F♯ Training

Domain workflows

2025 September

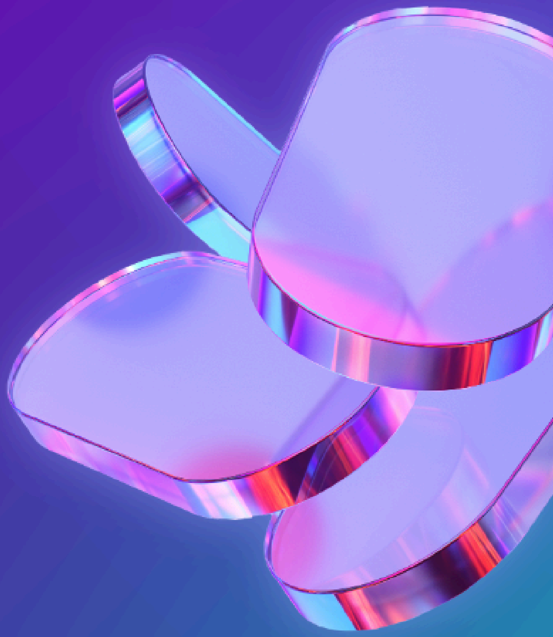


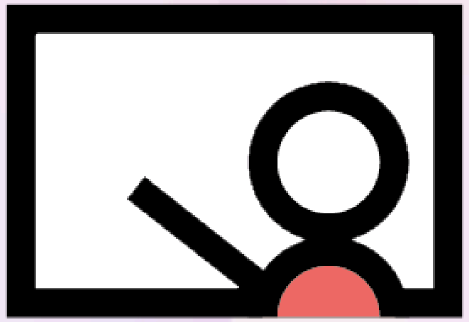
Table of contents

➔ Dependency injection: de facto state of the art in C#

➔ Dependency interpretation: `program` CE V1

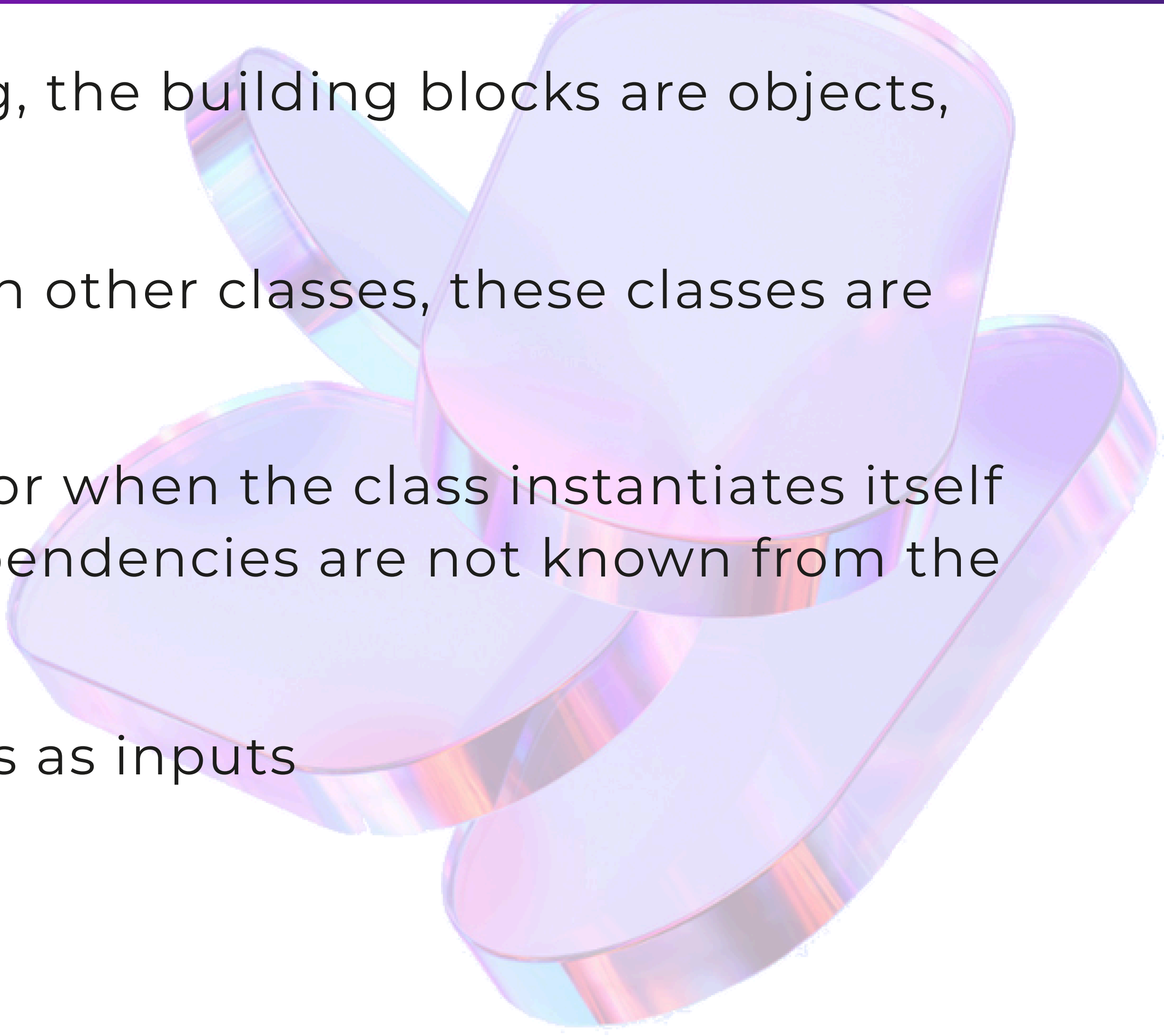
➔ Free monad: program CE V2

➔ Algebraic effects: program CE v3



1. Dependency Injection (DI)

Dependency injection: introduction



Context: In object-oriented programming, the building blocks are objects, more specifically **classes** in C#.

Definition: When a class collaborates with other classes, these classes are called **dependencies**.

Problem: When dependencies are static or when the class instantiates itself its dependencies, it's a black box: the dependencies are not known from the caller. This class is difficult to unit test.

Solution: the class takes its dependencies as inputs

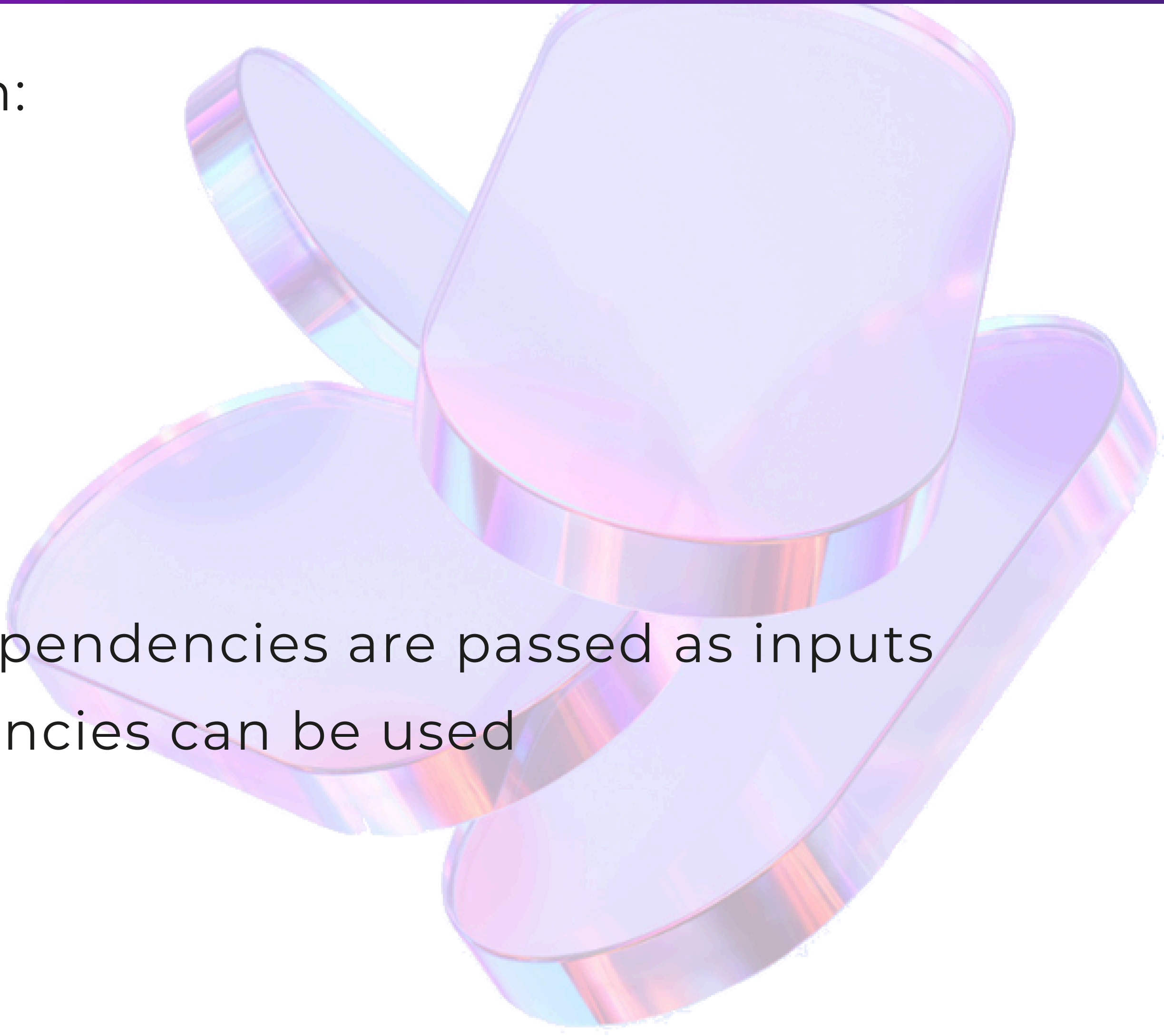
Dependency injection types

There are 3 types of dependency injection:

- Constructor injection
- Method injection
- Property injection

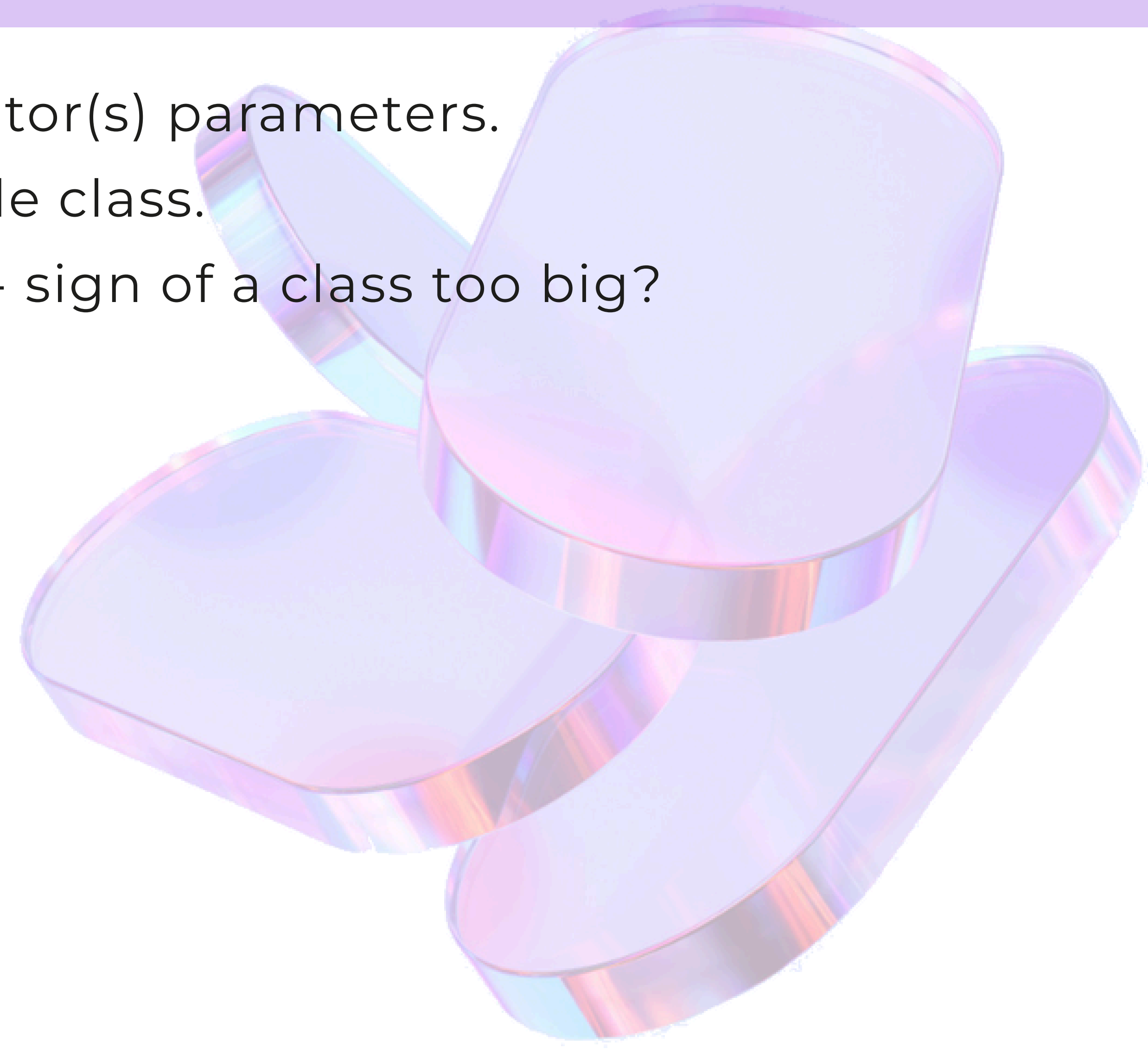
Each dependency injection type:

- depends on the **location** where the dependencies are passed as inputs
- impacts the **scope** where the dependencies can be used



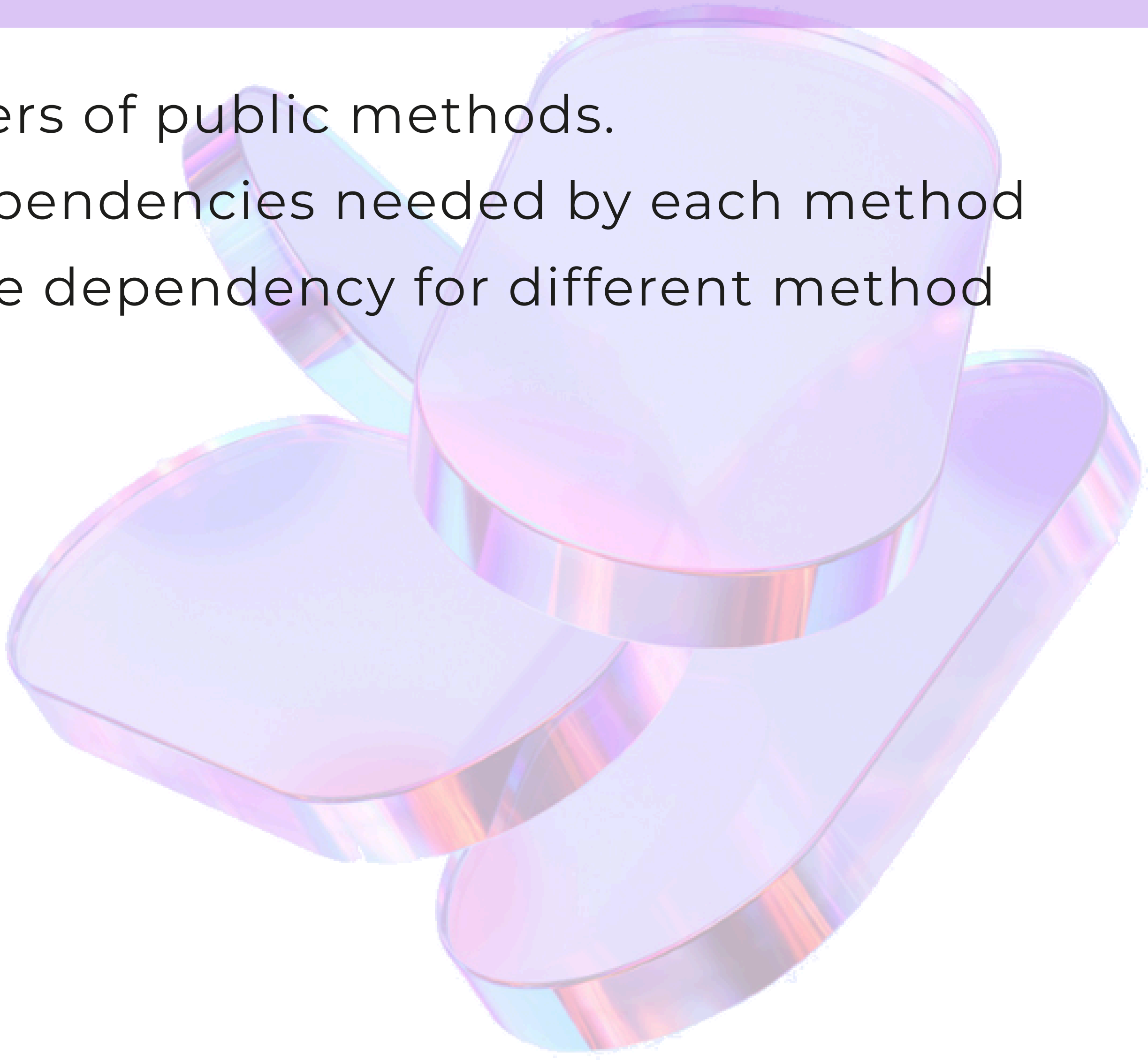
Constructor injection

- Dependencies are defined as constructor(s) parameters.
- Benefit: can be used through the whole class.
- Limit: too many dependencies visible - sign of a class too big?



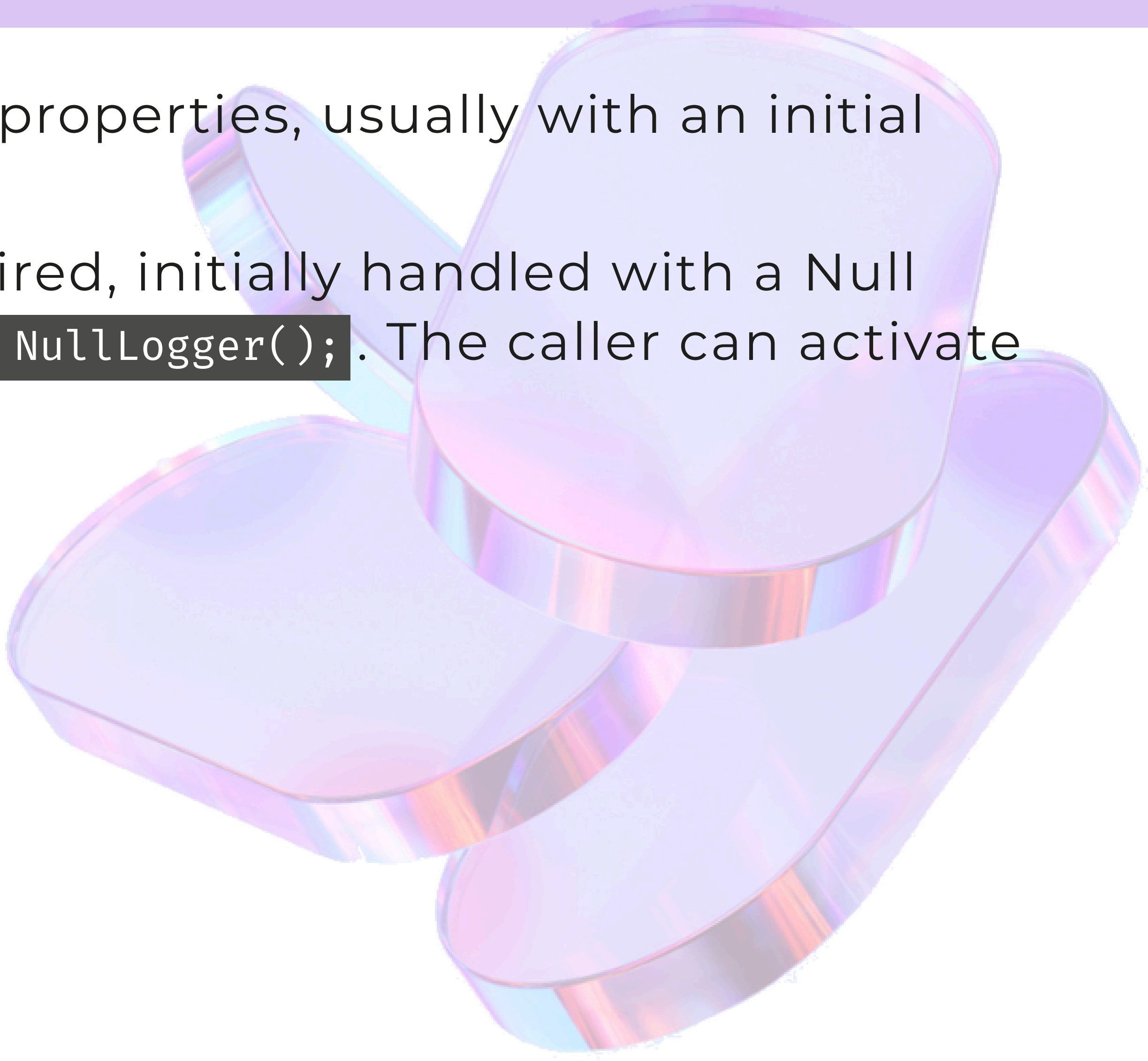
Method injection

- Dependencies are defined as parameters of public methods.
- Benefit: more explicit: we know the dependencies needed by each method
- Limit: boilerplate in repeating the same dependency for different method



Property injection

- Dependencies are defined as mutable properties, usually with an initial value to never be null
- Benefit: model dependencies not required, initially handled with a Null Object: `public ILogger { get; set; } = new NullLogger();`. The caller can activate features by passing real objects.
- Limit: mutability…

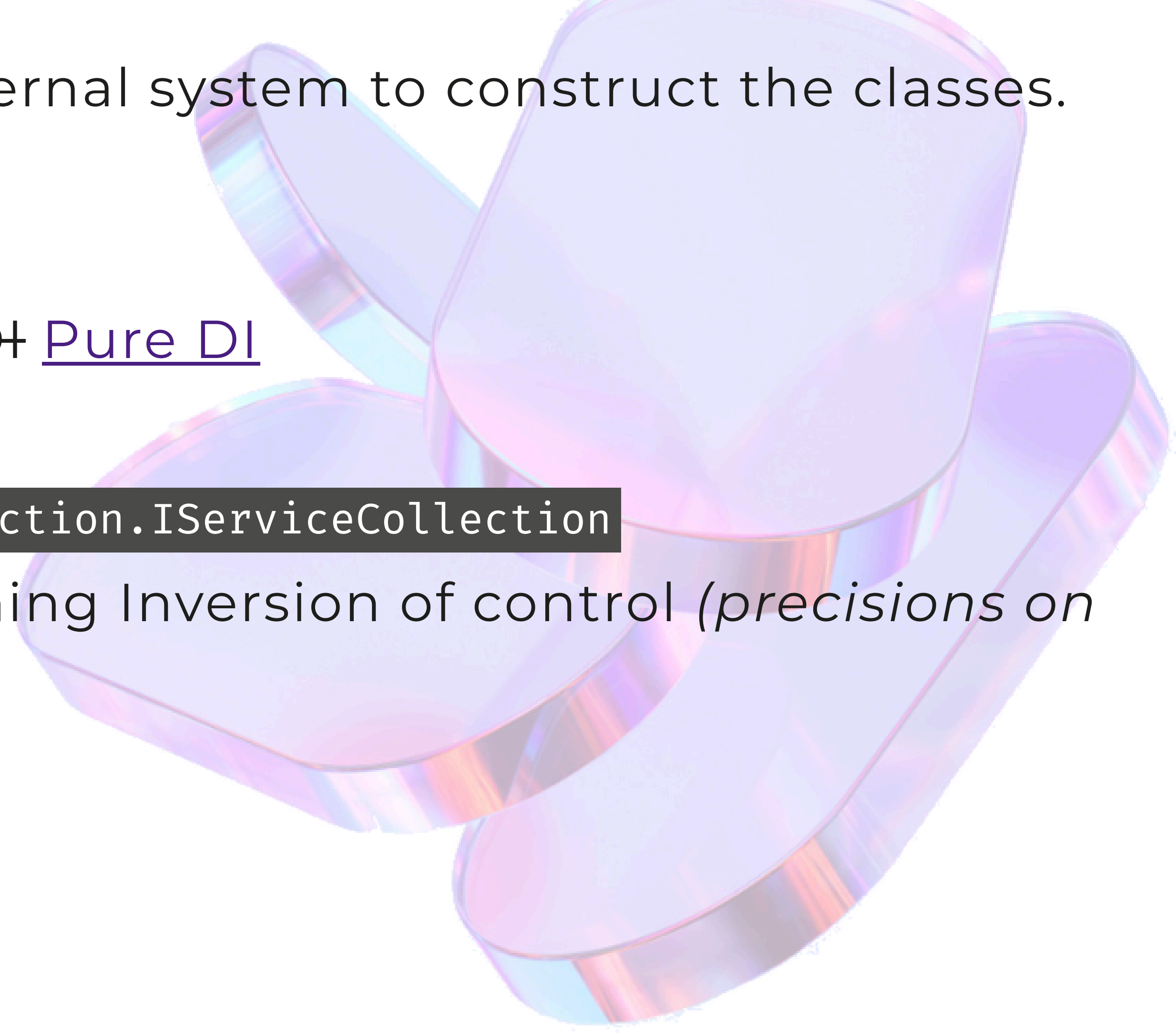


Dependency injection system

The dependency injection implies an external system to construct the classes.

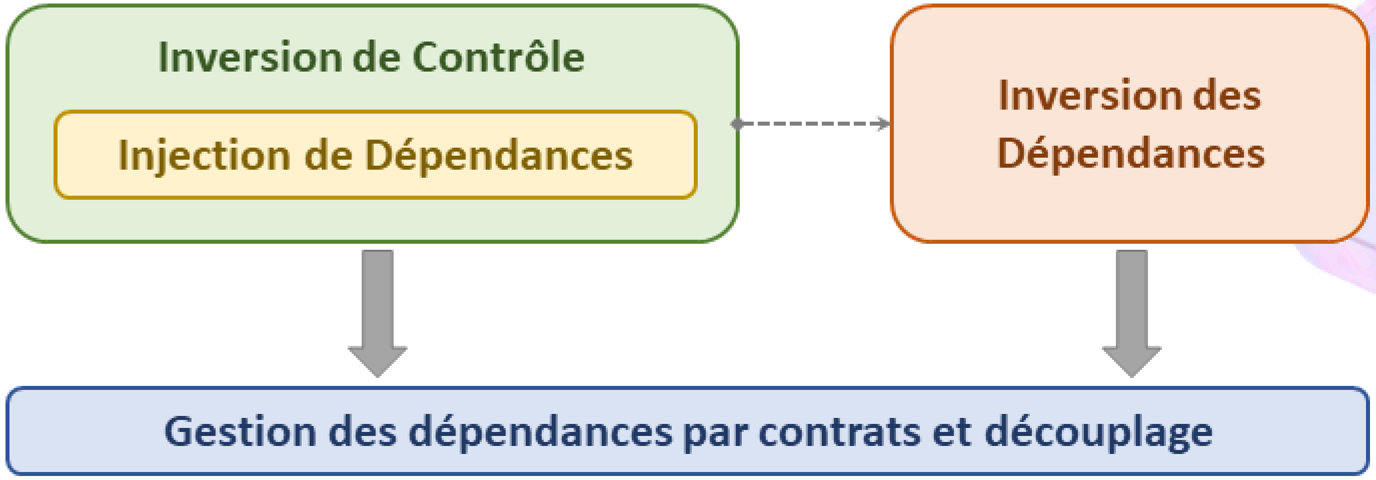
There are 2 kinds of DI system:

- By hand, in the app root: ~~Poor man's DI~~ Pure DI
- With a **DI container**
  - E.g. `Microsoft.Extensions.DependencyInjection.IServiceCollection`
  - Also called IOC container, IOC meaning Inversion of control *(precisions on the next slide)*



Inversion of control

- Inversion of the flow of control
- AKA the **Hollywood Principle**: “Don't call us, we'll call you”.
- *Dependency injection* is a type of *Inversion of control*.
- IOC ≠ Dependency inversion principle (DIP)


Inversion de Contrôle
Injection de Dépendances
Inversion des Dépendances
Gestion des dépendances par contrats et découplage

🔗 https://www.neosoft.fr/nos-publications/blog-tech/dependances-dip-et-ioc/

Service locator anti-pattern

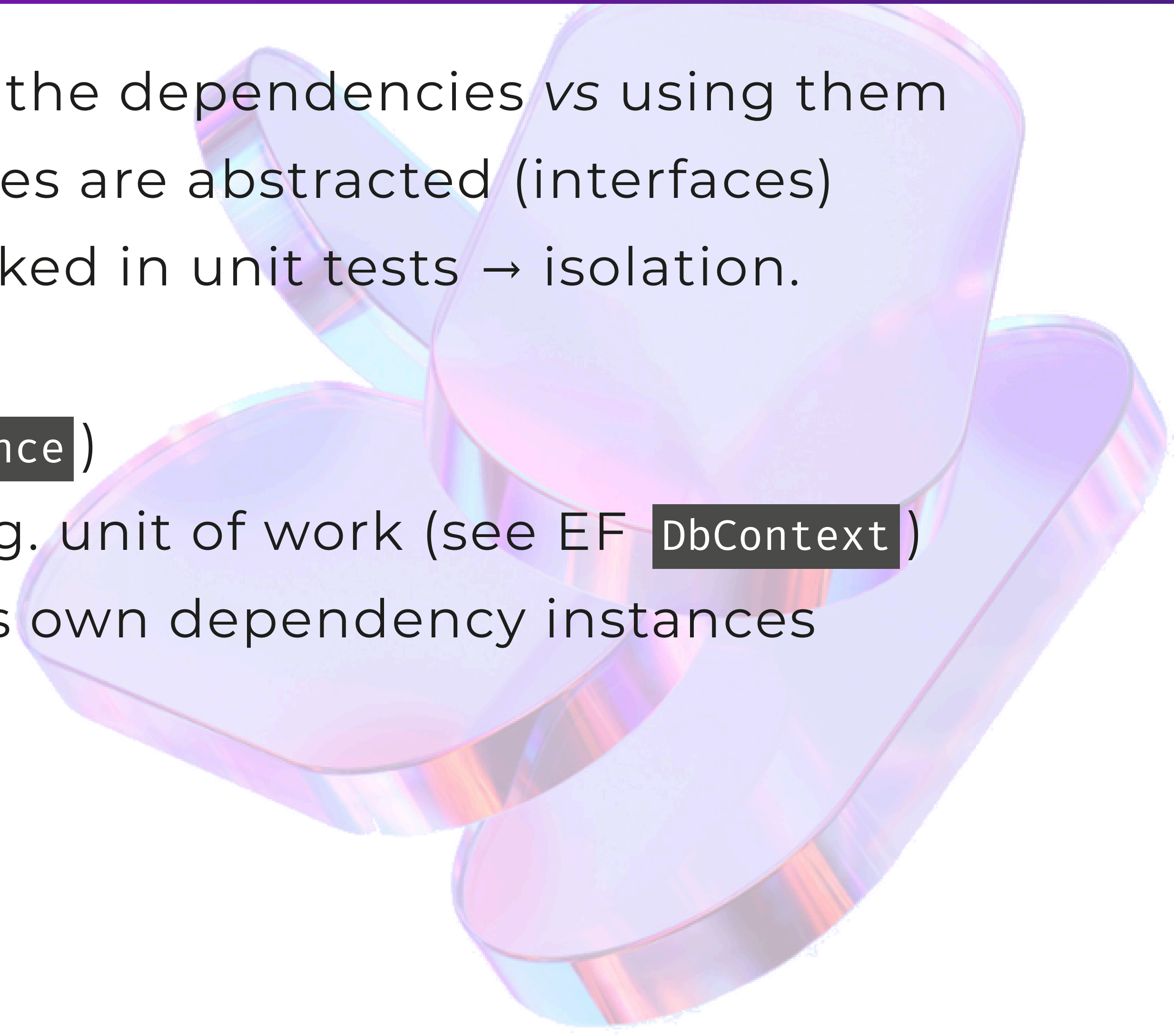
Service locator = central registry to obtain a service instance (any dependency registered)

⚠️ **Issue:** hides the class' dependencies

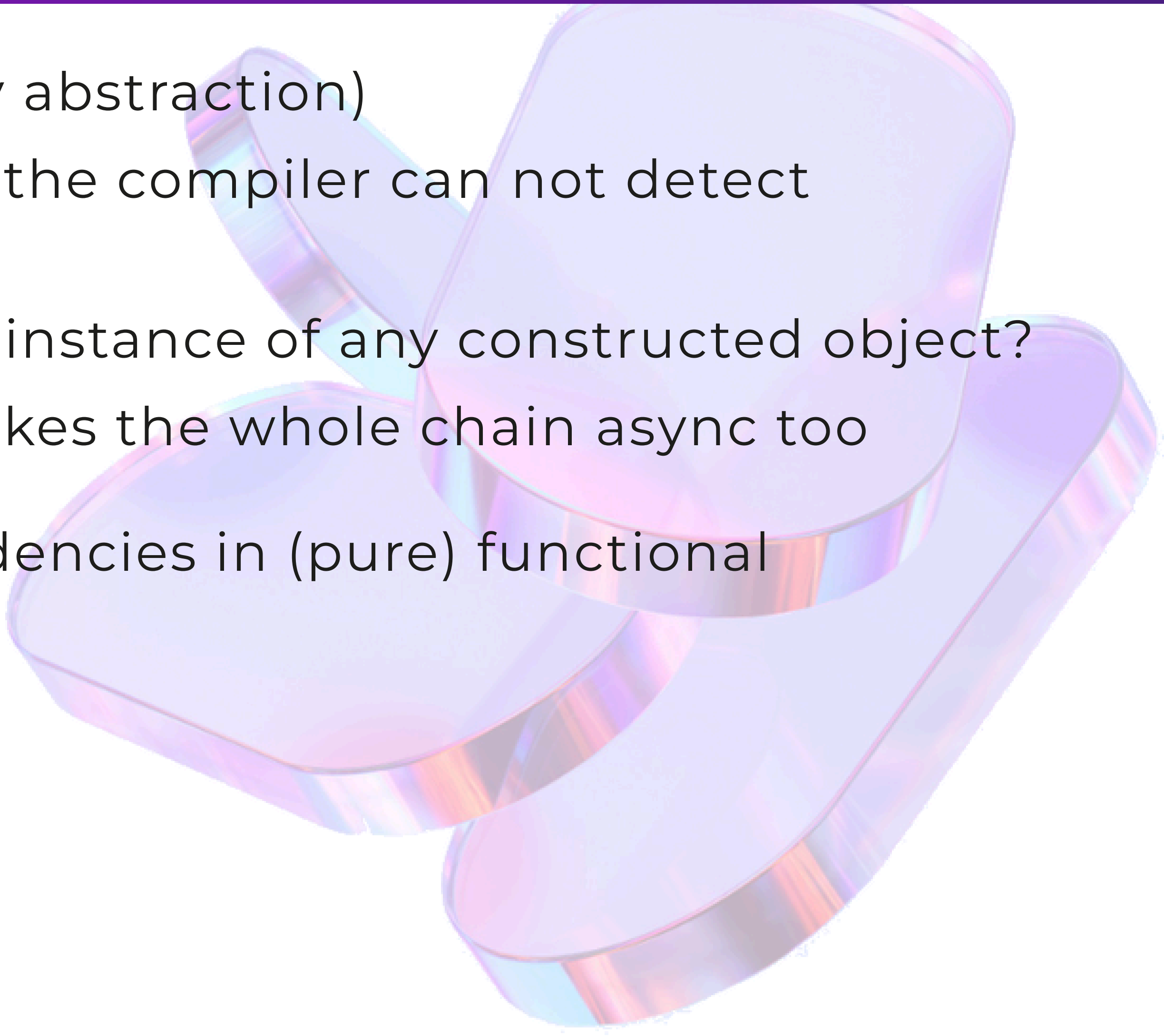
❌ DON'T inject `IServiceCollection`!
✅ DO inject explicitly each dependency

Dependency injection benefits

- **Separation of concerns:** constructing the dependencies *vs* using them
- **Loose coupling** when the dependencies are abstracted (interfaces)
- **Testability:** dependencies can be mocked in unit tests → isolation.
- **Dependency life cycle:** way to get
  - **Singleton**: not static (≠ `MyClass.Instance`)
  - **Scoped** (per request in ASP.NET): e.g. unit of work (see EF `DbContext`)
  - **Transient:** every class will receive its own dependency instances



Dependency injection limits



- Create **indirections** (like the use of any abstraction)
- Real dependencies known at **runtime**: the compiler can not detect dependency mis-configuration
- Can be **difficult to test**: can we get an instance of any constructed object?
- **Async leak**: any async dependency makes the whole chain async too

👉 There are other ways to handle dependencies in (pure) functional programming.

Builders and Factories

Intermediate way to handle dependencies:

- A builder or a factory is injected.
- The class controls WHAT dependencies to build and WHEN.

2. Dependency interpretation - program CE V1

Data dependencies

Given the following Domain layer dependencies from the Data layer:

```
[<Interface>]
type IChannelClient =
    abstract GetChannelDescription: channelId: int → Async<string option>

[<Interface>]
type IMailSender =
    abstract Send: MailEntities.Mail → Async<unit>

[<Interface>]
type IMappingClient =
    abstract NotifyLinkEvent: channelId: int * hotelId: int * LinkStatus → Async<unit>
    abstract GetMappingActivation: channelId: int * hotelId: int → Async<MappingEntities.MappingActivationDto option>
```

Program type

Dependencies abstracted as instructions into the following union type:

```fsharp
type Program<'a> =
    | Stop of 'a

    // Channel
    | GetChannelDescription of ChannelId * (ChannelDescription option -> Program<'a>)

    // Mail
    | SendMail of MailType * (Result<unit, Error> -> Program<'a>)

    // Mapping
    | NotifyLinkEvent of (ChannelId * CrsHotelId * LinkStatus) * (Result<unit, Error> -> Program<'a>)
    | GetMappingActivation of (ChannelId * CrsHotelId) * (MappingActivation option -> Program<'a>)
```

Program instructions

Instructions are defined as union cases following a common pattern:

```
| Instruction           of Input                                   * (Output                   -> Program<'a>)

// Queries
| GetChannelDescription of ChannelId                               * (ChannelDescription option -> Program<'a>)
| GetMappingActivation  of (ChannelId * CrsHotelId)                * (MappingActivation  option -> Program<'a>)

// Commands
| NotifyLinkEvent       of (ChannelId * CrsHotelId * LinkStatus) * (Result<unit, Error>        -> Program<'a>)
| SendMail              of MailType                                * (Result<unit, Error>        -> Program<'a>)
```

Command/Query Separation Convention:

- **Commands** return a `Result<unit, _>` (no output data)
- **Queries** return an `option`: `None` ≃ HTTP Error 404 NotFound

Program type: how it works

```
type Program<'a> = // <────────Recursion───────────┐
    │ Stop of 'a    //                         ┌─────┴─────┐
    │ Instruction1 of Input1 * (Output1 → Program<'a>)
    │ Instruction2 of Input2 * (Output2 → Program<'a>)
    │ ...                  // └───────Continuation──────┘
```

`Program` type is **recursive**: it's a **list of instructions**.

`Stop` is the program **terminal case**, containing its **returned value**.

Instruction second element—`Output → Program<'a>`—is a **continuation**:
– Processes the instruction output.
– Returns the rest of the program.
– Contains the program logic.

Program examples (1)

```fsharp
// 1. Program returning a value—10 here
let p : Program<int> = Stop 10

// 2. Program returning the channel description by its id
let getChannelDescription (channelId: ChannelId) : Program<ChannelDescription option> =
    GetChannelDescription (channelId, Stop)
                                  //   └──┘ Continuation
```

💡 **Key points:**

- `Stop` matches the continuation signature— `'a -> Program<'a>`.
- The program returned value is passed to `Stop`, hence the returned type: `Program<int>` and `Program<ChannelDescription option>`.

Program examples (2)

```
// 3. Program returning the channel name by its id
let getChannelName channelId : Program<ChannelName option> =
    GetChannelDescription (channelId, fun channelOption → Stop (channelOption ▷ Option.map _.Name))
```

Can we use the previous `getChannelDescription`?

Yes, with a functorial `map` function:

```
let getChannelName channelId : Program<ChannelName option> =
    getChannelDescription channelId
    ▷ Program.map Option.map _.Name
```

Program `map` function

`map f program` is based on the `Program` type pattern matching.

- Each instruction has a continuation function—called `next` that returns the rest of the program… to map in turn—`next >> map f`
- Until reaching the terminal `Stop` containing the program returned value `x` that we can map with `f`—`f x`

```
module Program =
    let rec map (f: 't → 'u) (program: Program<'t>) : Program<'u> =
        match program with
        | GetChannelDescription(x, next) → GetChannelDescription(x, next >> map f)
        | GetMappingActivation(x, next) → GetMappingActivation(x, next >> map f)
        | NotifyLinkEvent(x, next) → NotifyLinkEvent(x, next >> map f)
        | SendMail(x, next) → SendMail(x, next >> map f)
        | Stop x → Stop(f x)
```

Program examples (3)

Can we improve how to use the previous `getChannelDescription`?

Yes, with a `program` CE:

```
let getChannelName channelId : Program<ChannelName option> =
    program {
        let! channelDescription = getChannelDescription channelId
        return channelDescription ▷ Option.map _.Name
    }
```

Program computation expression

The `program` is a monadic computation expression.
→ The bear minimum methods for its builder are:

- `Return` elevates a value up to a `Program`: it's `Stop`.
- `Bind` is delegating to `Program.bind` function 📍

```fsharp
type ProgramBuilder() =
    member _.Return(x) = Stop x
    member _.Bind(px, f) = Program.bind f px

let program = ProgramBuilder()
```

Program `bind` function

The monadic `bind f program` is very similar to `map`:

- We bind the program returned by instruction continuation.
- The `Stop x` case is matched to the program returned by `f x`.

```fsharp
module Program =
    let rec bind (f: 't → Program<'u>) (program: Program<'t>) : Program<'u> =
        match program with
        | GetChannelDescription(x, next) → GetChannelDescription(x, next >> bind f)
        | GetMappingActivation(x, next) → GetMappingActivation(x, next >> bind f)
        | NotifyLinkEvent(x, next) → NotifyLinkEvent(x, next >> bind f)
        | SendMail(x, next) → SendMail(x, next >> bind f)
        | Stop x → f x
```

Program instruction helpers

The previous example shows the usefulness of helpers to call an instruction from the program body.

It's just boilerplate code: `let instruction args = Instruction(args, Stop)`

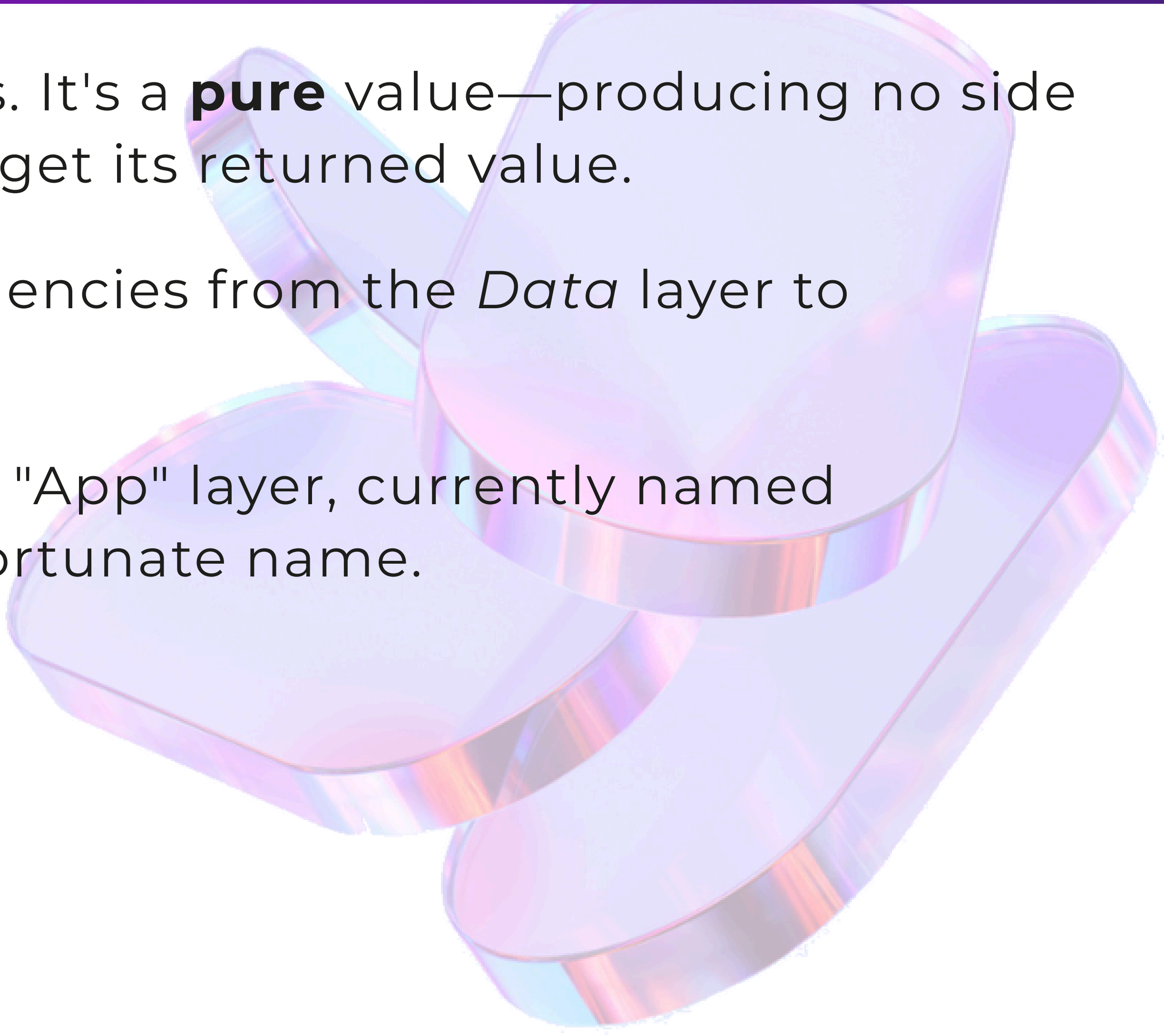
```
// Queries
let getChannelDescription args = GetChannelDescription(args, Stop)
let getMappingActivation args = GetMappingActivation(args, Stop)

// Commands
let notifyLinkEvent args = NotifyLinkEvent(args, Stop)
let sendMail args = SendMail(args, Stop)
```

Program interpreter (1)

A program is a chained list of instructions. It's a **pure** value—producing no side effects— that needs to be **interpreted** to get its returned value.

The **interpreter** collaborates with dependencies from the *Data* layer to execute the instructions.

It lives not in the *Domain* layer but in the "App" layer, currently named *Infrastructure* in the SCM but it is an unfortunate name.



Program interpreter (2)

The interpreter is a function:

- **recursive:** as the program
- **asynchronous:** as the dependencies

```fsharp
let rec interpretProgram dependencies (prog: Program<'a>) : Async<'a> =
    async {
        match prog with
        | Stop x -> return x
        | GetChannelDescription(x, next) ->
            let! (res: ChannelDescription option) = dependencies.ChannelClient.getChannelDescription x
            return! interpretProgram dependencies (next res)
        // ...
        | SendMail(x, next) ->
            let! (res: Result<unit, Error>) = dependencies.MailSender.sendMail x
            return! interpretProgram dependencies (next res)
    }
```

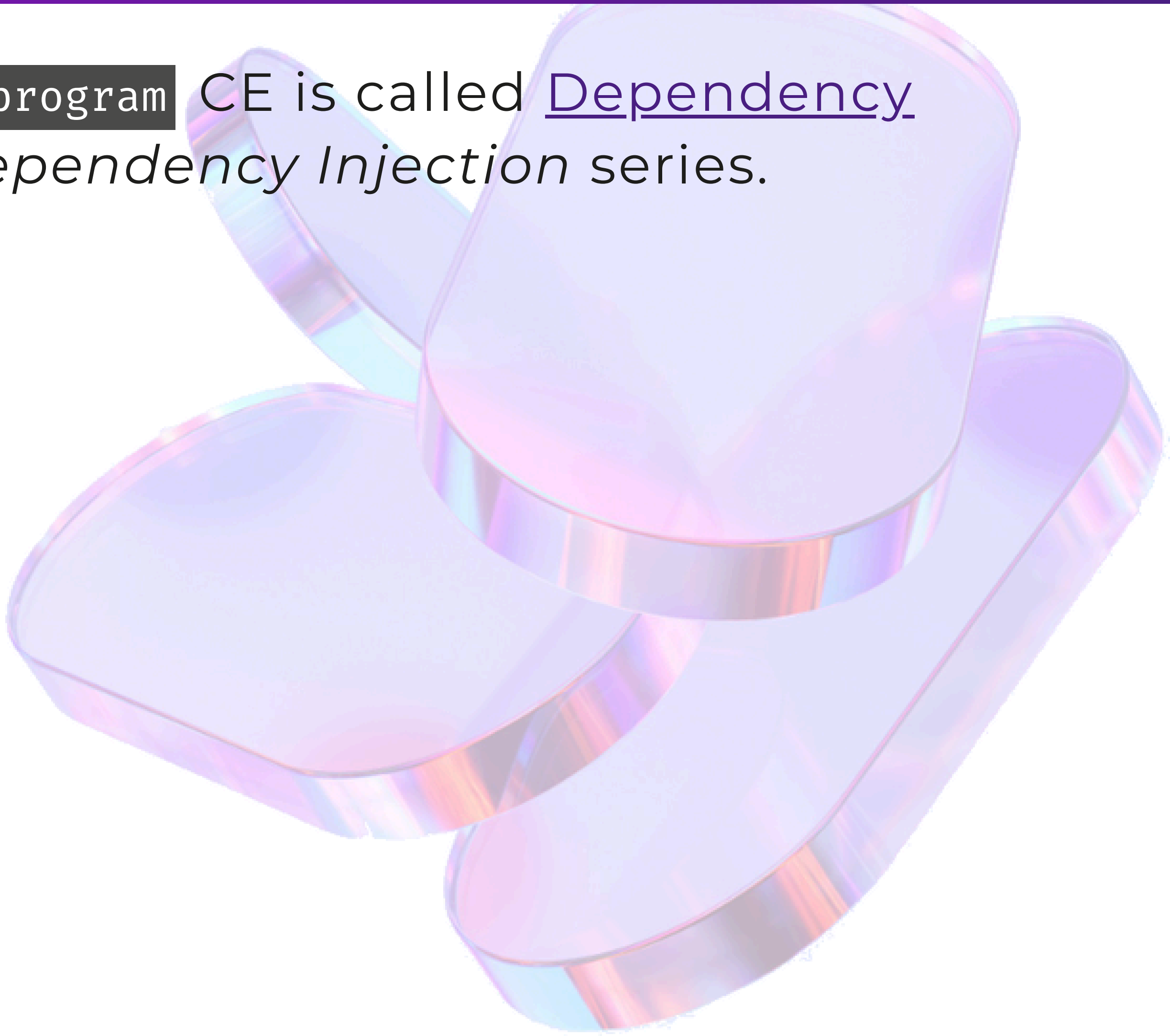
Program CE origin

🔗 [Fighting complexity in software development > Design application > Business logic](), by Roman Liman

```fsharp
let processPayment (currentDate: DateTimeOffset, payment) =
    program {
        let! cmd = validateProcessPaymentCommand payment ▷ expectValidationError
        let! card = tryGetCard cmd.CardNumber
        let today = currentDate.Date ▷ DateTimeOffset
        let tomorrow = currentDate.Date.AddDays 1. ▷ DateTimeOffset
        let! operations = getBalanceOperations (cmd.CardNumber, today, tomorrow)
        let spentToday = BalanceOperation.spentAtDate currentDate cmd.CardNumber operations
        let! (card, op) =
            CardActions.processPayment currentDate spentToday card cmd.PaymentAmount
            ▷ expectOperationNotAllowedError
        do! saveBalanceOperation op ▷ expectDataRelatedErrorProgram
        do! replaceCard card ▷ expectDataRelatedErrorProgram
        return card ▷ toCardInfoModel ▷ Ok
    }
```

Dependency interpretation

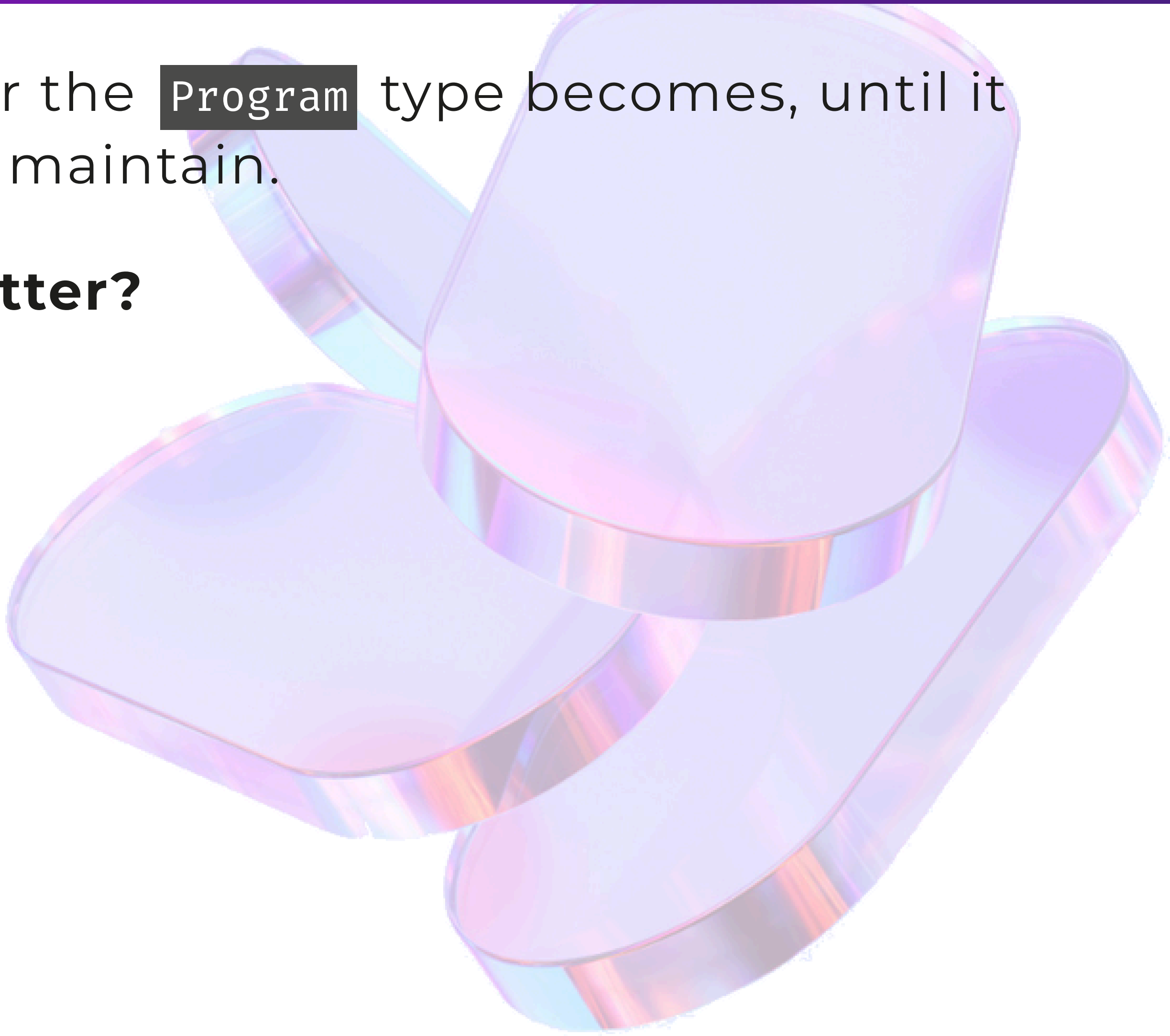
The pattern based on this version of the `program` CE is called Dependency Interpretation by Scott Wlaschin in his *Dependency Injection* series.



Program limits

The more instructions you add, the bigger the `Program` type becomes, until it gets hard to read and too complicated to maintain.

How to improve this pattern to scale better?



3. Free monad with program CE V2

Free monad

F# does not support a general definition of the free monad.

Its definition adapted for our `Program` type looks like that, where `Instruction` is a functor—hence its `map` function:

```fsharp
type Program<'a> =
    | Free of Instruction<Program<'a>>
    | Pure of 'a

module Program =
    let rec bind f = function
        | Free inst → inst ▷ Instruction.map (bind f) ▷ Free
        | Pure x → f x
```

`Pure` is our `Stop`. The `Free of Instruction` case gives us a way to separate instructions from the program and to group them by domain.

Separation of instructions

Folder structure:

```
🗃️ Domain/
├──📂 Program/
│  ├──📄 Instructions.fs
│  ├──📄 Program.fs
│  └──📄 Environments.fs
└──📂 Workflows/
   ├──📄 CategoryWorkflow.fs
   └── ...
```

Instruction types and map functions

```fsharp
type Mail<'a> =
    | SendMail of MailType * (Result<unit, Error> → 'a)

module Mail =
    let map f = function
        | SendMail(x, next) → SendMail(x, (next >> f))

type Partner<'a> =
    | GetChannelDescription of ChannelId * (ChannelDescription option → 'a)
    | GetMappingActivation of (ChannelId * CrsHotelId) * ((MappingActivation * LinkStatus) option → 'a)
    | NotifyLinkEvent of (ChannelId * CrsHotelId * LinkStatus) * (Result<unit, Error> → 'a)

module Partner =
    let map f = function
        | GetChannelDescription(x, next) → GetChannelDescription(x, (next >> f))
        | GetMappingActivation(x, next) → GetMappingActivation(x, (next >> f))
        | NotifyLinkEvent(x, next) → NotifyLinkEvent(x, (next >> f))
```

Program type

```
type Program<'a> =
    | Stop of 'a
    | Mail of Instructions.Mail<Program<'a>>
    | Partner of Instructions.Partner<Program<'a>>

module Program =
    let rec bind f = function
        | Stop x -> f x
        | Mail inst -> Mail(inst |> Instruction.Mail.map (bind f))
        | Partner inst -> Partner(inst |> Instruction.Partner.map (bind f))
```

Program instruction helpers

Defined in `Environments.fs` · Each returns a `Program<_>`

```fsharp
module Mail =
    let sendMail args =
        Instructions.Mail.SendMail(args, stop)
        ▷ Mail

module Partner =
    let getChannelDescription args =
        Instructions.Partner.GetChannelDescription(args, stop)
        ▷ Partner

    let getMappingActivation args =
        Instructions.Partner.GetMappingActivation(args, stop)
        ▷ Partner

    let notifyLinkEvent args =
        Instructions.Partner.NotifyLinkEvent(args, stop)
        ▷ Partner
```

Domain error

A workflow usually returns a `Program<Result<xxx, Error>>` where `Error` is an union type like:

```
type Error =
    | DataError of DataRelatedError
    | ValidationError of ValidationError
    | ...
```

`ValidationError` is usually the type used in Domain type smart constructors.

How to call a smart constructor within a `program`?

Validation error helpers

```
// Helpers
let expectValidationError result = Result.mapError ValidationError result

let createOrValidationError constructor input = constructor input ▷ expectValidationError

// Domain
type ChannelId = private ChannelId of int with
    member this.Value = match this with ChannelId i → i

    static member FromString(str: string) =
        match System.Int32.TryParse(str) with
        | true, value → ChannelId value ▷ Ok
        | false, _ → validationError "ChannelId" $"Should be positive. Value was %s{str}"

// Workflow
program {
    let! channelId = createOrValidationError ChannelId.FromString args.channelId
     ...
}
```

Query helpers

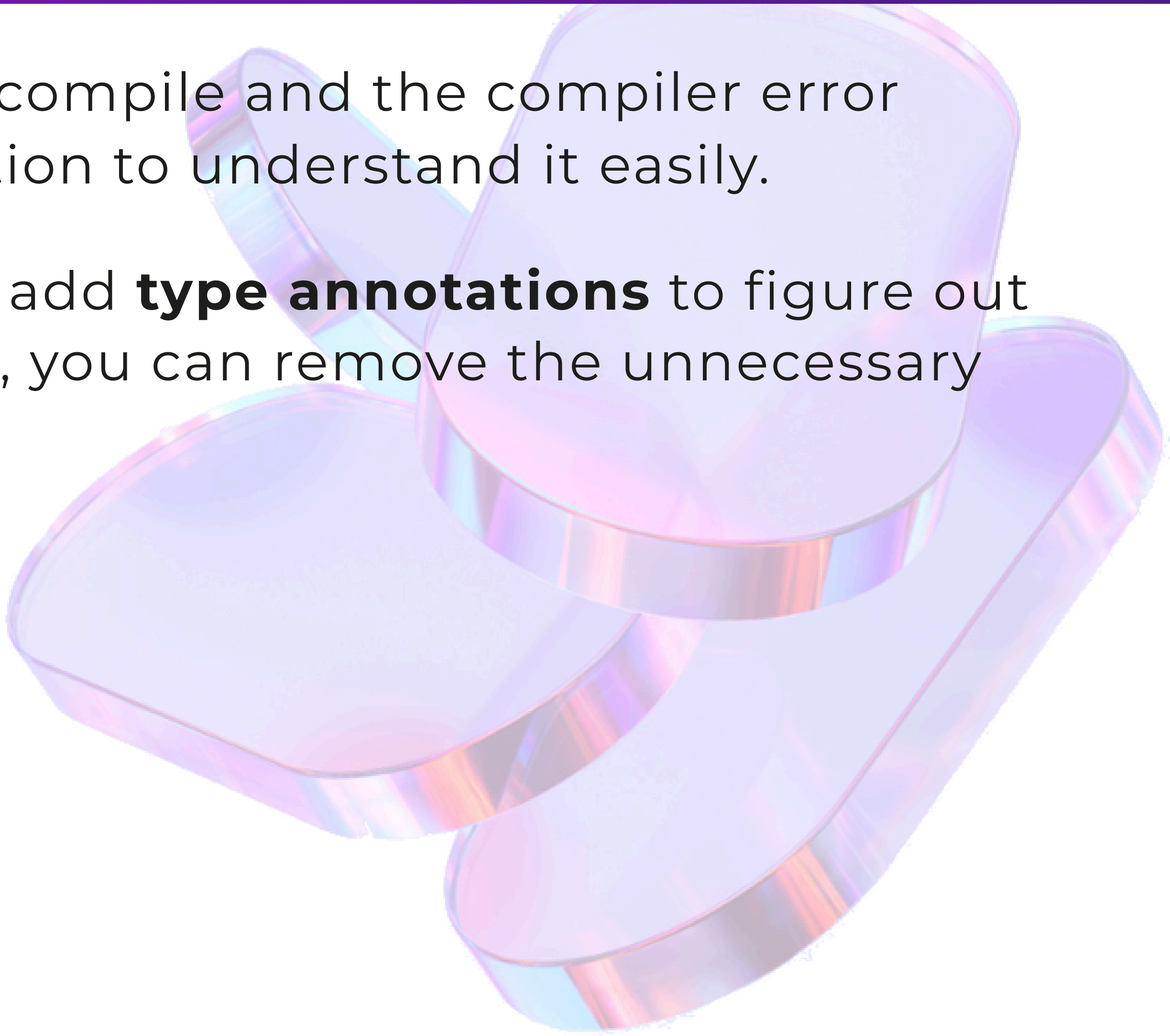
When the result of a query is required, how to handle it in a workflow?

```
// Helpers
let expectDataRelatedError result = Result.mapError DataError result

let inline noneToError id (a: 'a option) =
    let error = DataNotFound(id, $"%s{typeof<'a>.Name}")
    Result.ofOption error a

// Workflow
program {
    let! hotelContact = getHotelContact hotelId

    let! hotelContact = // ⚠️ shadowing
        hotelContact
        ▷ noneToError $"H%s{hotelId.Value}"
        ▷ expectDataRelatedError
    ...
}
```

Program writing tips

Without these helpers, the `program` won't compile and the compiler error message is not placed at the proper location to understand it easily.

💡 If your `program` does not compile, try to add **type annotations** to figure out where is the unexpected type. Once fixed, you can remove the unnecessary annotations.

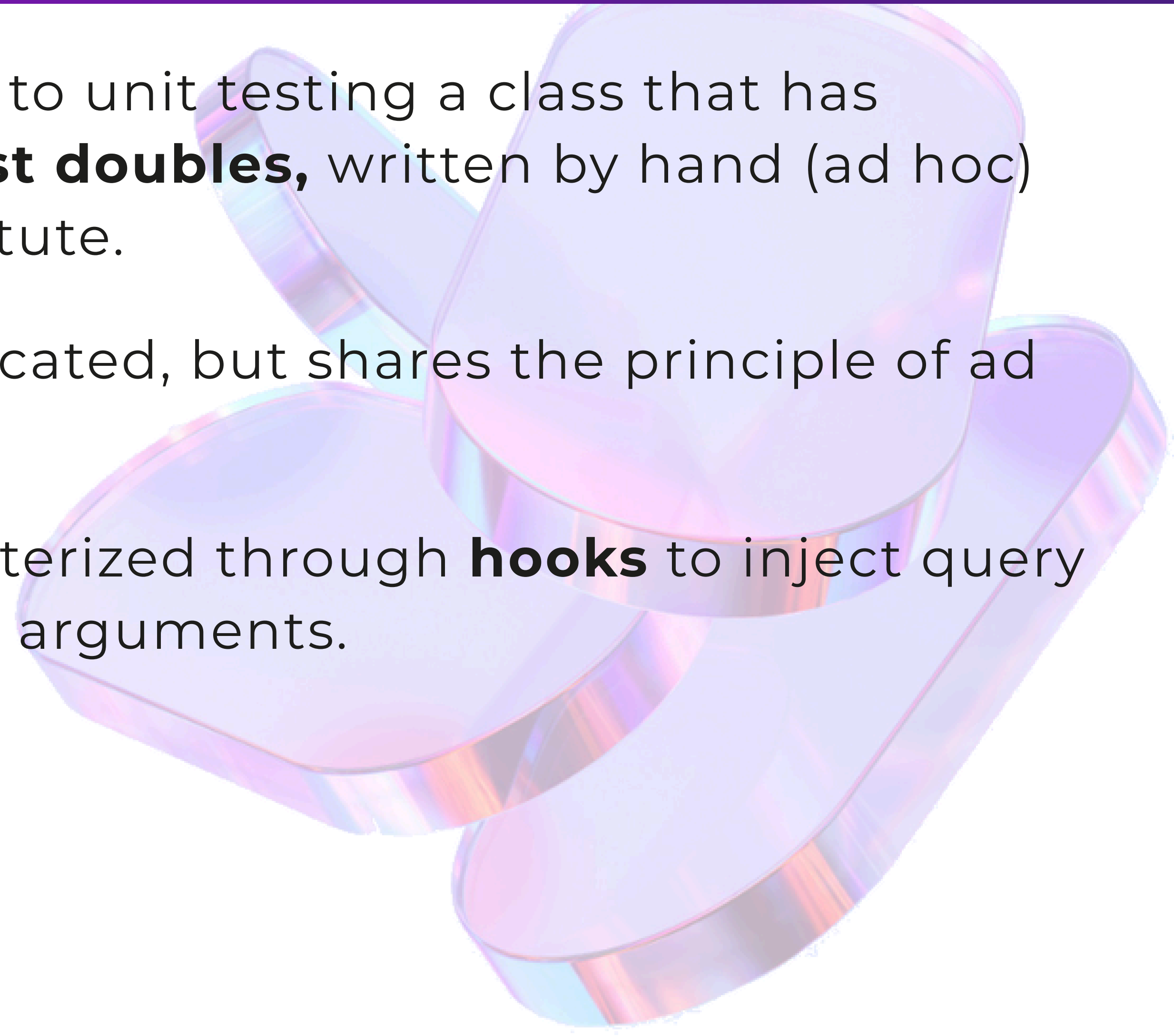


Unit testing a workflow

The traditional object-oriented approach to unit testing a class that has dependencies is to replace them with **test doubles,** written by hand (ad hoc) or using a library such as Moq or NSubstitute.

Testing a workflow is a little more complicated, but shares the principle of ad hoc test duplicates.

Main idea: a **custom interpreter,** parameterized through **hooks** to inject query output data and records command input arguments.



Unit testing a workflow: hooks

```fsharp
// Command input arguments
type HookCalls =
    { NotifiedLinks: (ChannelId * CrsHotelId * LinkStatus) list
      SentMails: Mail.MailType list }

// Query output data
type HookData =
    { ChannelDescription: Map<ChannelId, ChannelDescription>
      MappingActivation: Map<ChannelId * CrsHotelId, MappingActivation * LinkStatus> }

type Hooks = { Calls: HookCalls; Data: HookData }

module Hooks =
    let empty = { Calls = ...; Data = ... }

    let addMappingActivation key value hooks =
        { hooks with Data.MappingActivation = hooks.Data.MappingActivation |> Map.add key value }

    // ...
```

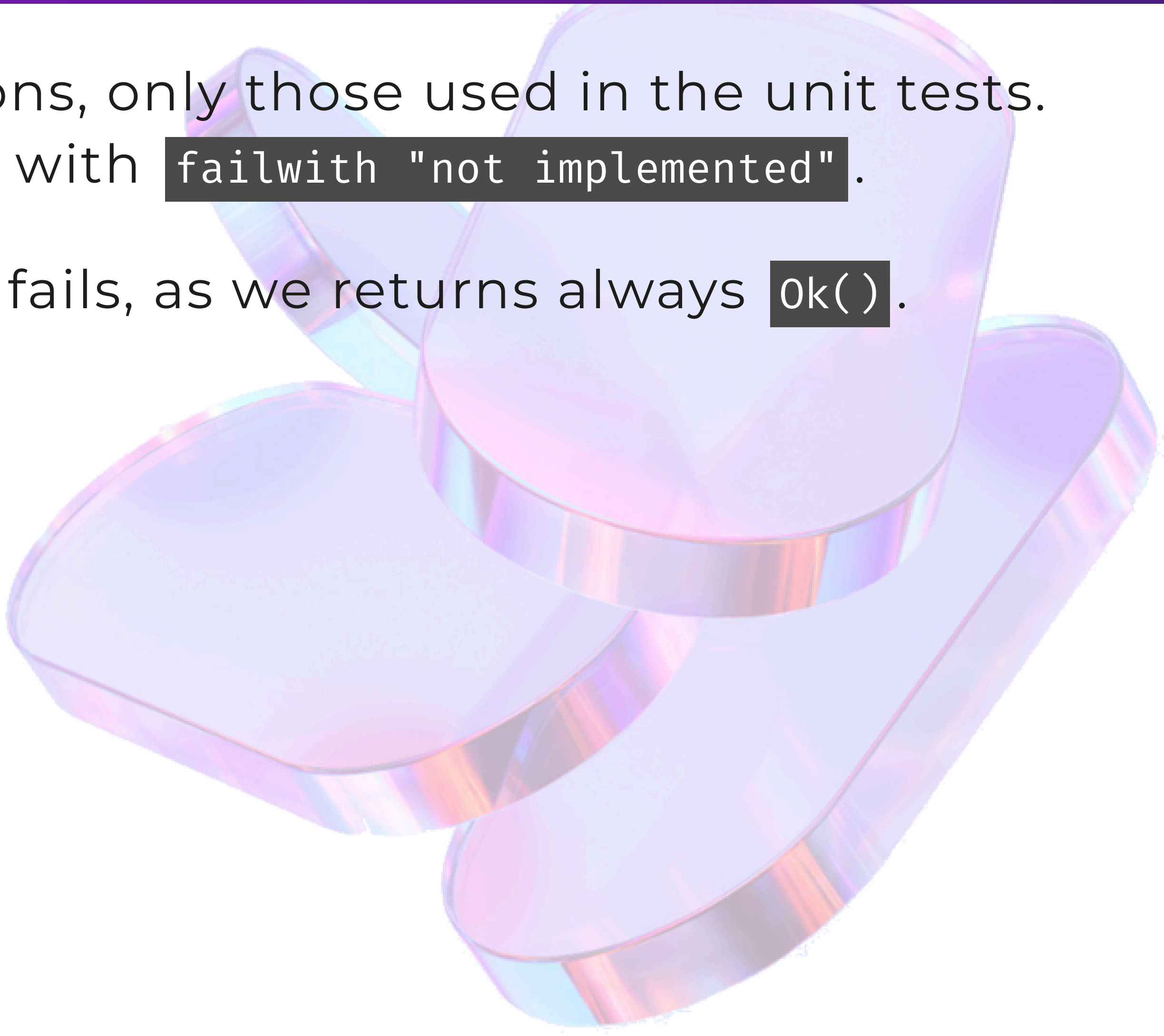
Unit testing a workflow: interpreter (1)

```
let runProgram hooks program =
    let rec loop ({ Calls = calls } as hooks: Hooks) (subprogram: Program<'T>) : Hooks * 'T =
        match subprogram with
        | Stop a → hooks, a

        | Mail(Instructions.Mail.SendMail(key, next)) →
          Ok()
          ▷ next
          ▷ loop { hooks with Calls = { calls with SentMails = key :: calls.SentMails } }

        | Partner(Instructions.Partner.NotifyLinkEvent(key, next)) →
          Ok()
          ▷ next
          ▷ loop { hooks with Calls = { calls with NotifiedLinks = key :: calls.NotifiedLinks } }

        | Partner(Instructions.Partner.GetChannelDescription(key, next)) →
          hooks.Data.ChannelDescription
          ▷ Map.tryFind key
          ▷ next
          ▷ loop hooks

        | Partner(Instructions.Partner.GetMappingActivation(key, next)) →
          hooks.Data.MappingActivation
          ▷ Map.tryFind key
          ▷ next
          ▷ loop hooks

    loop hooks program
```

Unit testing a workflow: interpreter (2)

We don't need to handle all the instructions, only those used in the unit tests.
The instructions not used can be skipped with `failwith "not implemented"`.

Limit: we can't simulate a command that fails, as we returns always `Ok()`.



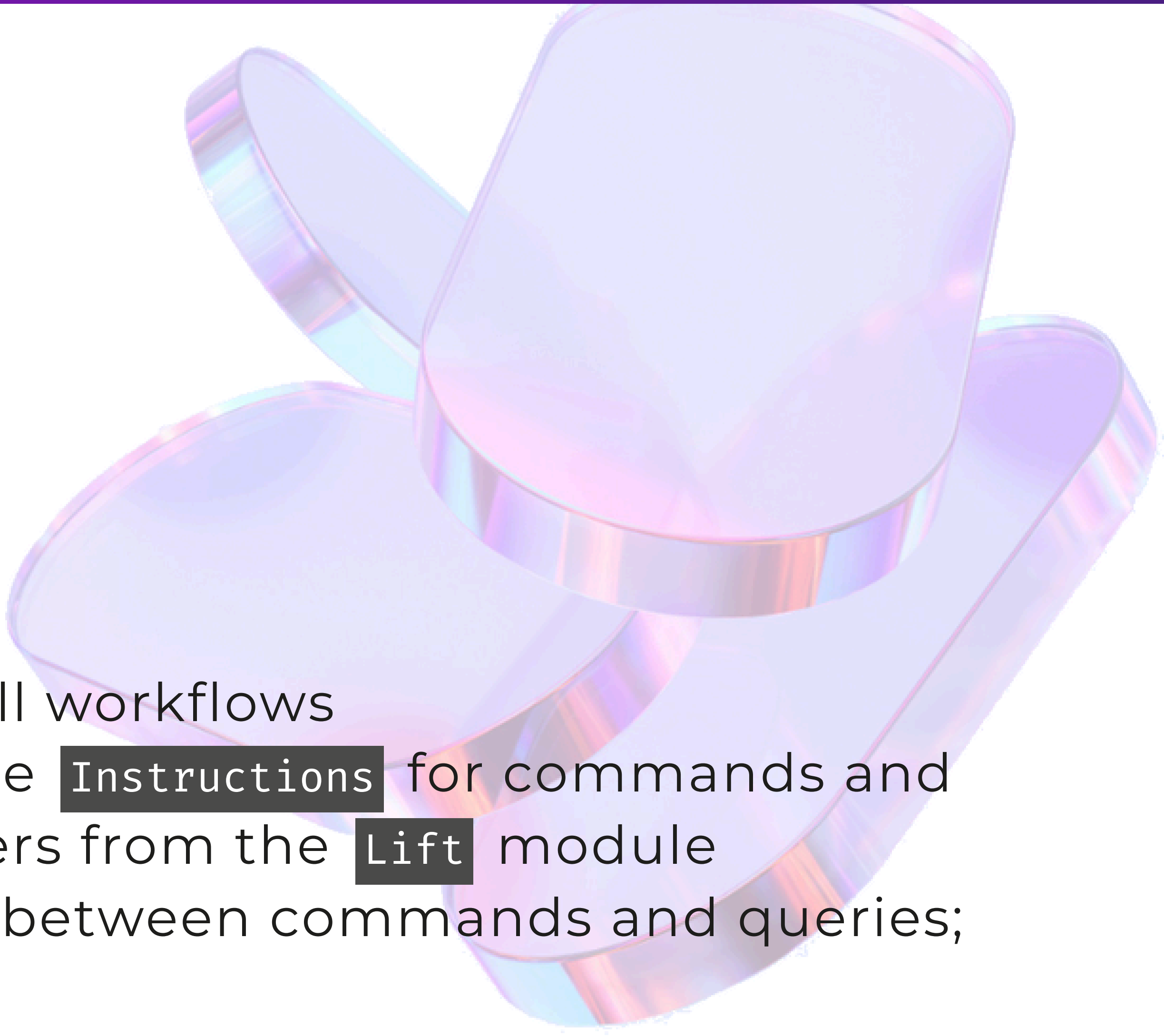
Unit testing a workflow: other helpers

```fsharp
// Code simplified for brievety

type WorkflowCheck<'success> =
    | FailedWithError of expectedError: Error
    | SucceededWithResult of expectedValue: 'success
    | SucceededWithCalls of expectedCalls: HookCalls

let checkWorkflow check (initialHooks, (hooks, result) as args) =
    match check with
    | FailedWithError expectedError → result =! Error expectedError
    | SucceededWithResult expectedValue → result =! Ok expectedValue
    | SucceededWithCalls expectedCalls →
        test <@ Result.isOk result @>
        hooks.Calls =! expectedCalls

let runWorkflow (initialHooks: Hooks) (program: Program<'T>) =
    initialHooks, runProgram initialHooks program
```

Unit testing a workflow: example

```fsharp
module MappingStatusShould =
    [<Theory; AutoDataExt>]
    let ``be deactivated given related mapping is inactive (XDC)`` channelId deactivatedHotelId =
        let initialHooks =
            Hooks.empty
            ▷ Hooks.addMappingActivation
                (channelId, deactivatedHotelId)
                (MappingActivation.Deactivated, LinkStatus.NoneOrDeleted)

        PartnerWorkflow.getMappingStatus (deactivatedHotelId, channelId, ChannelCategory.DistributionXdcChannel)
        ▷ runWorkflow initialHooks
        ▷ checkWorkflow (SucceededWithResult(Some MappingStatus.Deactivated))
```

AppStore *versus* SCM (1)



Error helper naming

- AppStore: `expectXxxError`
- SCM: `liftXxxError`

Command error return type

- AppStore: `Error`
- SCM: `DataRelatedError`

Instruction helpers

- AppStore: in `Environments.fs`, shared for all workflows
- SCM: each workflow has a private module `Instructions` for commands and queries locally used, declared using helpers from the `Lift` module
  → Pros: **separation** between workflows, between commands and queries; **flexibility** (`Lift.query` vs `Lift.instruction`)

AppStore *versus* SCM (2)

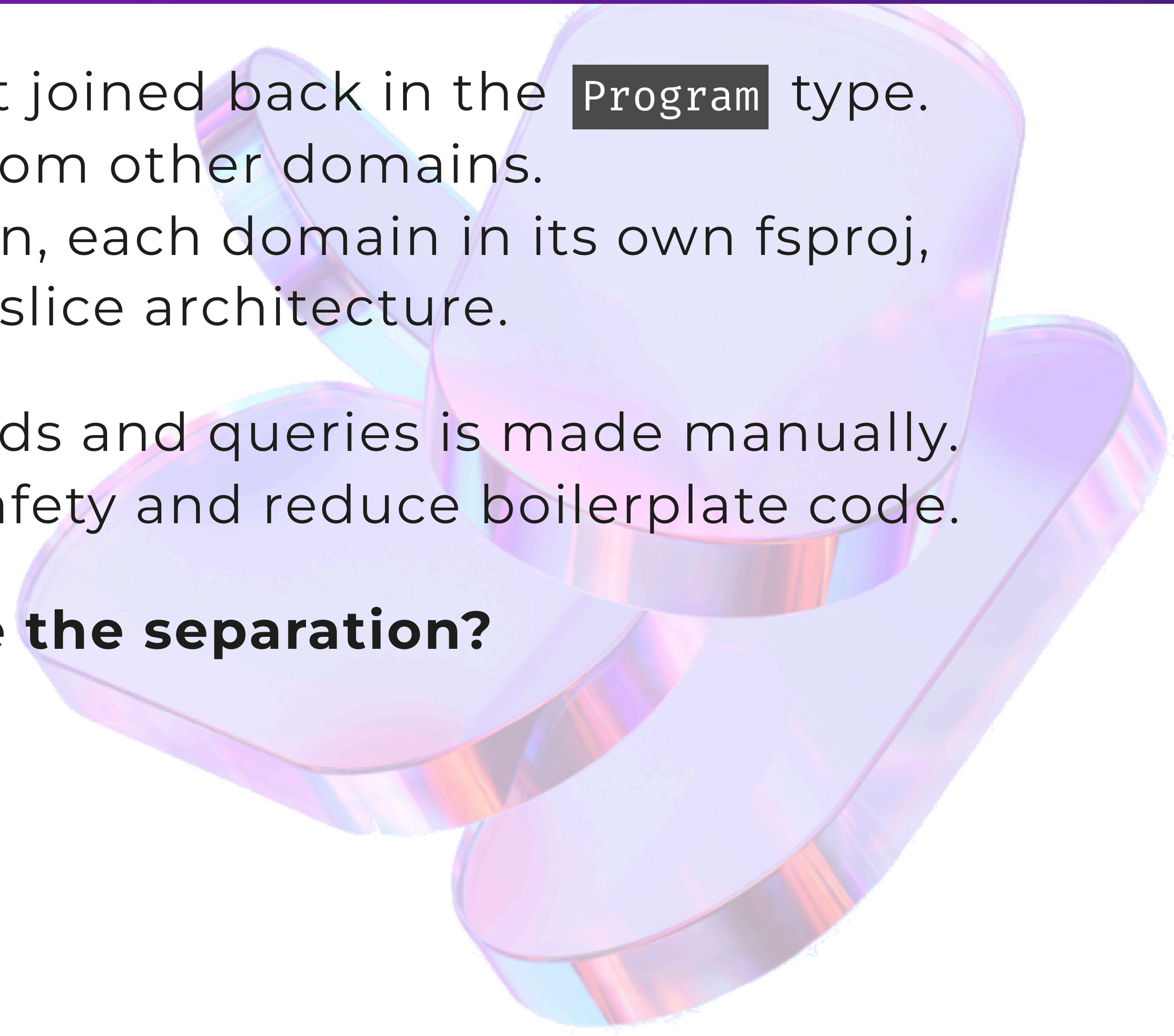
```fsharp
module Connectivity.SmartChannelManager.Domain.Workflow.SynchronizationWorkflow

[<AutoOpen>]
module private Instructions =
    let command =
        {| SynchronizePlanning = Lift.command Program.Synchronization Instructions.Synchronization.SynchronizePlanning |}

    let query =
        {| GetLastSynchronizationStatus = Lift.instruction Program.Synchronization Instructions.Synchronization.GetLastSynchronizationStatus |}

let getLastSynchronizationStatus (args: {| hotelId: string; channelId: string |}) =
    program {
        let! hotelId = createAndLiftGuardClause CrsHotelId.Create args.hotelId
        let! channelId = createAndLiftGuardClause ChannelId.FromString args.channelId

        let! syncStatus = query.GetLastSynchronizationStatus(hotelId, channelId)
        return Ok syncStatus
    }

let startSynchronizationIf (syncCondition: SyncTriggerCondition) (channelId: ChannelId, hotelId: CrsHotelId, userIdentity: UserIdentity) =
    program {
        if syncCondition.IsSatisfied then
            let! (syncId: SynchronizationId) = command.SynchronizePlanning(channelId, hotelId, userIdentity)
            return Ok(SyncTriggerResult.triggered syncId syncCondition)
        else
            return Ok(SyncTriggerResult.aborted syncCondition)
    }
```

Free monad limits

Instructions are separated by domain but joined back in the `Program` type.
→ Workflows can still used instructions from other domains.
→ We cannot perform a stricter separation, each domain in its own fsproj, to get screaming architecture or vertical slice architecture.

Instruction separation between commands and queries is made manually.
→ We cannot use types to enforce type safety and reduce boilerplate code.

How to improve this pattern to improve the separation?



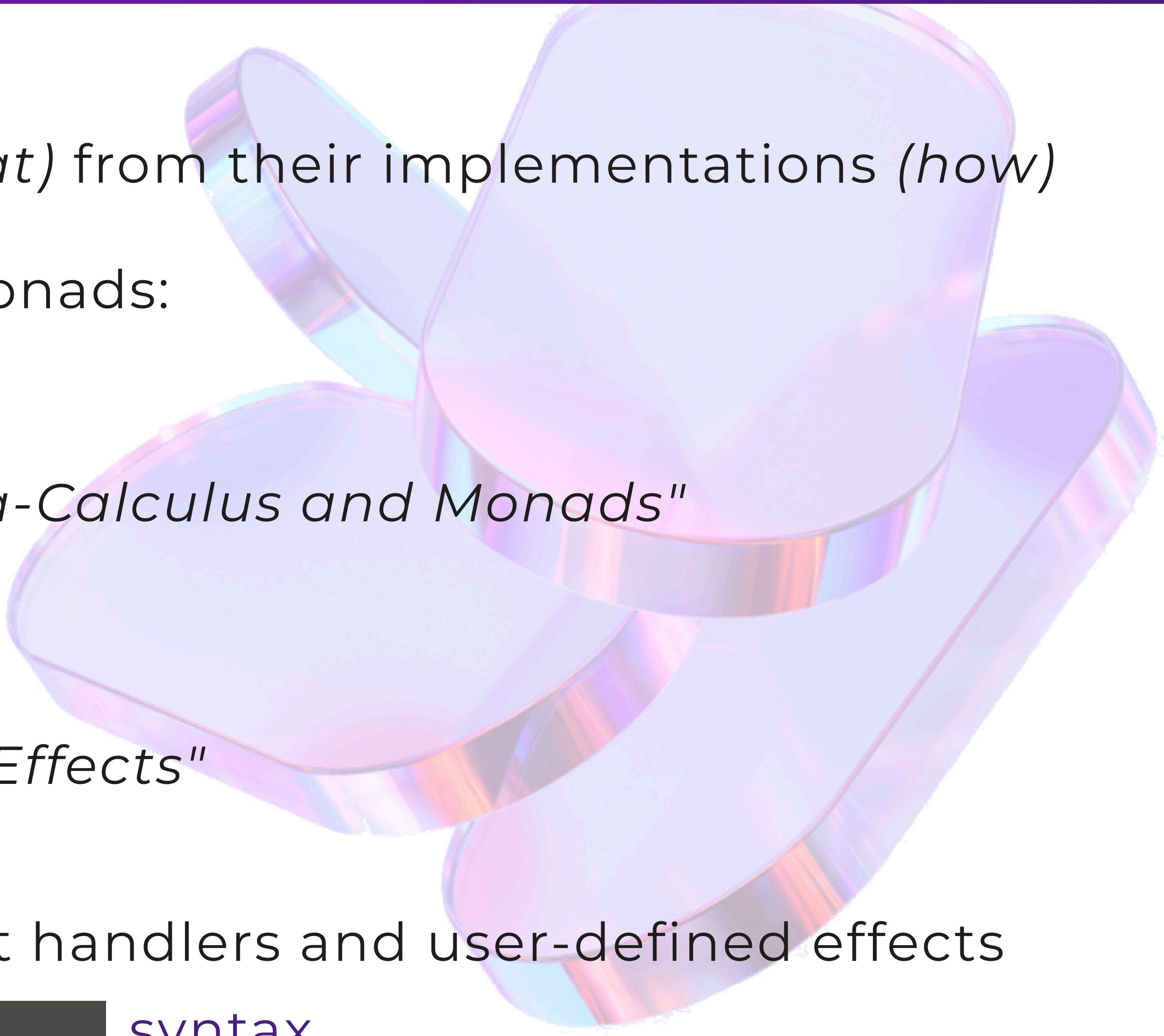
4. Algebraic effects with program CE V3

Algebraic effects

Principle: what *vs* how
→ Separating operation declarations *(what)* from their implementations *(how)*

📅 Research domain more recent than monads:

- Monads
  - 1991: paper *"Computational Lambda-Calculus and Monads"*
  - 1992: integration in Haskell
- Algebraic effects
  - 2009: paper *"Handlers of Algebraic Effects"*
  - 2012: Eff language (research)
  - 2014..2022: Multicore OCaml → effect handlers and user-defined effects
  - 2023: OCaml 5.3 → `try ... with effect ...` syntax

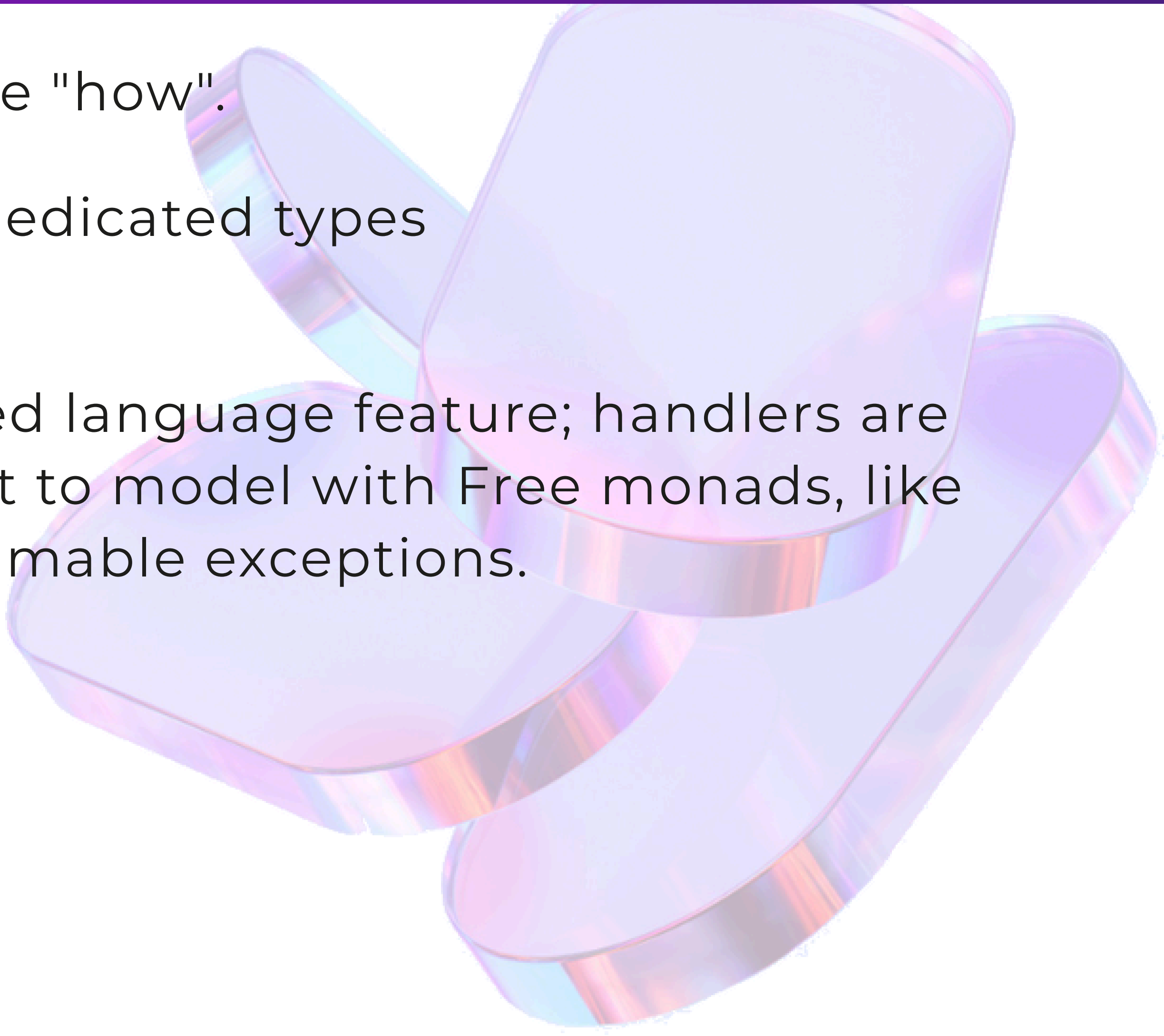


Algebraic effects *versus* Free monad

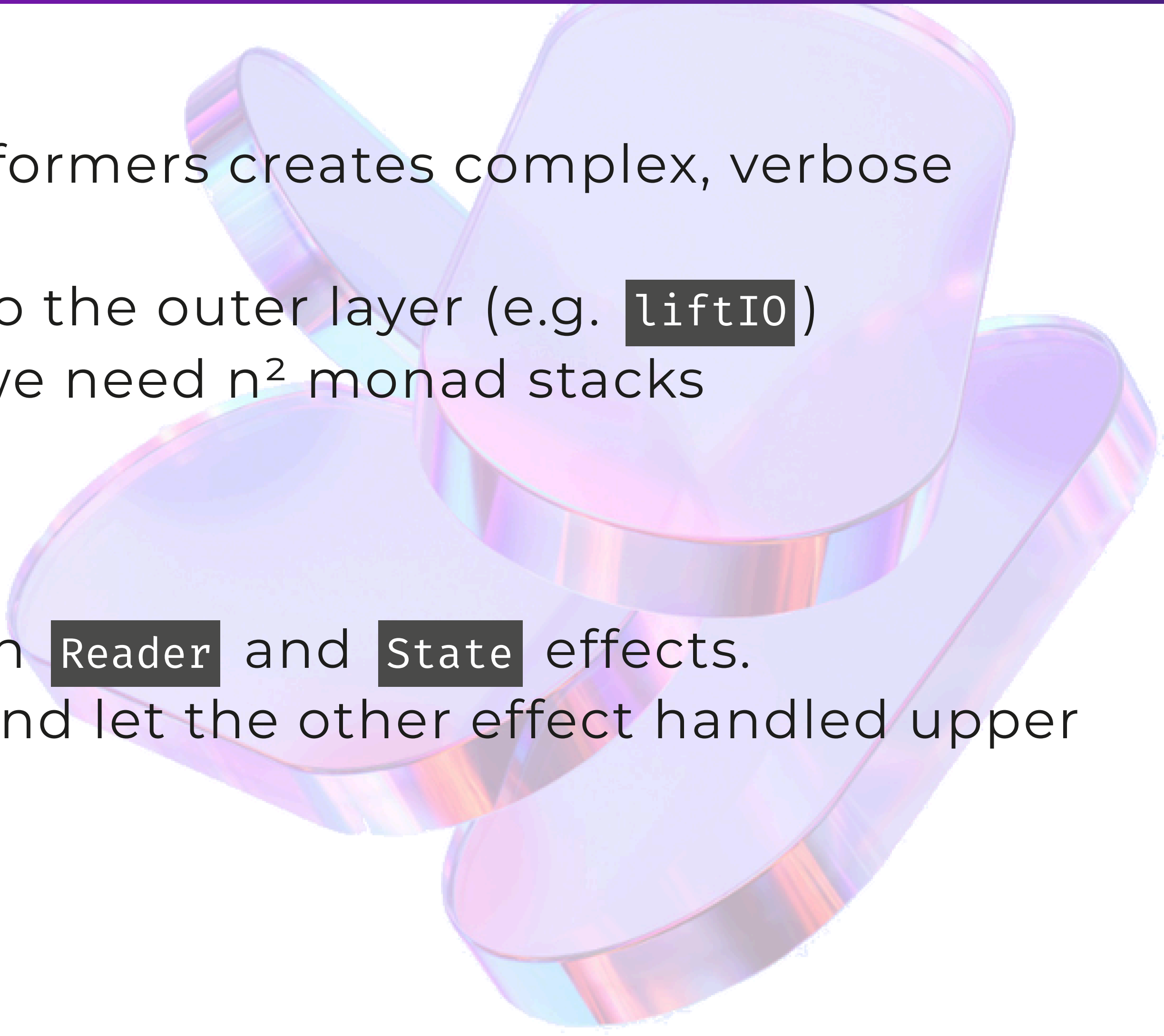
Same goal: separating the "what" from the "how".

Free monad: ad hoc solution, built with dedicated types
(see our `Instructions` and `Program` types)

Algebraic effects: native, highly optimized language feature; handlers are composable, allowing capabilities difficult to model with Free monads, like non-local control flow, generators, or resumable exceptions.



Algebraic effects *versus* Monad stacks



Monad transformers drawbacks:

- Boilerplate & Complexity: stacking transformers creates complex, verbose types.
- Constant Lifting: from an inner monad to the outer layer (e.g. `liftIO`)
- The n^2 Problem: for n monadic effects, we need n^2 monad stacks
- Rigid Stacks

Algebraic effects = set of capabilities.

- A function can declare that it needs both `Reader` and `State` effects.
- A handler can handle both, or just one and let the other effect handled upper in the call stack.

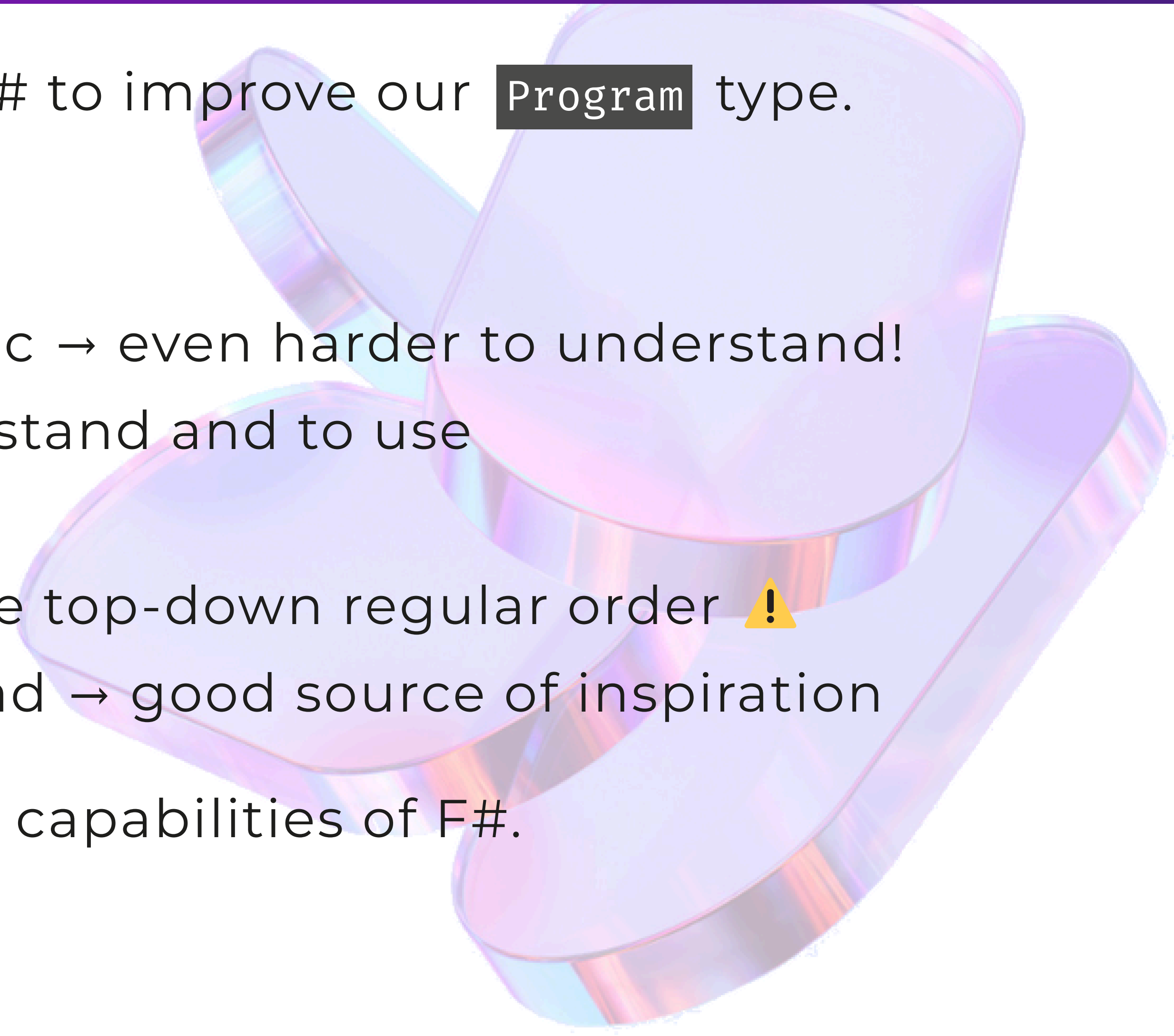
Program inspired by algebraic effects

Purpose: use alg eff implementation in F# to improve our `Program` type.

F# algebraic effects librairies:

- *Nick Palladinos'* Eff: hard to use, no doc → even harder to understand!
- *Brian Berns'* AlgEff: less hard to understand and to use
  - based on class inheritance ⚠️
  - types defined with `and`, breaking the top-down regular order ⚠️
  - overkill, but based on the free monad → good source of inspiration

Both used **generics** and **object-oriented** capabilities of F#.



Program V3 guideline

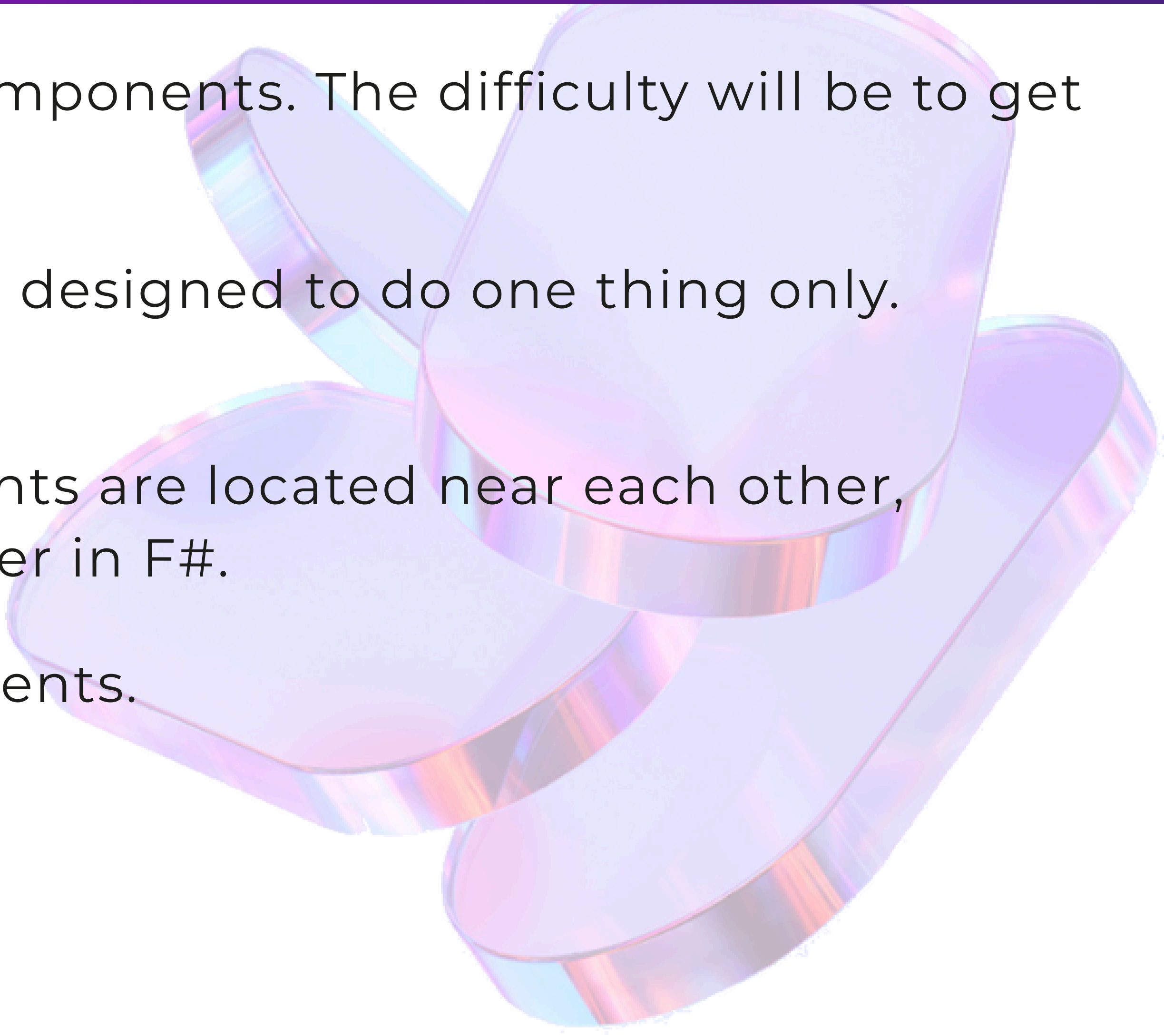
Alg eff can be implemented in F# only with generics and object-oriented, but the implementation should strive to combine simplicity and type safety.

Generics can be tricky, especially with constraints and many type parameters.
→ Here, simplicity trumps type safety.

OO design principles:

- Limited inheritance: no class hierarchy, only interfaces.
- Interface segregation principles · for generics: `I<T, U>` → `I1<T>` and `I2<U>`
- When possible, downcast an interface to an union type to get exhaustiveness.
- Use type aliases to simplify writing programs, especially with generic classes.

Program V3 core components



This new version is composed of more components. The difficulty will be to get the full picture of how it all works.

Each component is the simplest possible, designed to do one thing only.
→ Easier to understand.

Whenever possible, the related components are located near each other, declared top-down to get the regular order in F#.

Let's take a look at each of these components.

Program V3: open to any effect

This version of the `Program` is a free monad variation handling any effect that is a functor by implementing the `IProgramEffect<'a>` generic interface:

```fsharp
// Identify an effect that can be inserted in a program.
// The `Map` method satisfies the Functor laws.
[<Interface>]
type IProgramEffect<'a> =
    abstract member Map: f: ('a → 'b) → IProgramEffect<'b>

type Program<'ret> =
    // Last step in a program, containing the returned value.
    | Stop of 'ret

    // One step in a program.
    | Effect of IProgramEffect<Program<'ret>>
```

Program V3 computation expression

The `ProgramBuilder` class is almost unchanged. Only the `bind` function needs now to call the effect `Map` method:

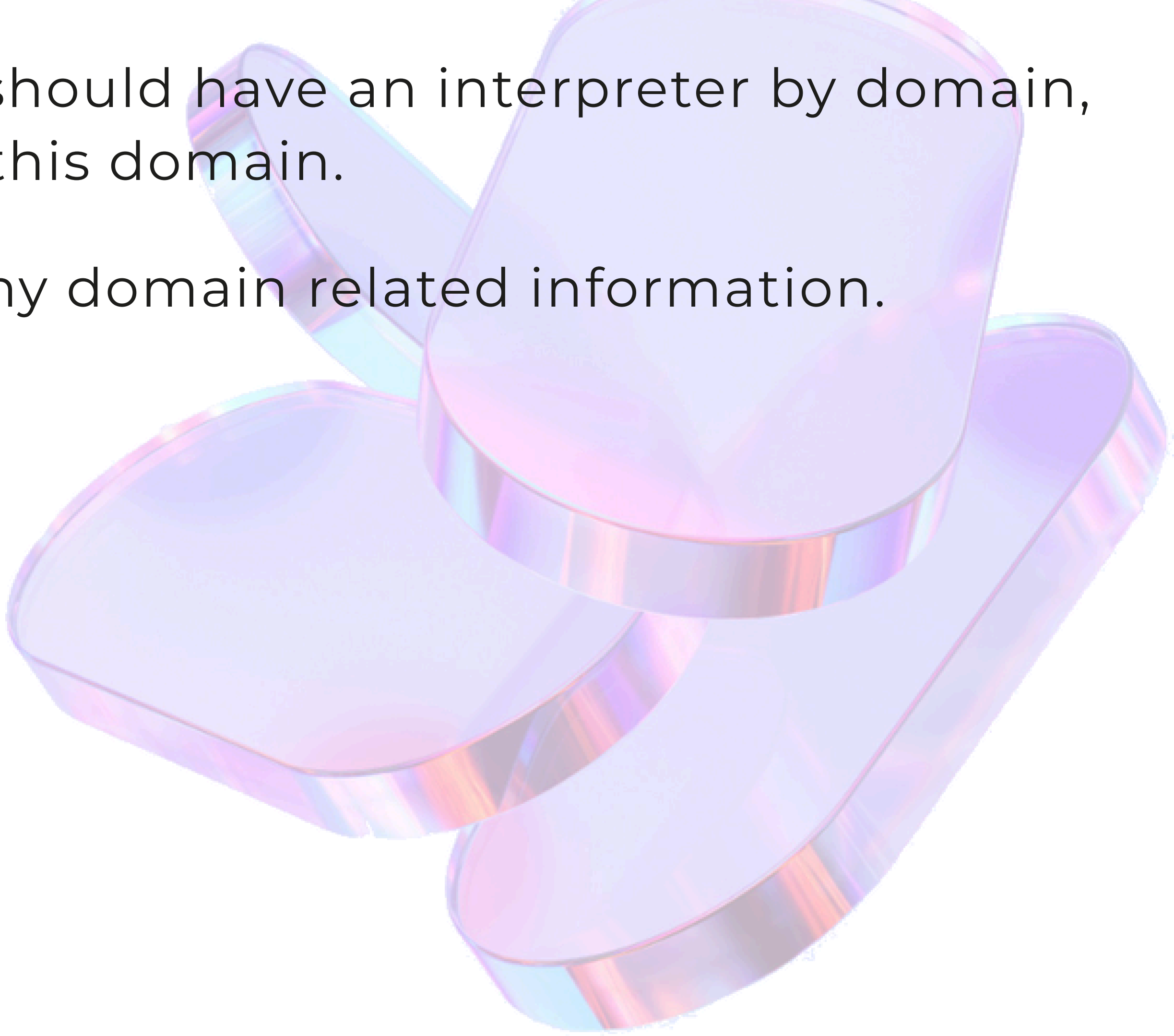
```fsharp
[<AutoOpen>]
module ProgramBuilder =
    let rec private bind f program =
        match program with
        | Stop x -> f x
        | Effect effect -> effect.Map(bind f) |> Effect

    // Same code...

    let program = ProgramBuilder()
```

Domain workflow type

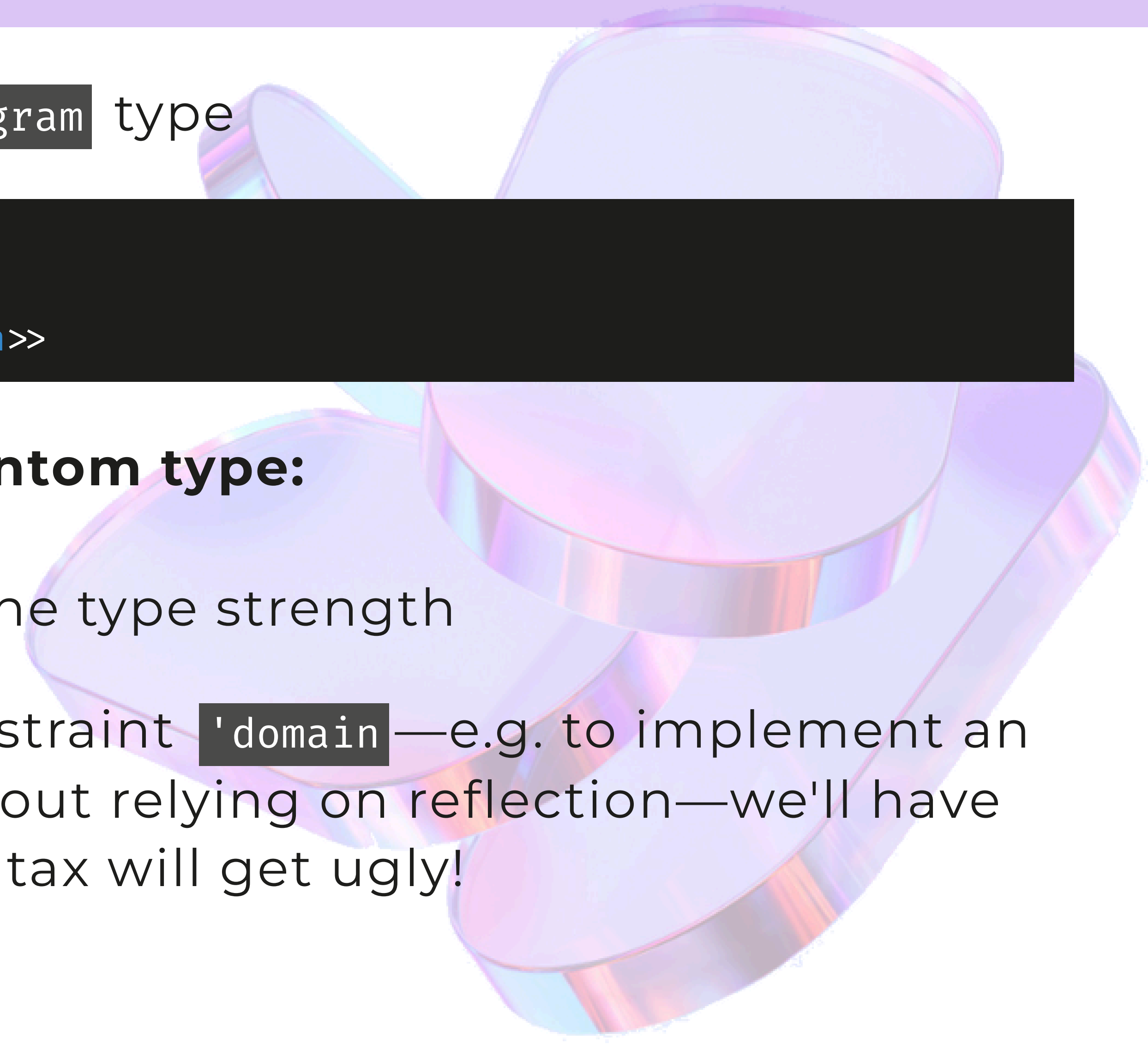
For a proper separation of concerns , we should have an interpreter by domain, handling only the effects-instructions of this domain.

Problem: our programs does not reveal any domain related information.

How to type the domain workflow?



Workflow Type #1: Generics



Add a generic type parameter to the `Program` type

```
type Program<'ret, 'domain> =
    | Stop of 'ret
    | Effect of IProgramEffect<Program<'ret, 'domain>>
```

💡 **Note:** `'domain` type parameter is a **phantom type:**
→ no runtime representation
→ only used at compile time to enforce the type strength

⚠️ **Limitation:** As long as we want to constraint `'domain`—e.g. to implement an interface and get the domain name without relying on reflection—we'll have to repeat the constraint a lot and the syntax will get ugly!

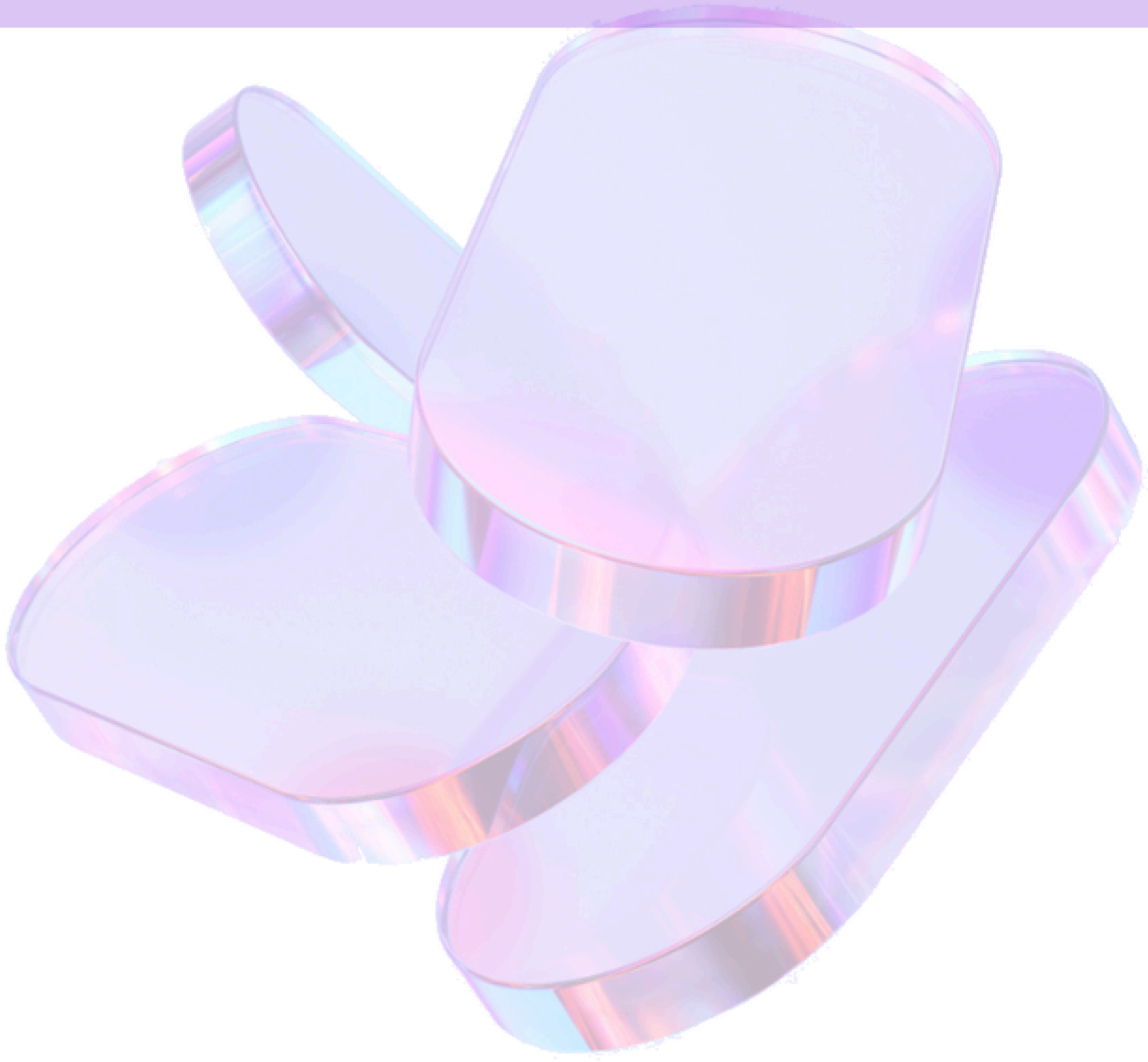
👉 **Option discarded.**

Workflow Type #2: FP

3 types of components:

- Domain type: single-case union
- Workflow type: record
- Workflows: functions

🔗 MR 1: !207



Workflow Type #2: Domain Type

Step 1: Define a type per domain

- implementing `IEffectDomain` to give access to the domain name

```fsharp
// Core/Effects/Program.fs
[<Interface>]
type IEffectDomain =
    abstract member Name: string

// Feat/Partner/Workflows/Instructions
type PartnerDomain =
    | PartnerDomain

    interface IEffectDomain with
        member _.Name = "Partner"
```

Workflow Type #2: Record

Step 2: Define the Workflow type as a **Record**

- wrapping both Program and Domain
- built by a dedicated helper in each domain

```fsharp
// Core/Effects/Program.fs
type Workflow<'ret, 'dom when 'dom :> IEffectDomain> =
    { Program: Program<Result<'ret, Error>>
      Domain: 'dom }

// Feat/Partner/Workflows/Instructions.fs
let partnerWorkflow program =
    { Program = program
      Domain = PartnerDomain }
```

Workflow Type #2: Workflows

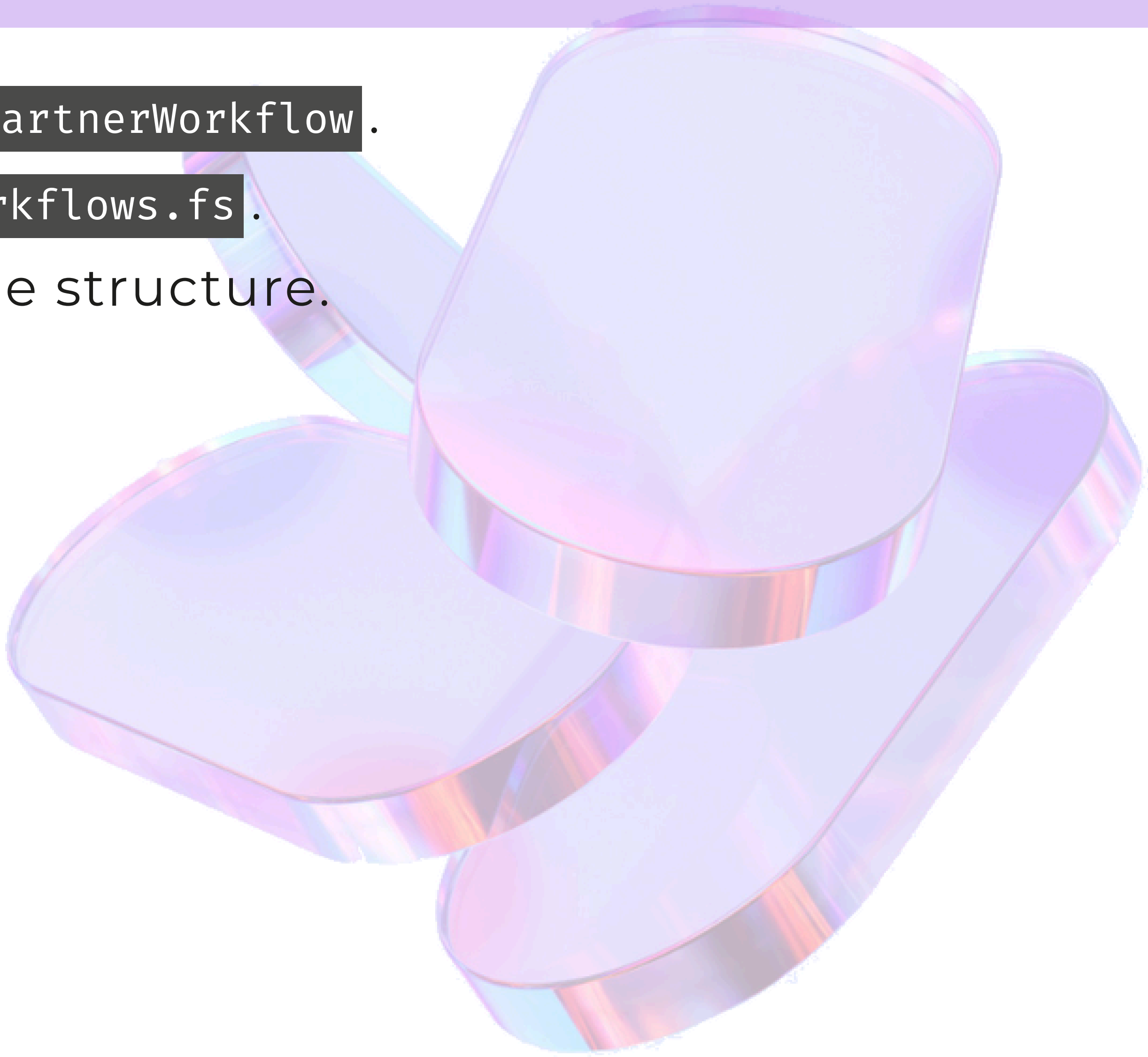
Each workflow needs to be wrapped using the appropriate domain helper:

```fsharp
// Domain/Partner/Workflows/Workflows.fs
module Dedge.AppStore.Domain.Partner.Workflows

let saveHotelIdentification (args: {| channelId: int; hotelIdentification: HotelIdentification |}) =
    program {
        let! channelId = createOrValidationError ChannelId.Create args.channelId
        do! Program.saveHotelIdentification (channelId, args.hotelIdentification)
        return Ok()
    }
    |> partnerWorkflow // 👈👈👈
```

- **Arguments** are grouped in an anonymous record for the API layer.
- Instructions are available under the **qualifier** `Program` to ease their discoverability and their identification when reading the code.

Workflow Type #2: Limits

- Risk to forget to pipe the program to `partnerWorkflow`.
- Workflows are mixed in a single file `Workflows.fs`.
- Workflows are not identifiable in the file structure.



Workflow Type #3: OOP

3 types of components:

- Domain type: single-case union (same)
- Workflow type: **abstract class**
- Workflows: **classes**, 1 file per class workflow

🔗 MR 2: !208

Workflow Type #3: Abstract class

(Same step 1: `IEffectDomain`, `HomeDomain`, `PartnerDomain`*)*

Step 2: Define the Workflow type as an **abstract class**

- `Workflow` is also an abstract class, more convenient than an interface

```fsharp
// Core/Effects/Program.fs
[<AbstractClass>]
type Workflow<'dom, 'arg, 'ret when 'dom :> IEffectDomain>() =
    abstract member Domain: 'dom
    abstract member Run: 'arg → Program<Result<'ret, Error>>

// Feat/Partner/Workflows/Instructions.fs
[<AbstractClass>]
type PartnerWorkflow<'arg, 'ret>() =
    inherit Workflow<PartnerDomain, 'arg, 'ret>()
    override val Domain = PartnerDomain
```

Workflow Type #3: File

Step 3: Define each workflow `Xxx` in a class in a dedicated file

- The file contains the eventual `XxxRequest` type for input argument *(replacing the anonymous record of solution #2)*

```
// Feat/Partner/Workflows/SaveHotelIdentification.fs
type SaveHotelIdentificationRequest = { ... }

[<Sealed>]
type SaveHotelIdentificationWorkflow() =
    inherit PartnerWorkflow<SaveHotelIdentificationRequest, unit>()

    override _.Run(args: SaveHotelIdentificationRequest) =
        program {
            let! channelId = createOrValidationError ChannelId.Create args.channelId
            let! hotelId = createOrValidationError CrsHotelId.Create args.hotelId
            let! hotelCode = createOrValidationError DistributorHotelId.Create args.hotelCode

            do! Program.saveHotelIdentification (channelId, hotelId, hotelCode, args.hotelIdentification)

            return Ok()
        }
```

Effect holding Instructions

To complement `IProgramEffect<'a>`, we define another interface.

`IInterpretableEffect<'union>` links an effect with a set of **instructions** given by its `'union` type parameter.

```fsharp
[<Interface>]
type IInterpretableEffect<'union> =
    abstract member Instruction: 'union
```

Notes:

- `'union` is usually an union type, but it's not mandatory.
- **Interpretable** because it will be used while interpreting the program.

Instruction

The instructions are defined with a single sealed class `Instruction` replacing the `Program` V2 cases `Instruction of Arg * cont: (Ret → 'a)`:

```
[<Sealed>]
type Instruction<'arg, 'ret, 'a>(name: string, arg: 'arg, cont: 'ret → 'a) =
    member val Name = name
    member _.Map(f: 'a → 'b) = Instruction(name, arg, cont >> f)
    member _.Run(runner) = let ret = runner arg in cont ret
```

- `Name`: informative, usable for logging or debugging
- `arg`: private argument(s) for this instruction
- `cont`: continuation function, passing the result to the next instruction
- `Map`: functor `map` operation: chain the continuation with the given function
- `Run`: call the `runner: 'arg → 'ret` to get the result and continue with it

Commands and Queries

- Are instructions but do not inheriting from `Instruction` to avoid:
  - class inheritance
  - complex 3 type parameters passing
- Defined through simple type aliases, that works as constructors 👍

```
type Command<'arg, 'a>                 = Instruction<'arg, Result<unit, Error>, 'a>
type Query<'arg, 'ret, 'a>             = Instruction<'arg, 'ret option,         'a>
type QueryFailable<'arg, 'ret, 'a> = Instruction<'arg, Result<'ret, Error>, 'a>
```

Domain project (MR 1 !207)

There is still one Domain project for now, but it's organized by domain:

```
📂 Core/
├─🗃️ dedge.appstore.core/
│  └─📂 Effects/
│     ├─📄 Prelude.fs        👉 Effects, Instructions
│     └─📄 Program.fs        👉 Program type and companion module, program CE
├─🗃️ dedge.appstore.domain/
│  ├─📂 Mail/                👉 Mail domain folder
│  │  ├─📄 Instructions.fs
│  │  └─📄 Workflows.fs
│  └─📂 Partner/             👉 Partner domain folder
│     └─ ...
└─🗃️ dedge.appstore.infrastructure/
   ├─📄 Interpreter.fs
   └─📄 Api.fs               👉 MailApi, PartnerApi types, exposing their interpreted workflows
```

Feat projects (MR 2 !208)

New `Feat` solution folder · New projects per domain

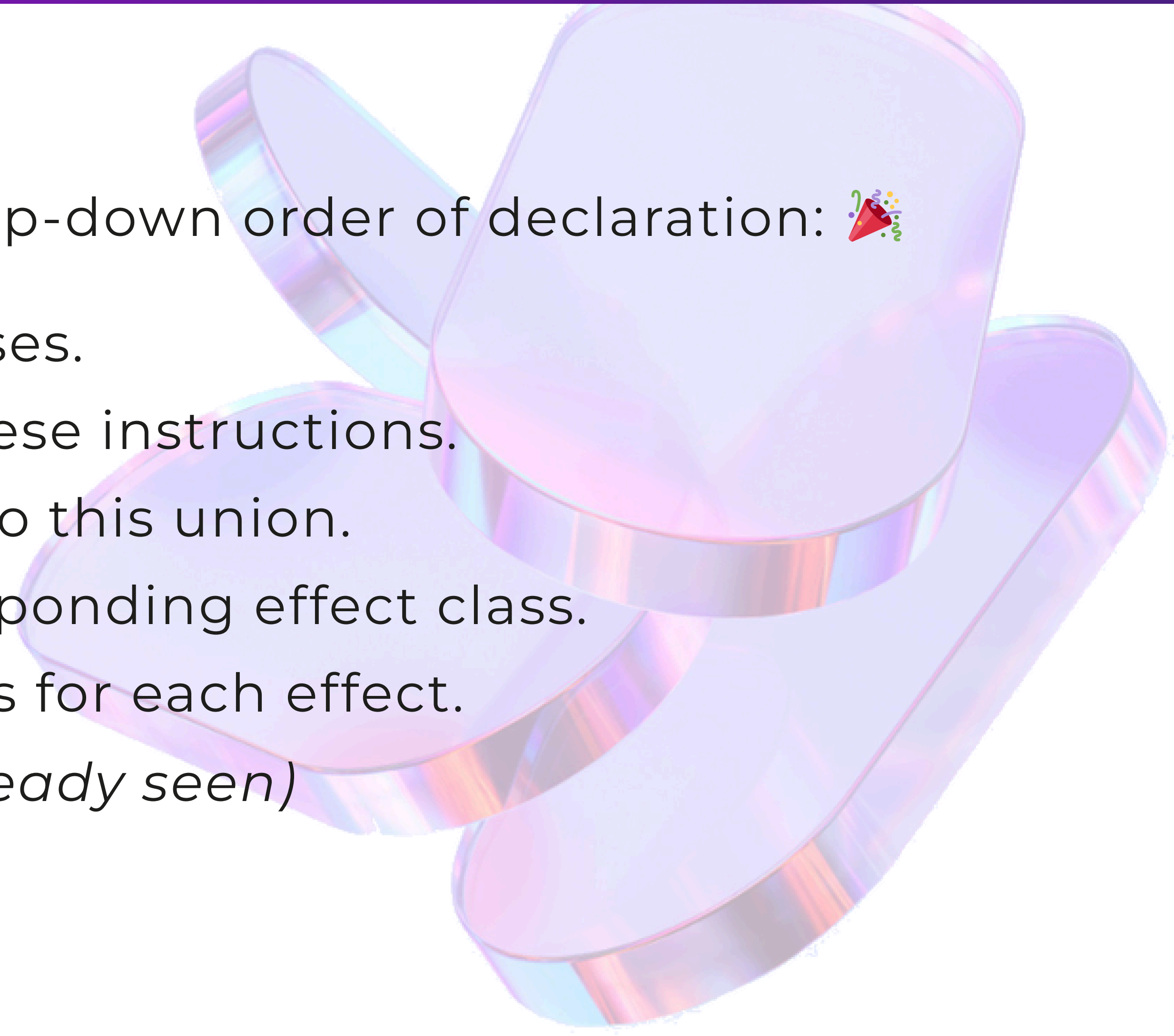
```
📂 /
├──📂 Core/
│  └──🗃️ dedge.appstore.core/
│     └──📂 Effects/
│        ├──📄 Prelude.fs          👉 Effects, Instructions
│        ├──📄 Program.fs          👉 Program type and companion module, program CE
│        └──📄 Interpreter.fs      👉 Interpreter type
└──📂 Feat/
   ├──🗃️ dedge.appstore.home/      👉 Home domain project
   │  ├──📂 Workflows/
   │  │  ├──📄 Instructions.fs
   │  │  ├──📄 ...                 👉 Other workflows
   │  │  └──📄 Search.fs           👉 Search workflow
   │  └──📄 Api.fs                 👉 Api type, exposing interpreted workflows
   └──🗃️ dedge.appstore.partner/   👉 Partner domain project
      └── ...
```

Domain instructions (1/3)

Each domain defines its instructions.

It's done in 6 steps 😅, with the regular top-down order of declaration: 🎉

1. Define all queries and commands aliases.
2. Define the union type gathering all these instructions.
3. Define the effect interface dedicated to this union.
4. For each instruction, define the corresponding effect class.
5. In a `Program` module, define the helpers for each effect.
6. Define the domain workflow type. *(already seen)*



Domain instructions (2/3)

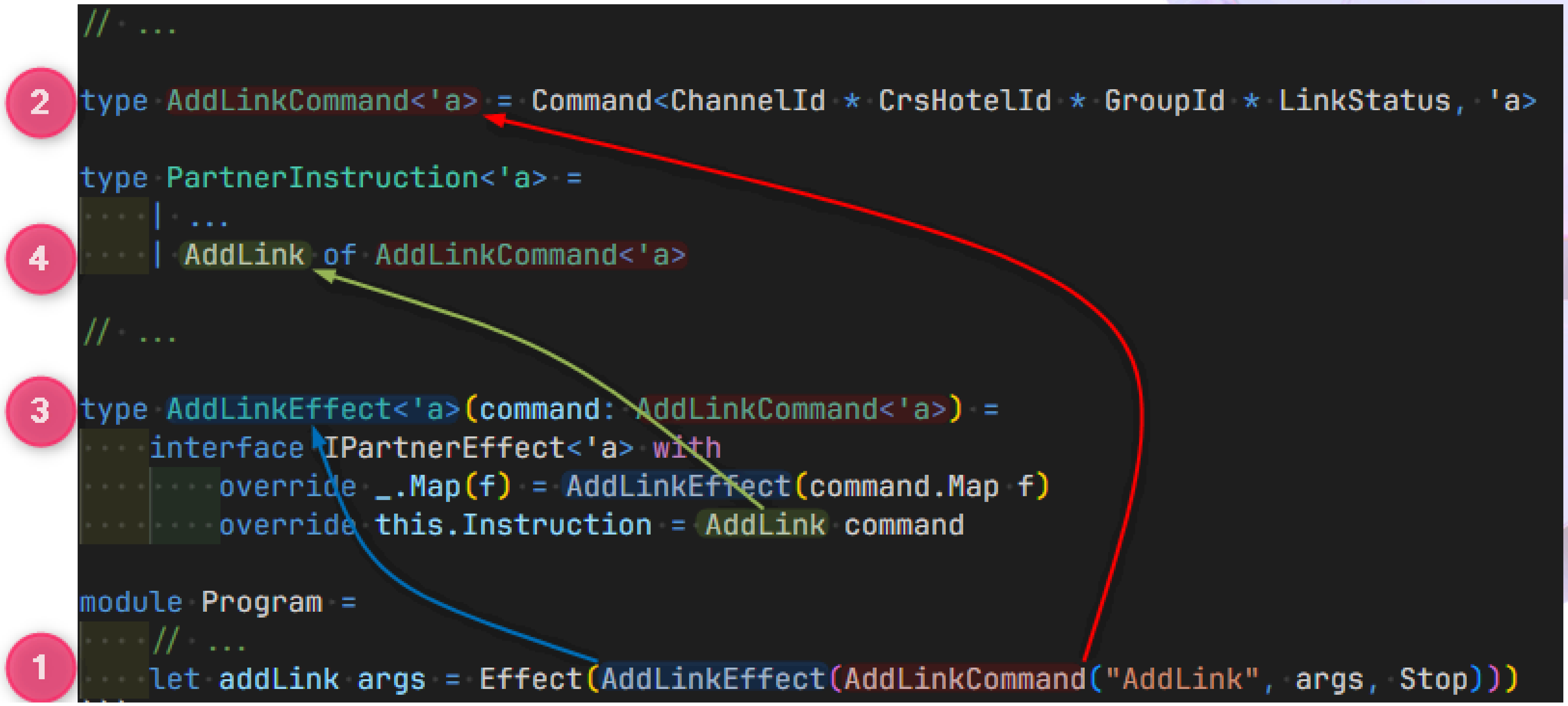
```fsharp
module Dedge.AppStore.Domain.Mail.Instructions

open Dedge.AppStore.Core.Effects
open Dedge.AppStore.Domain.Types.Mail

// Step 1 - queries and commands aliases
type GetTranslationsQuery<'a> = Query<Locale * PageCode list, Translations, 'a>
type SendMailCommand<'a> = Command<MailType, 'a>

// Step 2 - union type gathering all instructions
type MailInstruction<'a> =
    | GetTranslations of GetTranslationsQuery<'a>
    | SendMail of SendMailCommand<'a>

// Step 3 - effect interface linked to the union
[<Interface>]
type IMailEffect<'a> =
    inherit IProgramEffect<'a>
    inherit IInterpretableEffect<MailInstruction<'a>>

// Steps 4 and 5 ...
```

Domain instructions (3/3)

```fsharp
// Step 4: Effects by instruction
type GetTranslationsEffect<'a>(query: GetTranslationsQuery<'a>) =
    interface IMailEffect<'a> with
        override _.Map(f) = GetTranslationsEffect(query.Map f)
        override val Instruction = GetTranslations query

type SendMailEffect<'a>(command: SendMailCommand<'a>) =
    interface IMailEffect<'a> with
        override _.Map(f) = SendMailEffect(command.Map f)
        override val Instruction = SendMail command

// Step 5: Program instruction helpers
[<RequireQualifiedAccess>]
module Program =
    let getTranslations args =
        Effect(GetTranslationsEffect(GetTranslationsQuery("GetTranslations", args, Stop)))

    let sendMail args =
        Effect(SendMailEffect(SendMailCommand("SendMail", args, Stop)))
```

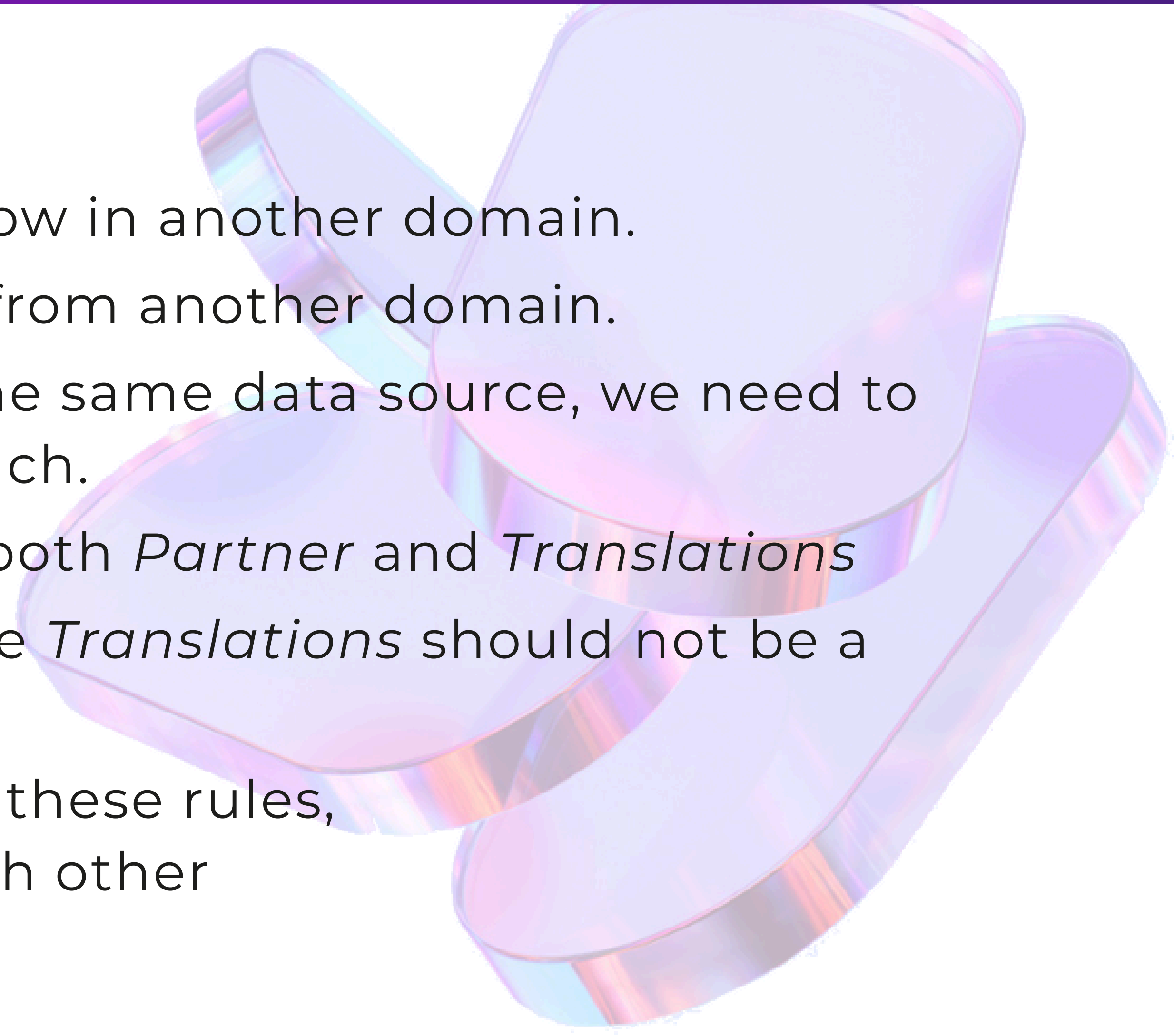
Domain instruction declaration overview

- An instruction is declared in 4 steps, but it's only boilerplate.
- Can be done bottom-up, fixing the compiler errors as we go along.



```
// ...

type AddLinkCommand<'a> = Command<ChannelId * CrsHotelId * GroupId * LinkStatus, 'a>

type PartnerInstruction<'a> =
    | ...
    | AddLink of AddLinkCommand<'a>

// ...

type AddLinkEffect<'a>(command: AddLinkCommand<'a>) =
    interface IPartnerEffect<'a> with
        override _.Map(f) = AddLinkEffect(command.Map f)
        override this.Instruction = AddLink command

module Program =
    // ...
    let addLink args = Effect(AddLinkEffect(AddLinkCommand("AddLink", args, Stop)))
```

Domain workflows isolation



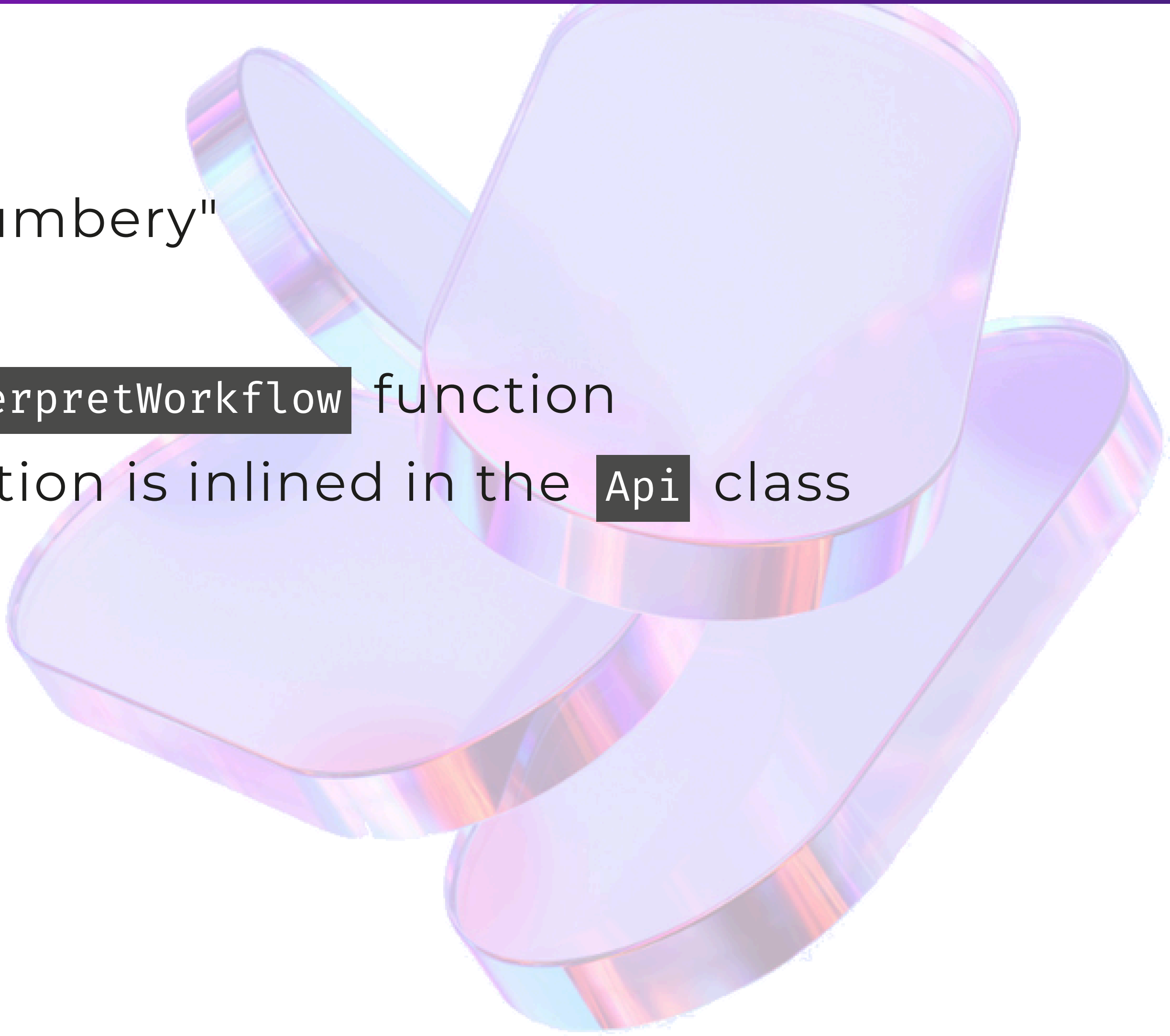
Rules to ensure proper domain isolation:

- A workflow cannot call another workflow in another domain.
- A workflow cannot call an instruction from another domain.
  - If several domains need to access the same data source, we need to declare a separate instruction for each.
  - Example: `GetTranslations` is used by both *Partner* and *Translations*
  - But it's a contrived example because *Translations* should not be a domain on is own...
- Solution #3 (domain projects) ensures these rules, as long as projects don't reference each other

Program interpreters

2 parts:

- `Interpreter` class: domain agnostic, "plumbery"
- For each domain:
    - MR 1 !207: a module defining an `interpretWorkflow` function
    - MR 2 !208: the `interpretWorkflow` function is inlined in the `Api` class



interpretWorkflow functions (1)

- `runEffect`: inner function, matching instructions with *Data* async pipelines
- final `interpret.Workflow runEffect`

```fsharp
module Dedge.AppStore.Infrastructure.Interpreter

// ...

module Category =
    open Dedge.AppStore.Domain.Category.Instructions

    let interpretWorkflow (dependencies: Dependencies) =
        let interpret = Interpreter(dependencies, CategoryDomain) // Interpreter<CategoryDomain>

        let runEffect (categoryEffect: ICategoryEffect<_>) =
            match categoryEffect.Instruction with
            | FindChannels query → interpret.Query(query, Channel.Pipeline.findChannels dependencies.ChannelApi)
            | ...

        fun runWorkflow args → interpret.Workflow runEffect (runWorkflow args)

// ...
```

interpretWorkflow functions (2)

```fsharp
//  ...

module Partner =
    open Dedge.AppStore.Domain.Partner.Instructions

    let interpretWorkflow (dependencies: Dependencies) =
        let interpret = Interpreter(dependencies, PartnerDomain) // Interpreter<PartnerDomain>

        let runEffect (partnerEffect: IPartnerEffect<_>) =
            match partnerEffect.Instruction with
            | GetTranslations query -> interpret.Query(query, Translations.Pipeline.getTranslations dependencies.TranslationsApi)
            | SendMail command -> interpret.Command(command, Mail.Pipeline.sendMail dependencies.MailSender)

        fun runWorkflow args -> interpret.Workflow runEffect (runWorkflow args)

//  ...
```

Interpreter class (1)

2 parts:

- `Command` and `QueryXxx` helpers: call `instruction.RunAsync` w/ *Logging*, *Timing*
- `Workflow` final method: call `runEffect` recursively, return `Async<Result<'a,Error>>`

A lot of code comes directly from V2.

Interpreter class (2)

```fsharp
[<Sealed>]
type private Interpreter<'dom when 'dom :> IEffectDomain>(dependencies: Dependencies, domain: 'dom) =
    let logger = dependencies.LoggerFactory.CreateLogger $"Dedge.AppStore.Domain.%s{domain.Name}.Workflow"
    let monitoring = Monitoring.StatsdAdapter.Timer(dependencies.EnvironmentName, dependencies.StatsSender)

    member private _.Instruction(instruction: Instruction<_, _, _>, withTiming, pipeline: 'arg → Async<_>) =
        let pipelineWithMonitoring =
            pipeline
            ▷ logifyPlainAsync logger (dependencies.GetContext()) instruction.Name
            ▷ withTiming instruction.Name

        instruction.RunAsync(pipelineWithMonitoring)

    member this.Command(command: Command<_, _>, pipeline) =
        this.Instruction(command, monitoring.timeAsyncResult, pipeline)

    member this.Query(query: Query<_, _, _>, pipeline) =
        this.Instruction(query, monitoring.timeAsyncOption NoneMeansFailed, pipeline)

    // ...
```

Interpreter class (3)

```fsharp
    // ...
    member _.Workflow<'a, 'effect when 'effect :> IProgramEffect<Program<Result<'a, Error>>>>(runEffect) =
        let rec loop program =
            match program with
            | Stop res → async { return res }
            | Effect eff →
                match eff with
                | :? 'effect as effect →
                    async {
                        let! res = runEffect effect
                        return! loop res
                    }
                | _ → failwithf $"Unsupported effect: %A{eff}" // ⚠️ only 1 type of effects - runtime check!

        fun (workflow: Workflow<_, 'dom>) →
            async {
                try
                    return! loop workflow.Program
                with FirstException exn →
                    return bug exn
            }
```

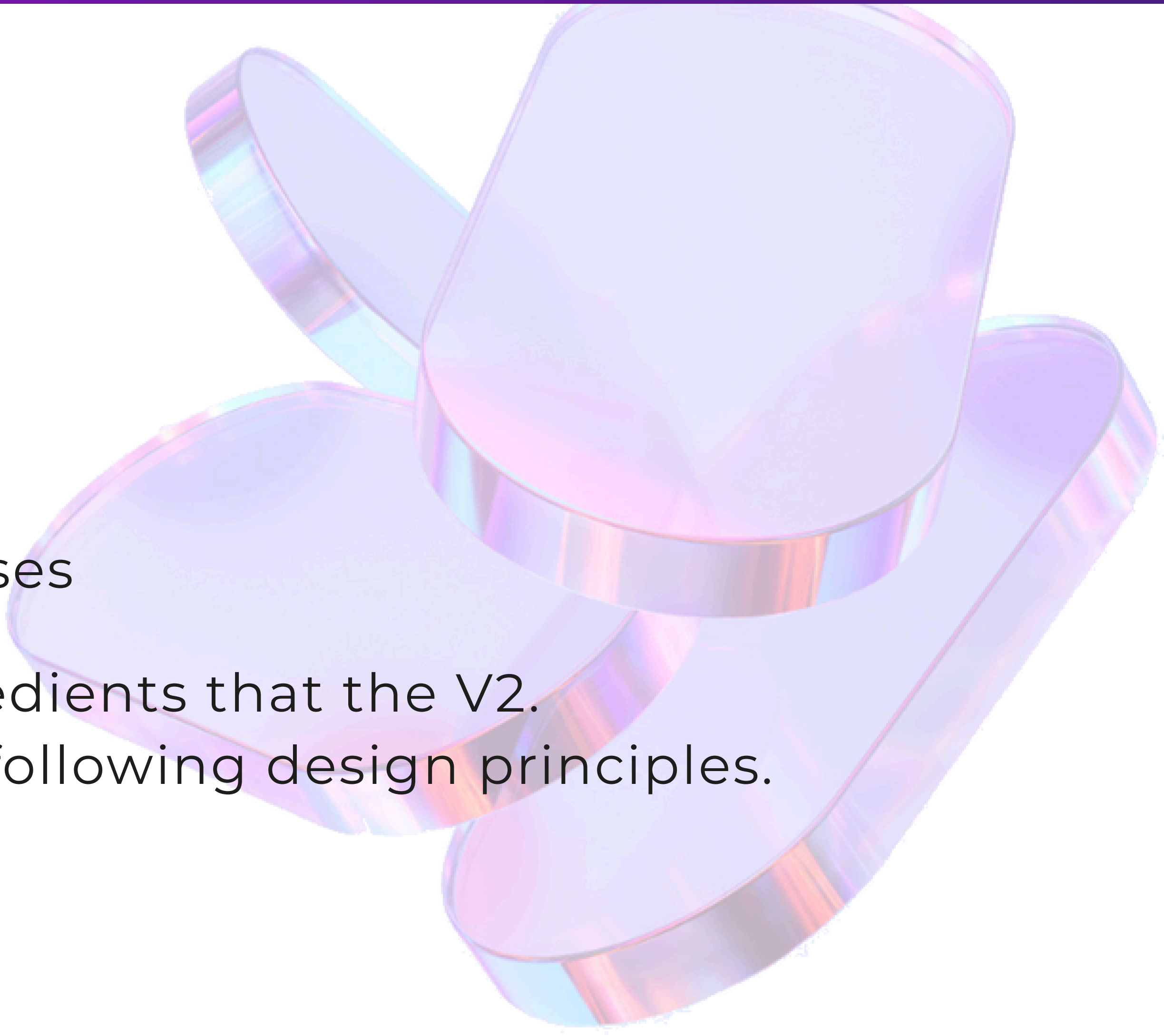
5. Conclusion

Program V3 recap

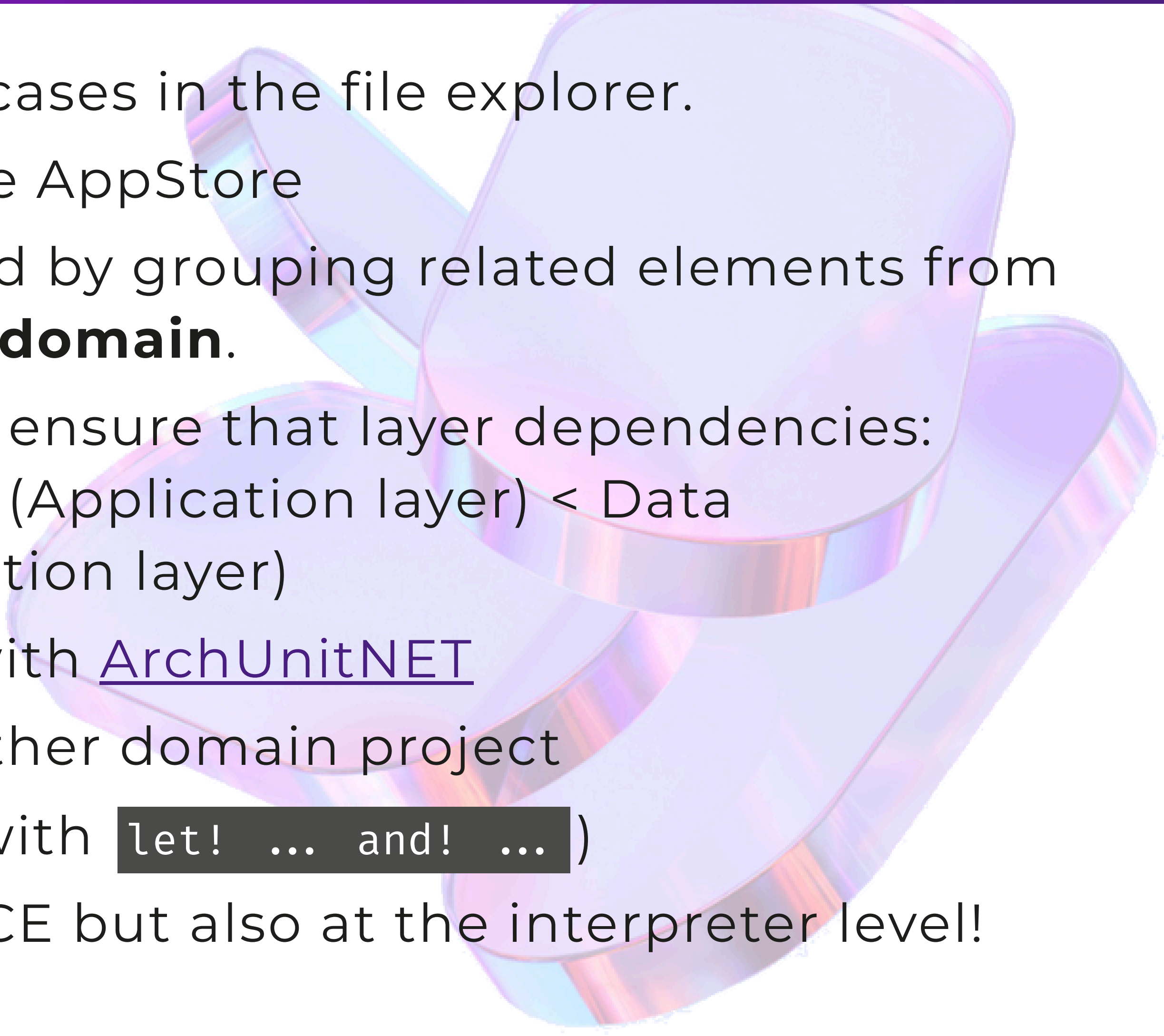
With this version, we are able to:

- Split apart each domain
  - Currently in separated modules
  - Possibly in separated projects
- Identify the type of instructions
  - with the `Command` and `Query` type aliases

The recipe to apply is made of more ingredients that the V2.
Still, they are simple and well separated, following design principles.



Possible improvements



- Split the `Workflow.fs` files to reveal use cases in the file explorer.
  - More relevant in the SCM than in the AppStore
- The split by domain could be continued by grouping related elements from all layers into a **dedicated project per domain**.
  - Leverage compilation order in F# to ensure that layer dependencies: Domain model < Domain workflows (Application layer) < Data (Infrastructure layer) < Api (Presentation layer)
  - Ensure domain project separation with ArchUnitNET
    - no domain project reference another domain project
- Performance: instructions in parallel (with `let! ... and! ...`)
  - Difficulty: to handle not only in the CE but also at the interpreter level!

Thanks 🙏