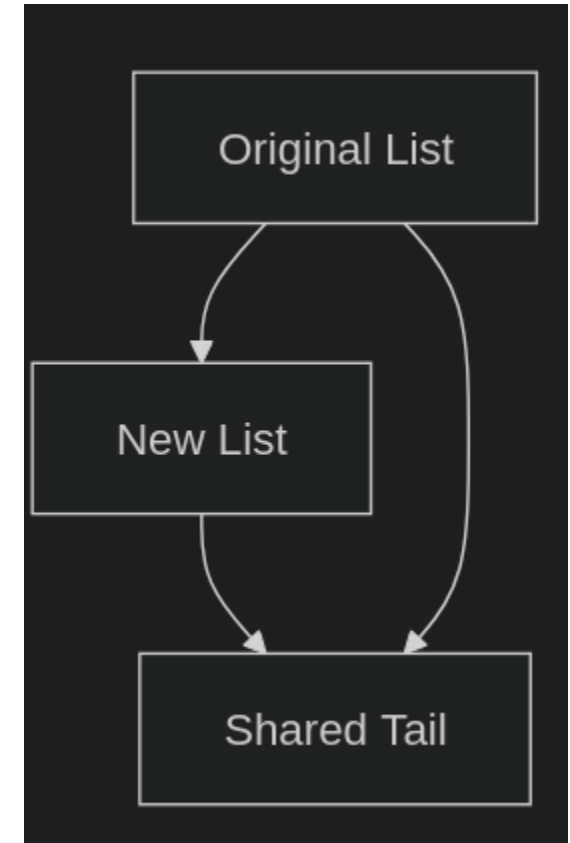
Benefits

- **Ease of Reasoning:** Simplifies reasoning about code
- **Thread-safety:** automatically guaranteed
- **Predictability:** Functions become predictable always produce the same output for the same input

STRUCTURAL SHARING

```
let originalList = [1; 2; 3]
let newList = 0 :: originalList
```

```
// originalList remains [1; 2; 3]
// newList is [0; 1; 2; 3]
```



MAP

Keys must be unique

Key-value pair storage

Lookup is via the key

Immutable

It is implemented via Balanced Binary Tree

SEQUENCES

```
let naturals = Seq.initInfinite id

// Only the 5th element computed
Seq.item 5 naturals

let cached = Seq.cache naturals

Seq.item 5 cached
Seq.item 5 cached // no need to recompute
```

EQUALITY

The = operator is defined for List, Sets and Maps

Equals mean same elements
(**structural equality**)

e.g List the order of elements matters

APPLICATIVES AND FUNCTIONS

WHAT IS A FUNCTION?

```
let square x = x * x

let add x y = x + y

let describeNumber x =
  match x with
  | 0 -> "Zero"
  | 1 -> "One"
  | _ -> "Other"
```



FUNCTION CURRYING

```

//let add = fun x -> fun y -> x + y
let add x y = x + y

// let add5 = int -> int
let add5 = add 5

printfn "add 2 3 = %d" (add 2 3)
printfn "add5 10 = %d" (add5 10)
```

FIRST CLASS CITIZENS

```
let square x = x * x
let result1 = square 5

let applyTwice f x = f (f x)
let result2 = applyTwice square 3

let adder x = (fun y -> x + y)
let add5 = adder 5
let result3 = add5 10
```

PIPING AND COMPOSITION



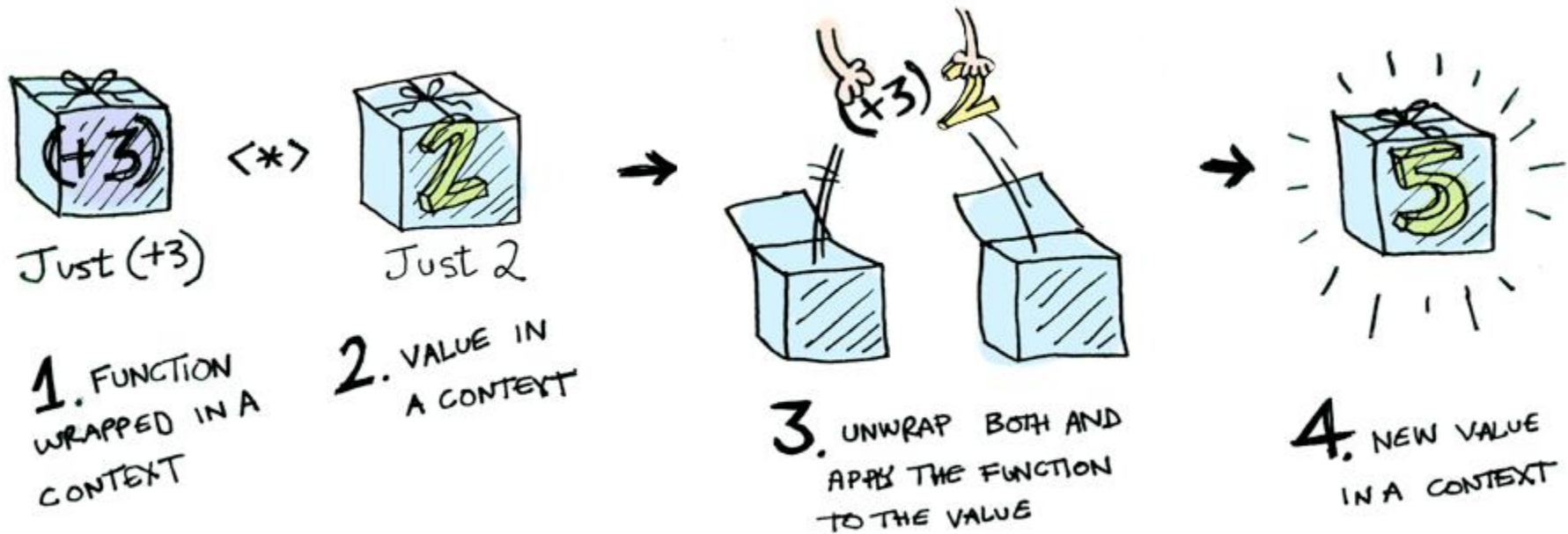
```
let add1 x = x + 1  
let square x = x * x
```

```
let resultPipe = 3 |> add1 |> square
```

```
let addThenSquare = add1 >> square  
let resultCompose = addThenSquare 3
```



WHAT ARE APPLICATIVES?



WHAT ARE APPLICATIVES?

Applicatives Laws

- **Identity:** $\text{id} \langle * \rangle v = v$
- **Composition:** $u \langle * \rangle v \langle * \rangle w = u \langle * \rangle (v \langle * \rangle w)$
- **Homomorphism:** $\text{pure } f \langle * \rangle v = \text{pure } (f \ v)$
- **Interchange:** $u \langle * \rangle \text{pure } y = \text{pure } (\text{fun } a \rightarrow a \ y) \langle * \rangle u$

Using Applicatives

- Apply a list of functions to some arguments
- When having a function with multiple arguments and they are wrapped in context

BASIC APPLICATIVE

```
module List =  
  let apply fs xs = fs |> List.collect (fun f -> xs |> List.map f)  
  
  let a = [1; 2]  
  let b = [10; 20]  
  
  let sums = List.apply (List.apply [fun x y -> x + y] a) b  
  // Partial: [fun y -> 1 + y; fun y -> 2 + y]  
  // Output: [11; 21; 12; 22]
```



**THANK YOU FOR YOUR
ATTENTION** – NOW TO THE
SCARY PART



AARHUS
UNIVERSITY