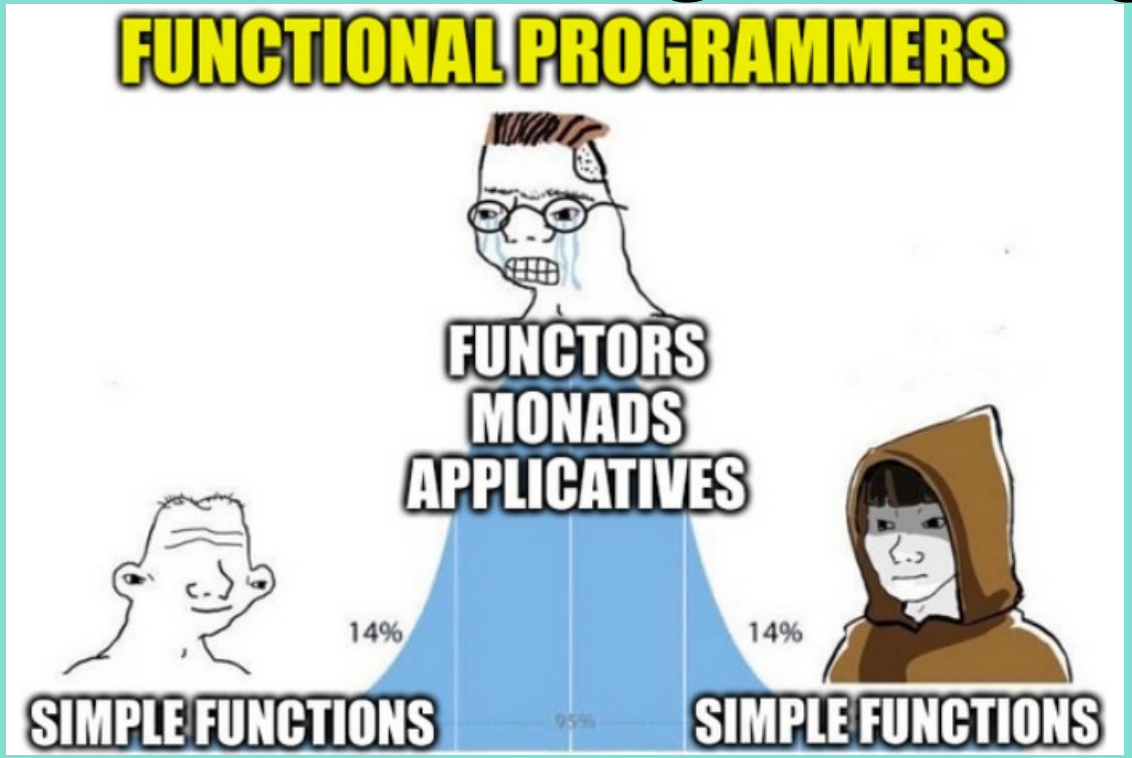
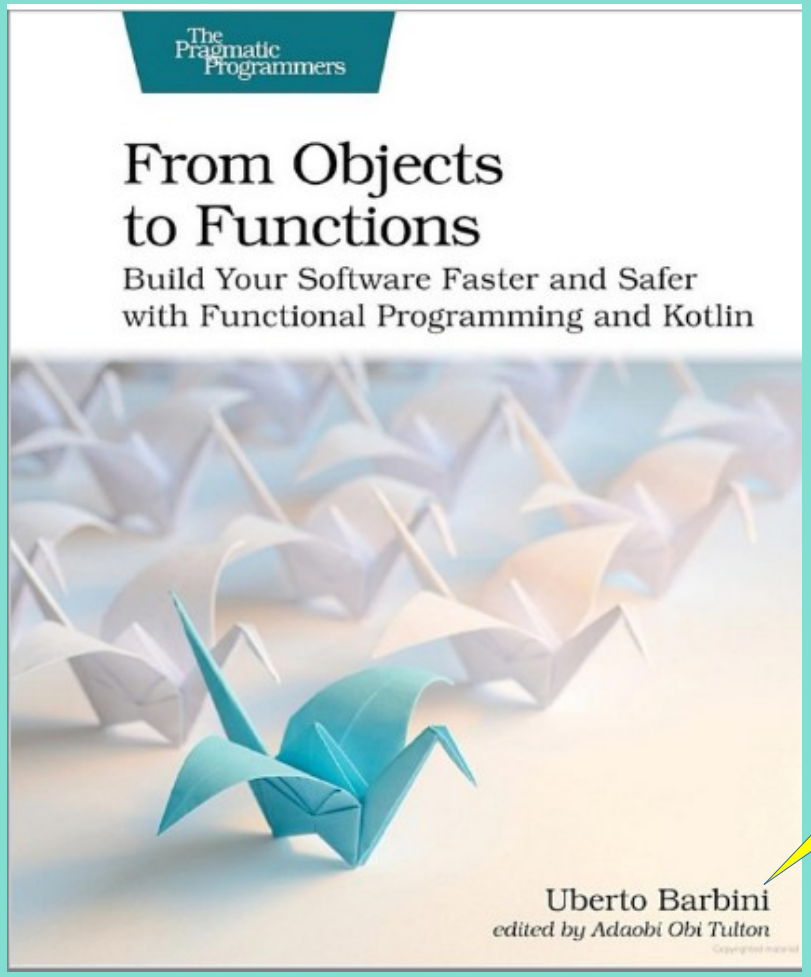
Let's Put the Fun Back into Functional Programming!



Uberto Barbini @ramtop

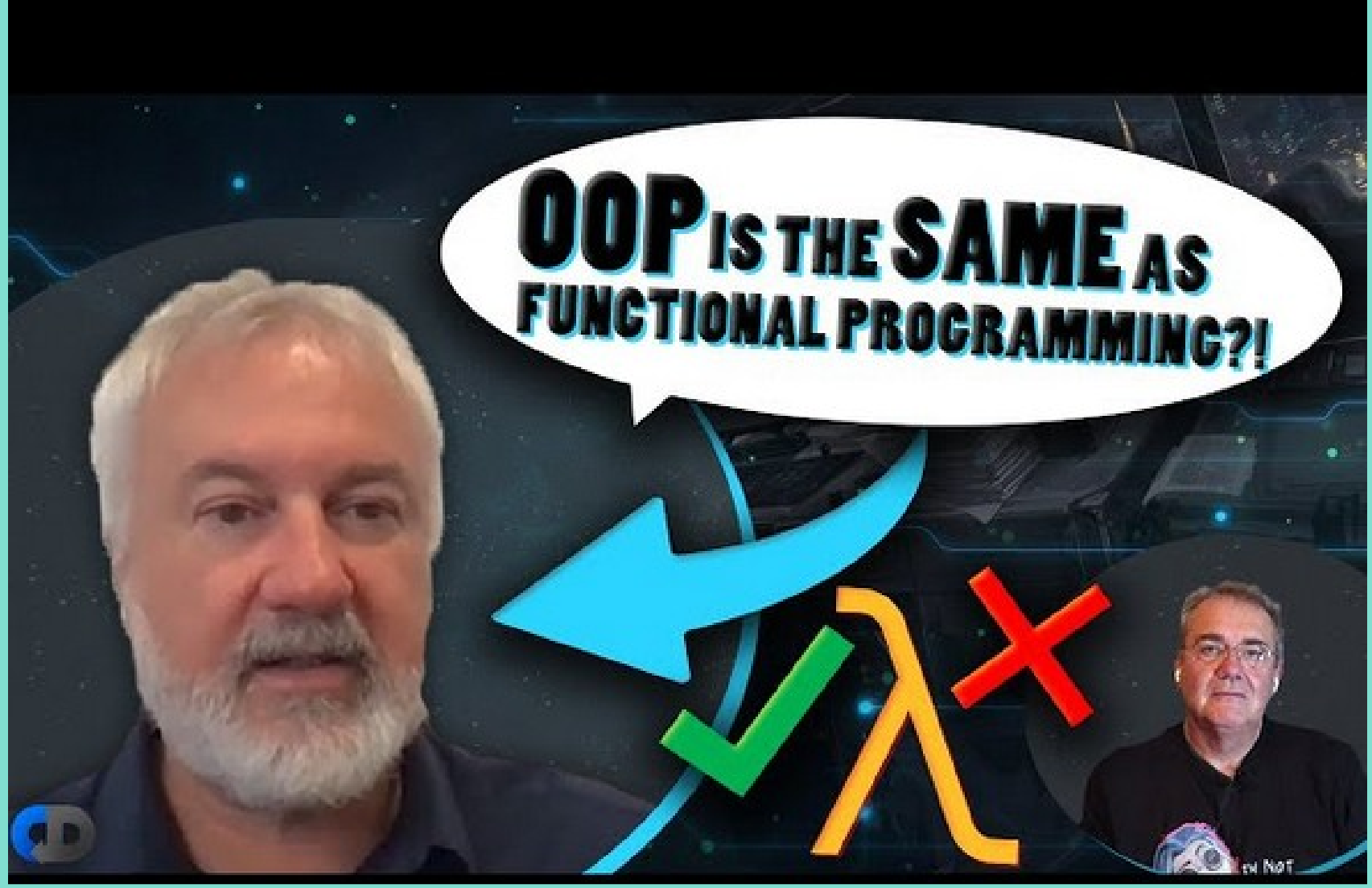
July 2019 - October 2023



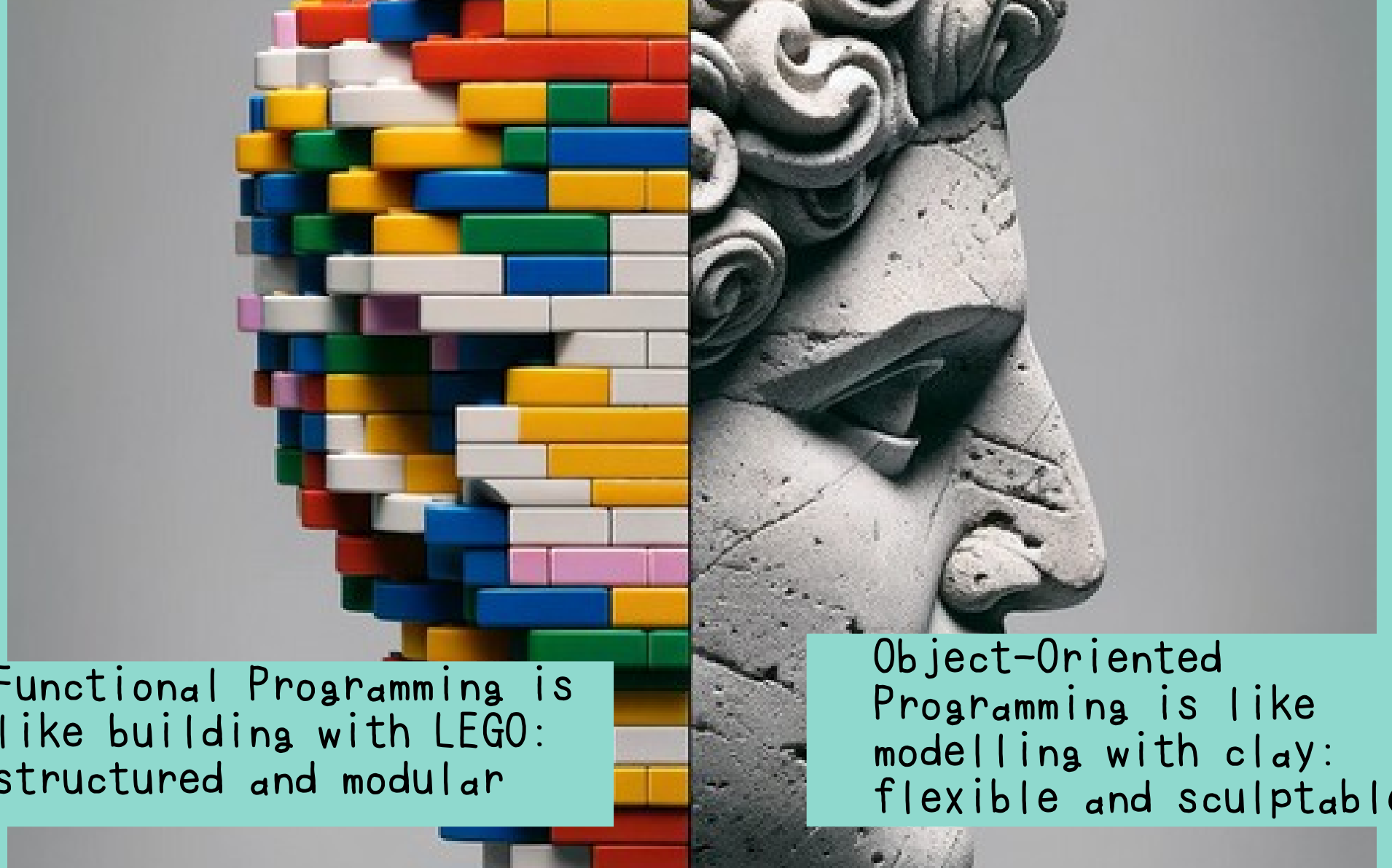
That's me!

How Do You Solve a Big Problem?

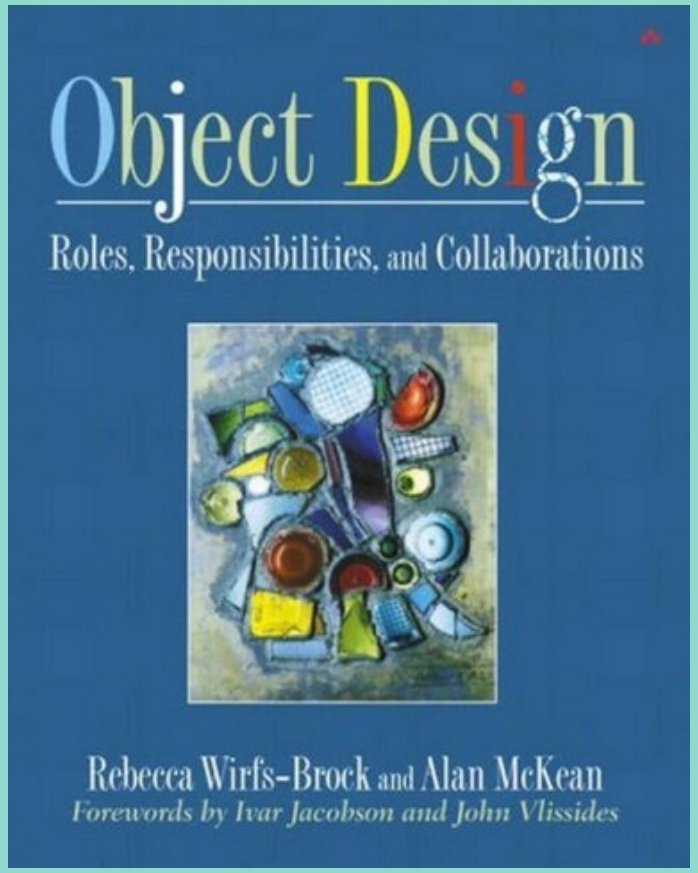

BIG
PROBLEM
TASK 1
TASK 2
TASK 3



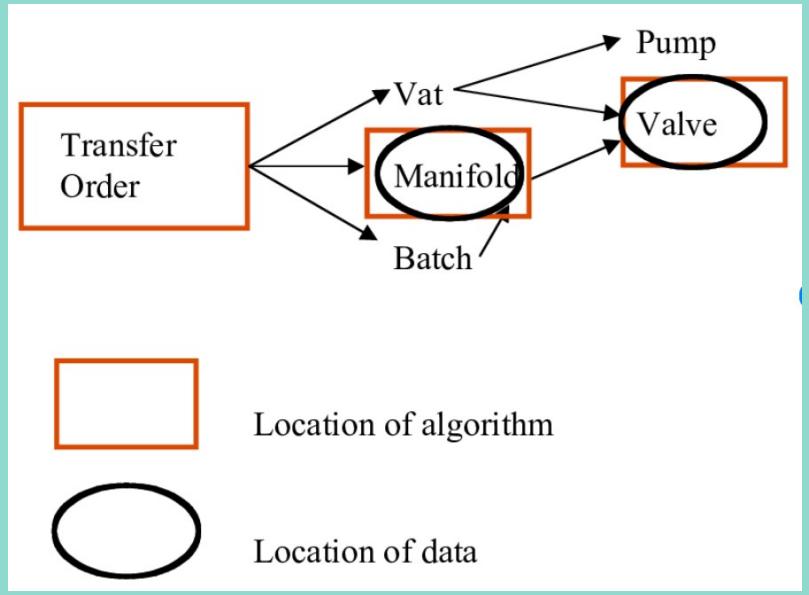
https://www.youtube.com/watch?v=j71n33A0CkI

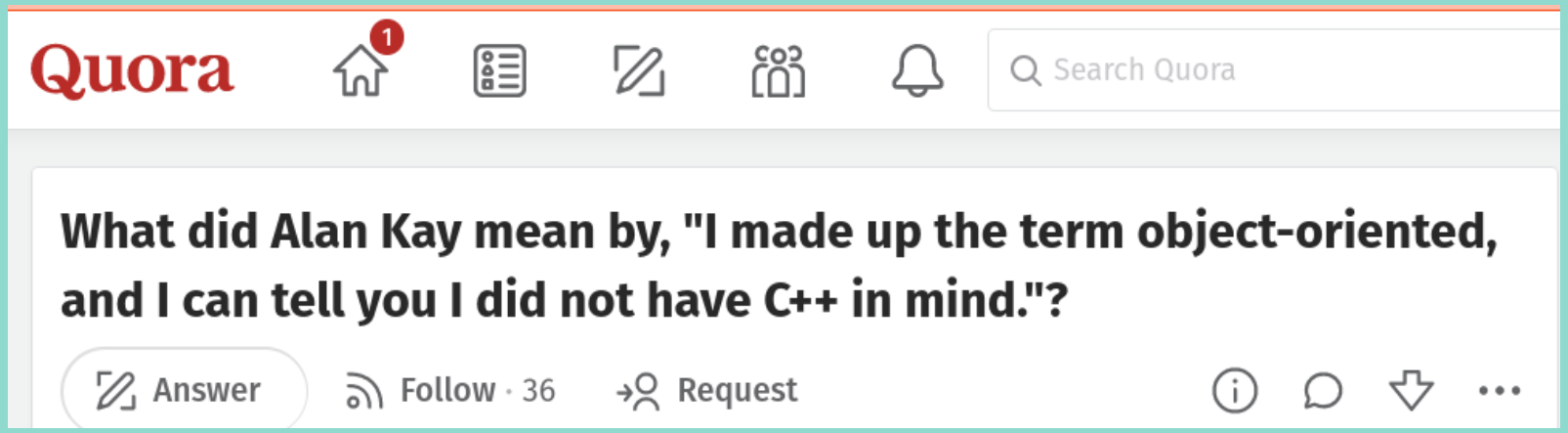

Functional Programming is like building with LEGO: structured and modular
Object-Oriented Programming is like modelling with clay: flexible and sculptable

Object Oriented Design



Possibly the best book to learn Object Design by Rebecca Wirfs-Brock





As explained elsewhere on Quora, and in “The Early History of Smalltalk”, I had chance encounters with Sketchpad and Simula in my first week of grad school in late 66, that shocked me into a realization about “computers as basic and universal units” via the connections and parallels with other like things, such as biological structures, computers on networks, processes in time-sharing systems, general systems of parts intercommunicating, and so forth. I started to think about dynamic languages to make such processes, and how the processes could be made efficient and parsimonious enough to be universal.

Someone asked me what I was doing, and without thinking, I said “object-oriented programming”. (A very bad choice as it turned out, for many reasons.)

Functional Design

Why Functional Programming Matters

John Hughes, Institutionen för Datavetenskap,
Chalmers Tekniska Högskola.

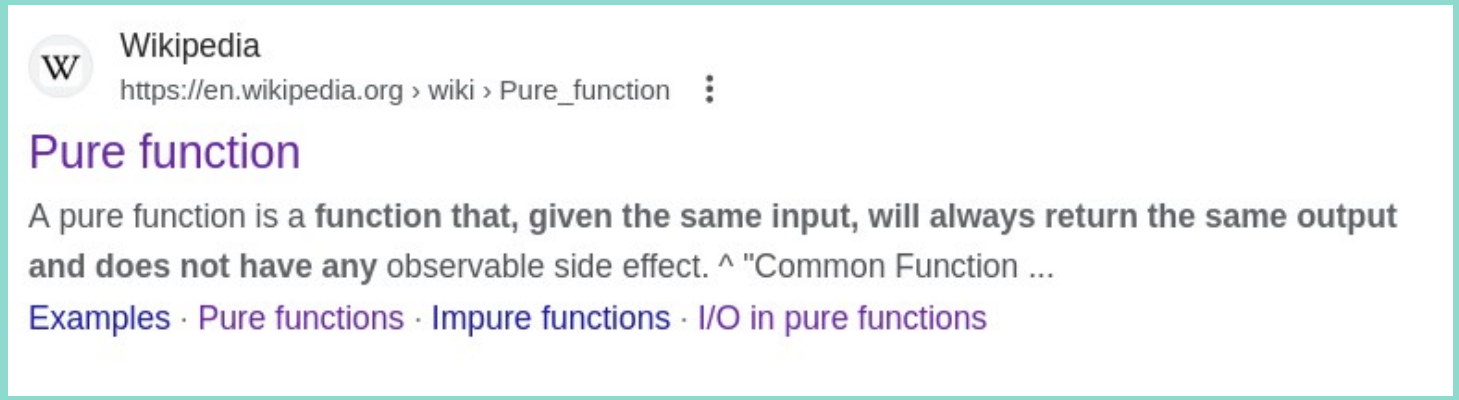
1 Introduction

This paper is an attempt to demonstrate to the "real world" that functional programming is vitally important, and also to help functional programmers exploit its advantages to the full by making it clear what those advantages are.

Functional programming is so called because a program consists entirely of functions. The main program itself is written as a function which receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives. These functions are much like ordinary mathematical functions, and in this paper will be defined by ordinary equations. Our

Pure Functions

The special characteristics and advantages of functional programming are often summed up more or less as follows. Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant - since no side-effect can change the value of an expression, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa - that is, programs are "referentially transparent".



Why Kotlin?

Why Kotlin?


```java
List<String> stringList = intList.stream()
        .map(String::valueOf)
        .collect(Collectors.toList());
```

```kotlin
val stringList = intList.map { it.toString() }
```

Why Kotlin?

```java
public static boolean isEven(Optional<Integer> optInt)
{
    return optionalInt.isPresent()
        && optionalInt.get() % 2 == 0;
}
```


```kotlin
fun isEven(number: Int?) =
       number?.let { it % 2 } == 0
```

Why Kotlin?

```java
public static <A, B, C> Function<A, C> compose(
            Function<A, B> f1,
            Function<B, C> f2) {
    return x -> f2.apply(f1.apply(x));
}
```


```kotlin
fun <A, B, C> compose(f1: (A)->B, f2: (B)->C): (A)->C =
   { f2(f1(it)) }
```

Kotlin


LET'S START
CODING

```
fun main() {
    val userView = UserView()
    val userService = UserService()
    val controller = UserController(userService, userView)

    embeddedServer(Netty, port = 8080) {
        routing {
            staticResources("/static", "static")
            get("/") {
                call.respond(HtmlContent(HttpStatusCode.OK, userView.indexHtml()))
            }
            get("/users") {
                call.respond(controller.getAllUsersPage())
            }
            get("/user/{id}") {
                val id = call.parameters["id"]?.toIntOrNull()
                call.respond(controller.getUserPage(id))
            }
        }
    }.start(wait = true)
}
```

Service is the Model facade

Controller is connected to Model and View

Http routes

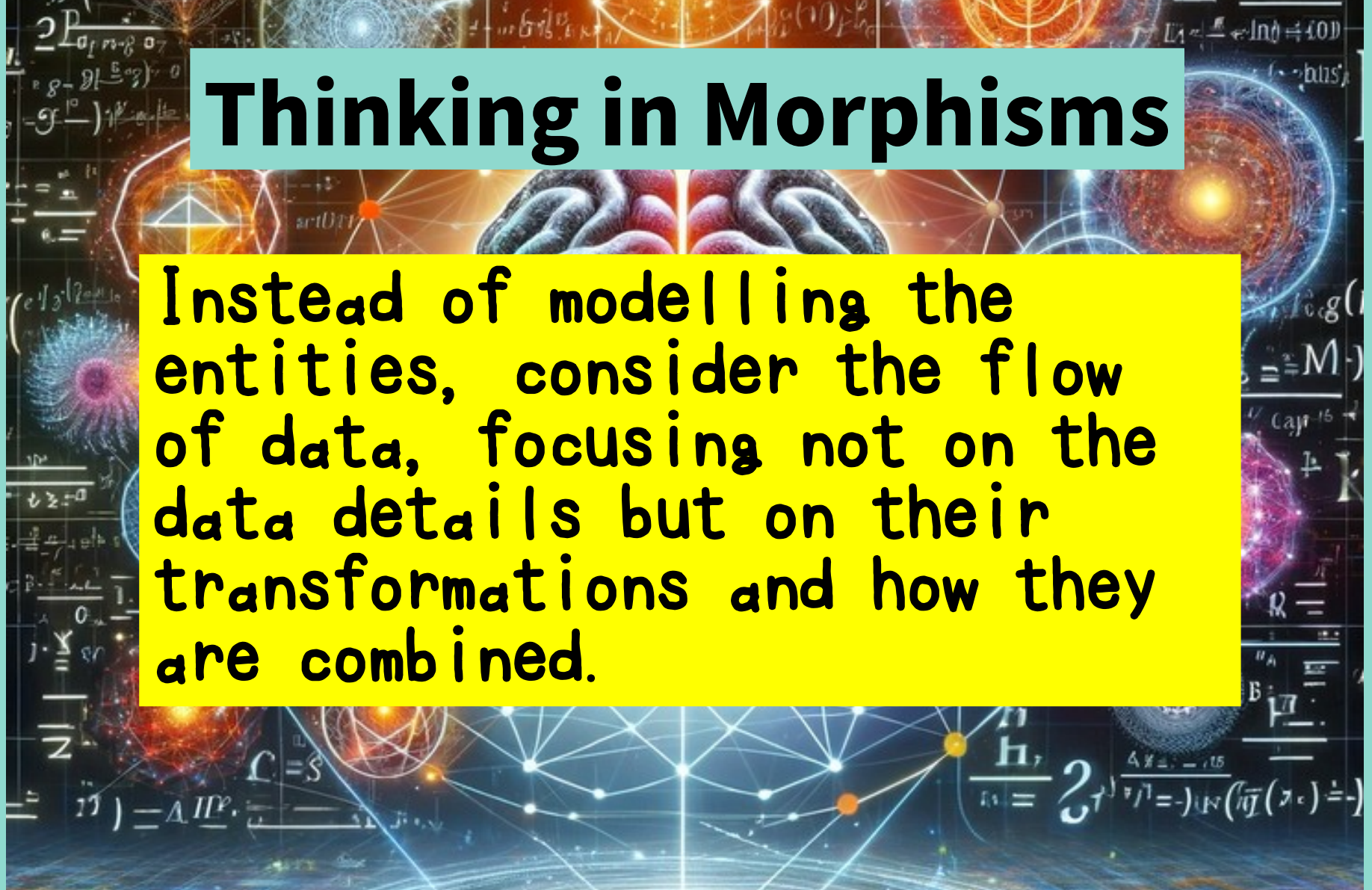
Controller get called with request parameters and will render the page

```
class UserController(private val userService: UserService,
                     private val userView: UserView) {

    fun getAllUsersPage(): HtmlContent {
        val users = userService.getAllUsers()
        return HtmlContent(HttpStatusCode.OK, userView.usersPage(users))
    }

    fun getUserPage(id: Int?): HtmlContent {
        if (id == null) {
            return HtmlContent(HttpStatusCode.BadRequest,
                userView.errorPage("Invalid ID format"))
        }

        val user = userService.getUserById(id)
        if (user != null) {
            return HtmlContent(HttpStatusCode.OK, userView.userPage(user))
        } else {
            return HtmlContent(HttpStatusCode.NotFound,
                userView.errorPage("User not found"))
        }
    }
}
```

Get request parameter

Pass to the the Service

Ask the view to render the page


Thinking in Morphisms
Instead of modelling the entities, consider the flow of data, focusing not on the data details but on their transformations and how they are combined.

```kotlin
class UserController(private val userService: UserService,
                     private val userView: UserView) {

    fun getAllUsersPage(): HtmlContent {
        val users = userService.getAllUsers()
        return HtmlContent(HttpStatusCode.OK, userView.usersPage(users))
    }

    fun getUserPage(id: Int?): HtmlContent {
        if (id == null) {
            return HtmlContent(HttpStatusCode.BadRequest,
                userView.errorPage("Invalid ID format"))
        }

        val user = userService.getUserById(id)
        if (user != null) {
            return HtmlContent(HttpStatusCode.OK, userView.userPage(user))
        } else {
            return HtmlContent(HttpStatusCode.NotFound,
                userView.errorPage("User not found"))
        }
    }
}
```

```
class UserController(private val userService: UserService) {


    fun getAllUsersPage(): HtmlContent {
        val users = userService.getAllUsers()
        return HtmlContent(HttpStatusCode.OK, usersPage(users))
    }

    fun getUserPage(id: Int?): HtmlContent {
        if (id == null) {
            return HtmlContent(HttpStatusCode.BadRequest,
                errorPage("Invalid ID format"))
        }

        val user = userService.getUserById(id)
        if (user != null) {
            return HtmlContent(HttpStatusCode.OK, userPage(user))
        } else {
            return HtmlContent(HttpStatusCode.NotFound,
                errorPage("User not found"))
        }
    }

}
```

No more a reference to a View object

Just simple functions

```
class UserService {

    fun getAllUsers(): List<User> = transaction {
        Users.selectAll().map {
            User(
                id = it[Users.id].value,
                name = it[Users.name],
                dateOfBirth = it[Users.dateOfBirth]
            )
        }
    }

    fun getUserById(id: Int): User? = transaction {
          Users.select { Users.id eq id }
              .map { User(it[Users.id].value, it[Users.name], it[Users.dateOfBirth]) }
              .singleOrNull()
    }
...
```

Transaction is referencing a singleton with a Db instance

```
fun Transaction.getAllUsers(): List<User> =
    Users.selectAll().map {
        User(
            id = it[Users.id].value,
            name = it[Users.name],
            dateOfBirth = it[Users.dateOfBirth]
        )
    }


fun Transaction.getUserById(id: Int): User? =
    Users.select { Users.id eq id }
        .map { User(it[Users.id].value, it[Users.name], it[Users.dateOfBirth]) }
        .singleOrNull()
...
```

Transaction is now a receiver parameter of the stand alone functions

```
class UserController() {


    fun getAllUsersPage(): HtmlContent {
        val users = getAllUsers()
        return HtmlContent(HttpStatusCode.OK, usersPage(users))
    }

    fun getUserPage(id: Int?): HtmlContent {
        if (id == null) {
            return HtmlContent(HttpStatusCode.BadRequest,
                errorPage("Invalid ID format"))
        }

        val user = getUserById(id)
        if (user != null) {
            return HtmlContent(HttpStatusCode.OK, userPage(user))
        } else {
            return HtmlContent(HttpStatusCode.NotFound,
                errorPage("User not found"))
        }
    }

}
```

No data fields, we can get rid of Controller as well

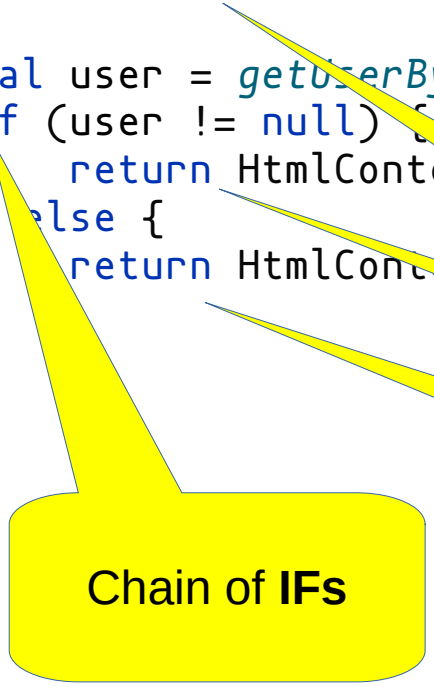
```
fun main() {
    initDatabase()


    embeddedServer(Netty, port = 8080) {
        routing {
            staticResources("/static", "static")

            get("/") {
                call.respond(HtmlContent(HttpStatusCode.OK, indexHtml()))
            }

            get("/users") {
                call.respond(
                    transaction {getAllUsersPage() }
                )
            }

            get("/user/{id}") {
                val id = call.parameters["id"]?.toIntOrNull()
                call.respond(
                    transaction {getUserPage(id) }
                )
            }
        }
```

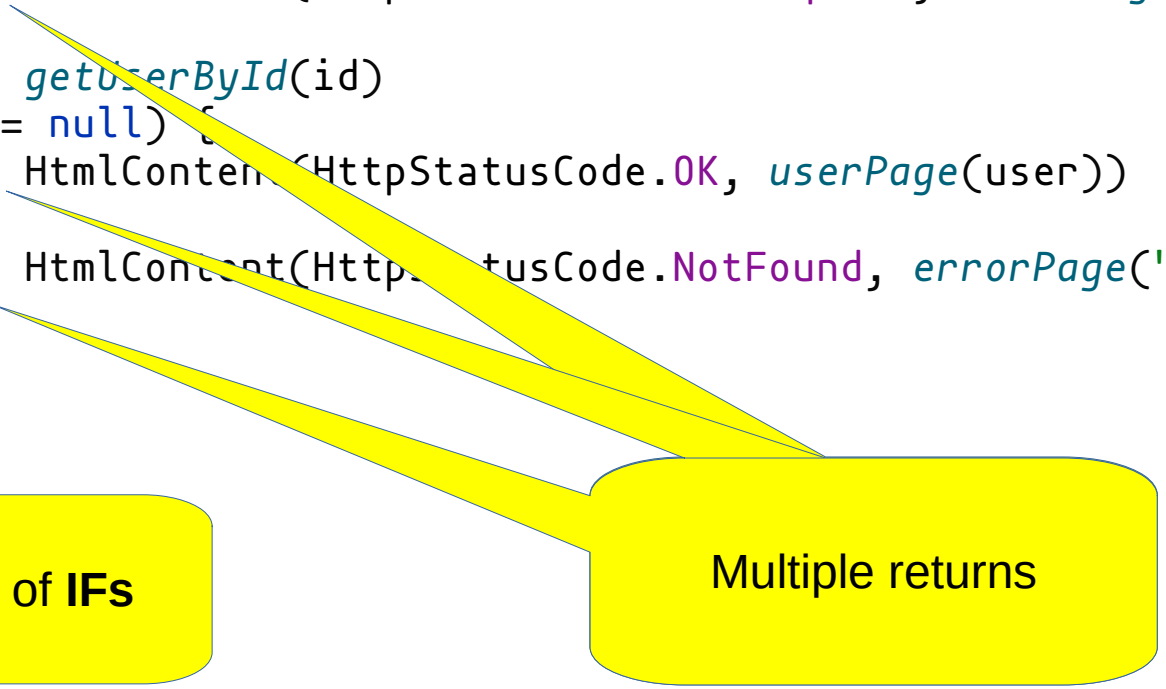
Controller and Service are gone! yeah!

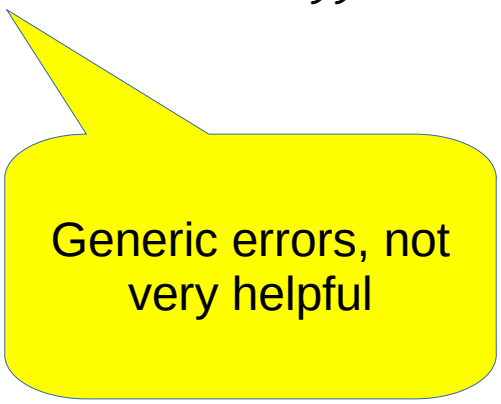
transaction blocks are now on main :(

Handling Errors

```
fun Transaction.getUserPage(id: Int?): HtmlContent {
    if (id == null) {
        return HtmlContent(HttpStatusCode.BadRequest, errorPage("Invalid ID format"))
    }
    val user = getUserById(id)
    if (user != null) {
        return HtmlConten (HttpStatusCode.OK, userPage(user))
    } else {
        return HtmlCont nt(Http tusCode.NotFound, errorPage("User not found"))
    }
}
```







```kotlin
sealed class Result<out T>

data class Success<T>(val value: T): Result<T>()

data class Failure(val error: Error): Result<Nothing>()



fun <T> T.asSuccess(): Result<T> = Success(this)
fun <E: Error> E.asFailure(): Result<Nothing> = Failure(this)


sealed interface Error{
    val msg: String
}

data class RequestError(override val msg: String, val request: HttpMessage): Error

data class DbError(override val msg: String, val exception: Exception? = null): Error

data class ResponseError(override val msg: String,
                         val statusCode: HttpStatusCode,
                         val cause: Error? = null): Error
```

Good

Bad
(Nothing cannot be instantiated)

Convenient constructors

Detailed context information for debug

Can keep the original error

```kotlin
fun Transaction.getAllUsers(): List<User> =
    Users.selectAll()
        .map {
            User(
                id = it[Users.id].value,
                name = it[Users.name],
                dateOfBirth = it[Users.dateOfBirth]
            )
        }



fun Transaction.getUserById(id: Int): User? =
    Users.select { Users.id eq id }
        .map {
            User(
                id = it[Users.id].value,
                name = it[Users.name],
                dateOfBirth = it[Users.dateOfBirth]
            )
        }.singleOrNull()
```

```kotlin
fun Transaction.getAllUsers(): Result<List<User>> =
    try {
        Users.selectAll().map {
            User(
                id = it[Users.id].value,
                name = it[Users.name],
                dateOfBirth = it[Users.dateOfBirth]
            )
        }.asSuccess()
    } catch (e: Exception) {
        DbError("Error loading all users", e).asFailure()
    }

fun Transaction.getUserById(id: Int): Result<User> =
    try {
        Users.select { Users.id eq id }
            .map {
                User(
                    id = it[Users.id].value,
                    name = it[Users.name],
                    dateOfBirth = it[Users.dateOfBirth]
                )
            }
            .single().asSuccess()
    } catch (e: Exception) {
```

Returns a Result

Success case

Keep the exception and add a context

```
fun Transaction.getUserPage(id: Int?): HtmlContent {

    if (id == null) {
        return HtmlContent(HttpStatusCode.BadRequest, errorPage("Invalid ID format"))
    }

    val userRes = getUserById(id)

    if (user != null) {
        return HtmlContent(HttpStatusCode.OK, userPage(user))
    } else {
        return HtmlContent(HttpStatusCode.NotFound, errorPage("User not found"))
    }
}
```

```
fun Transaction.getUserPage(id: Int?): HtmlContent =

    if (id == null) {
        ResponseError("Invalid ID format", HttpStatusCode.BadRequest).asFailure()
    } else {

        val userRes = getUserById(id)

        when (userRes) {
            is Success -> HtmlCont(HttpStatusCode.OK,
                                   serPage(userRes.value)).asSuccess()

            is Failure -> ResponseError( er not found",
                                HttpStatusC  NotFound, userRes.error).asFailure()

        }
    }.orThrow()
```

Return a Result

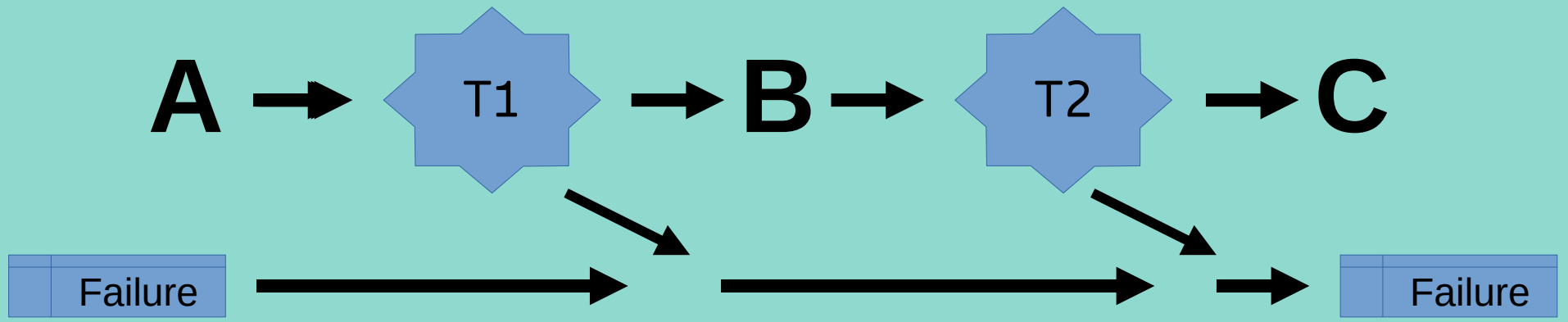
Still the IF

Also a When!

Throw exception in case of failure.

The Bowling Lane Transformer

The Bowling Lane Transformer



```
sealed class Result<out T> {

    fun <U> transform(f: (T) -> U): Result<U> =

        when(this){
            is Success -> Success(f(value))
            is Failure -> this
        }
}

data class Success<T>(val value: T): Result<T>()

data class Failure(val error: Error): Result<Nothing>()
```

Return a new type of Result

Apply the function to the value

Do nothing (a failure can represent any Result)

```
fun Transaction.getUserPage(id: Int?): HtmlContent =
    id.failIfNull(ResponseError("Invalid ID format", BadRequest))
        .transform { getUserById(it).orThrow() }
        .transform { HtmlContent(OK, userPage(it)) }
        .recover{ htmlForError(it) }
```

Fail if null and tranfomations

But still here

No more throw here

No more Result

```
fun recover(f: (Error) -> T): T =
    when(this){
        is Success -> value
        is Failure -> f(err
    }
```

Success value or recover the error

```
fun htmlForError(error: Error): HtmlContent =
    when(error){
        is ResponseError -> HtmlContent(error.statusCode, errorPage(error.msg))
        else -> HtmlContent(InternalServerError, errorPage(error.msg))
    }
```

```
sealed class Result<out T> {

...

 fun <U> bind(f: (T) -> Result<U>): Result<U> =
    when (this) {
        is Success -> f(value)
        is Failure -> this
    }
}
```

Bind two a Result with the Result from another function

If success we evaluate the function

otherwise we continue with failure

```
fun Transaction.getUserPage(id: Int?): HtmlContent =
    id.failIfNull(ResponseError("Invalid ID format", BadRequest))
        .bind { getUserById(it) }
        .transform { htmlUserPage(it) }
        .recover { htmlForError(it) }
```

No more throw, all uniform

Small function to make it more uniform

Implicit lambda parameter

```
fun htmlUserPage(it: User) = HtmlContent(OK, userPage(it))
```

```
fun getUserPage(id: Int?): HtmlContent {
        if (id == null) {
            return HtmlContent(HttpStatusCode.BadRequest,
                userView.errorPage("Invalid ID format"))
        }

        val user = userService.getUserById(id)
        if (user != null) {
            return HtmlContent(HttpStatusCode.OK, userView.userPage(user))
        } else {
            return HtmlContent(HttpStatusCode.NotFound,
                userView.errorPage("User not found"))
        }
    }
```

There is still this...

```
fun Transaction.getUserPage(id: Int?): HtmlContent =
    id.failIfNull(ResponseError("Invalid ID format", BadRequest))
        .bind(::getUserById)
        .transform(::htmlUserPage)
        .recover(::htmlForError)
```

Function references are easier to read than lambdas

Partial Application
(Functional DI)

$$(A, B) \rightarrow C \quad \equiv \quad A \rightarrow (B \rightarrow C)$$

```
fun <A, B, C> partialAppl(f: (A, B) -> C): (A) -> (B) -> C =
    { a -> { b -> f(a, b) } }
```

```
val db = initDatabase()
...
val userPageFromDb = inTransaction(db, Transaction::getUserPage)
val allUsersPageFromDb = inTransaction(db, Transaction::getAllUsersPage)

embeddedServer(Netty, port = 8080) {
    routing {
...
        get("/users") {
            call.respond(allUsersPageFromDb)
        }

        get("/user/{id}") {
            val id = call.parameters["id"]?.toIntOrNull()
            call.respond(userPageFromDb(id) )
        }
    }.start(wait = true)
}


fun <T, R> inTransaction(db: Database, f: (Transaction).(T) -> R): (T) -> R =
    { x: T -> transaction(db) { f(x) } }
```

Esplicit db handling

Partial application of Transaction

Use it as a Pure Function

```
data class TransactionRunner<T>(val inTxBlock: Transaction.() -> T) {

    fun <U> transform(f: (T) -> U): TransactionRunner<U> =
        TransactionRunner { f(inTxBlock(this)) }

    fun runOnDb(db: Database) = transaction(db) { inTxBlock }

}


fun getUserPage(id: Int?): TransactionRunner<HtmlContent> =
    id.failIfNull(ResponseError("Invalid ID format", BadRequest))
        .bind(::getUserById)
        .transform {x-> x.transform {  htmlUserPage(it)}
        .recover { htmlForError(it).inTx() }



get("/user/{id}") {
    val id = call.parameters["id"]?.toIntOrNull()
    val tx = TransactionRunner {getUserPage(id)}
    call.respond(tx.runOnDb(db))
}
```

We start with a function that return something from a Tx

Not easy to combine a Result with a TxRunner

Then we run everything on db at the end

Chase the Simplicity

Questions

Uberto Barbini
@ramtop
https://medium.com/@ramtop
https://pragprog.com/titles/uboop/from-objects-to-functions/

all the code of this talk: https://github.com/uberto/miniktorOOP