# Functional Domain Modeling

*In TypeScript*

*told by*
**Nicolas Carlo**

*ConFoo 2025*

@nicoespeon

# Nicolas Carlo

## Freelance Web Developer

Special skills:

- Legacy Code Sorcery
- Community Building

# TS won’t catch all type errors

## any

No time to type properly…
put TS to sleep

## @ts-ignore

Fighting against types…
let’s bail out

## x *as* Y

It’s not what you think…
please trust me

# It takes discipline & experience

~~@ts-ignore~~

`@ts-expect-error`

~~any~~

**unknown**


x *as* Y

*Sometimes* necessary


Brace yourself, bugs are coming

# As you grow, it will get worse

*aka “we spend more time fixing regressions than shipping features”*

# Let’s write automated tests!

*“If you liked it,*
*you shoulda put a test on it”*

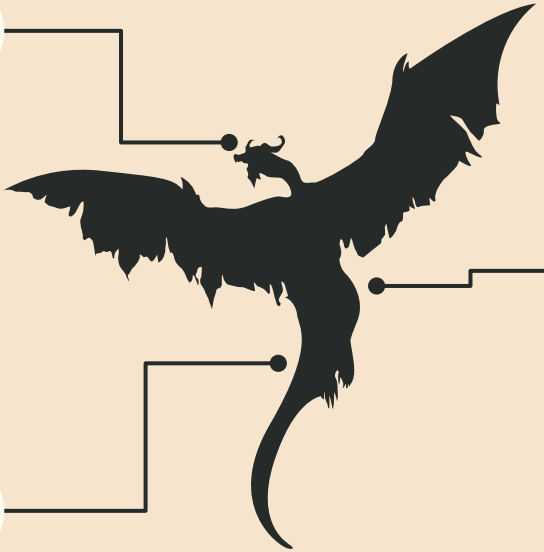
— The Beyoncé Rule

# But that won’t be enough...

## Time

Tedious work
on legacy code

1



## Skills

Approval Testing,
Mikado Method...

2

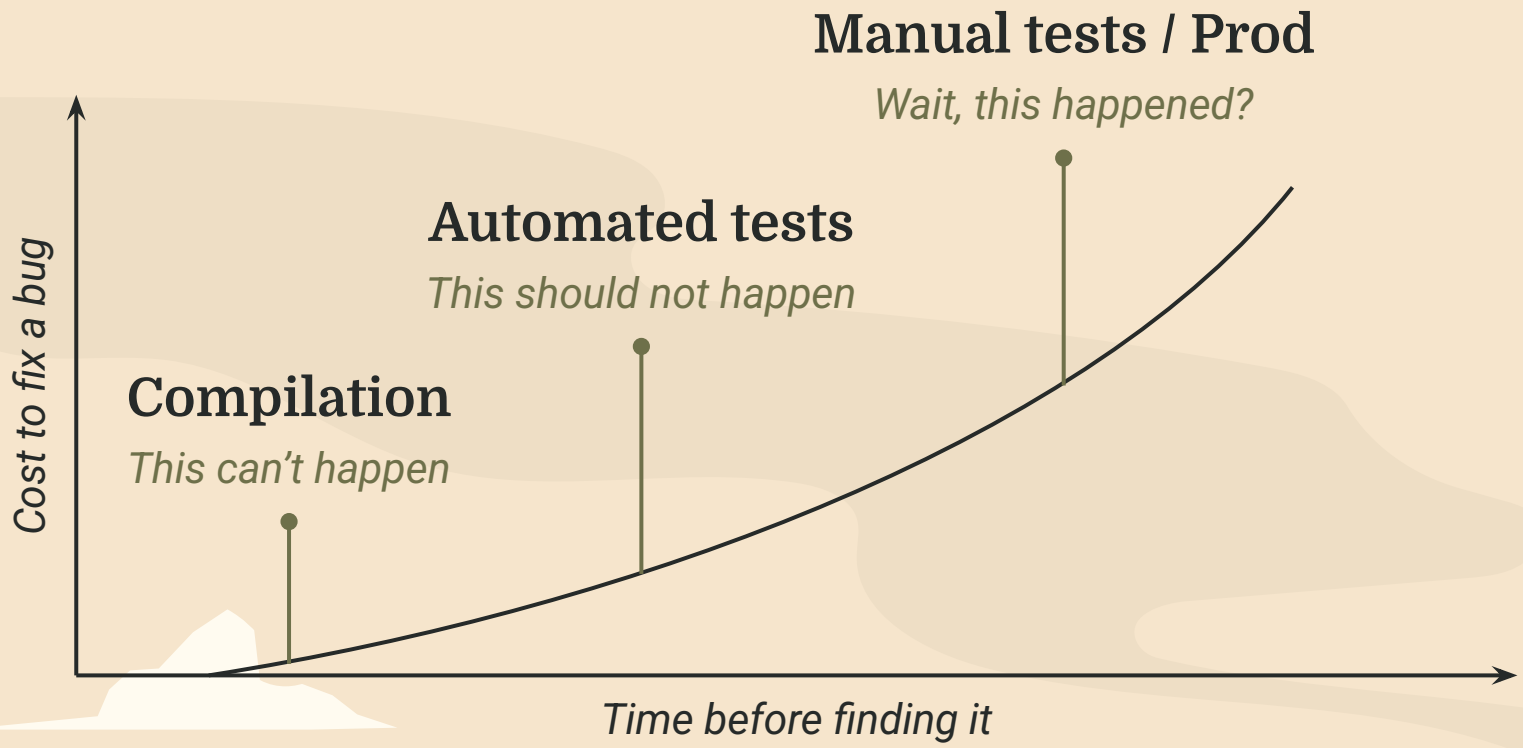
3

## Refactoring

Easy to write tests
that fail when you
refactor the code


Hope
is not a
strategy

# The faster you catch it, the better



# Our battle plan: 2 spells

## Discriminated Unions

Make impossible
states unrepresentable

## Brand Types

Fight against
Primitive Obsession

# 01

# Too many booleans

```typescript
type UserSubscription = {
    isSubscribed: boolean
    hasCanceledSubscription: boolean
    isTrialPeriod: boolean
    subscriptionEndDate: Date | null
    trialEndDate?: Date
}
```

5 booleans (ON or OFF)

$2^5 = 32$ representable states

**How many should be *impossible*?**

# You won’t test them all


Hope
is not a
strategy

# Enums are not enough in TypeScript

```
enum UserSubscription {
    NewCustomer = "new customer",
    InTrialPeriod = "in trial period",
    Subscribed = "subscribed",
    SubscriptionCanceled = "canceled",
    SubscriptionExpired = "expired",
}
```

TypeScript Enums can only be **`string`** or **`number`**

The TS equivalent of functional Enums are Discriminated Unions

# Introduce Discriminated Unions

```
type UserSubscription =
    | { status: "new customer" }
    | { status: "in trial period"; trialEndDate: Date }
    | { status: "subscribed"; subscriptionEndDate: Date }
    | { status: "subscription canceled"; subscriptionEndDate: Date }
    | { status: "subscription expired"; subscriptionEndDate: Date }
```

Only list the possible states of your Domain

Can’t represent impossible ones = fewer tests to write

# What client code looks like

```
function getExpirationWarnings(subscription: UserSubscription): string[] {
    if (
        subscription.status === "in trial period"
        && isWithinAWeek(subscription.trialEndDate)
    ) {
        return ["Your trial period is ending soon"];
    }

 // …
 return [];
}
```

# Move defensive logic to TypeScript

```
type UserSubscription =
      | NewCustomer
      | UserInTrial
      | UserSubscribed
      | UserSubscriptionCanceled
      | UserSubscriptionExpired;

declare function cancel(subscription: UserSubscribed): UserSubscriptionCanceled
```

# Pros & cons of Discriminated Unions

## Reduce states count

Fewer possible scenarios is easier to manage at scale

## Native TypeScript

You don’t need a type library to introduce the pattern

## Express your Domain

It helps you pause and think about business logic

## Need to teach it

You need to change the way your team do things

# 02

# Primitive Obsession

```typescript
export type Employee = {
    id: string
    fullName: string
    skills: string[]
    privateInfo: {
        referrals: number
        phone: string
        dateOfBirth: string
        monthlySalary: number
    }
}
```

Can you see more impossible states?

# `string` and `number` are often too loose

# 3 pernicious problems with Primitives

## Confusion

TypeScript won't prevent John to pass `fullName` instead of `id` by mistake

## Security

The surface area of potential injections or bad data is huge

## Duplication

You need to check for the same conditions everywhere in the code

*(and it's not consistent)*

# Type Aliases can only document

```
type Phone = string
type Money = number

type EmployeeInfo = {
 age: number
 phone: Phone
 dateOfBirth: string
 monthlySalary: Money
}
```

It can improve readability,
but it won't be effective

# Introduce Brand Types

```
type Phone = string & { readonly type: 'Phone' }
```

**Phone** can be assigned to a `string`

But only **Phone** can be assigned to a **Phone**

# Define a Brand<Type, Name>

```
// unique symbol = can't conflict with other properties
declare const __brand__: unique symbol

type Brand<BaseType, BrandName> = BaseType & {
    readonly [__brand__]: BrandName
}
```

# What client code looks like

```
type Phone = Brand<string, 'Phone'>

declare function addPhone(employee: Employee, phone: Phone): Employee
```

**Confusion** is gone, a valid type is enforced

“But how do I create a valid Phone?”

# A single source of truth


```
declare function validatePhone(phone: string): Phone
```

TypeScript will push your colleagues to use this

*Pit of Success*

It solves the **Duplication** and helps with **Security**

# What validation code looks like

```
function validatePhone(phone: string): Phone {
 if (phone.length < 10) {
   throw new Error("Phone number is too short");
 }
 // … more rules

 return phone as Phone;
}
```

# Pros & cons of Brand Types

## Pit of Success

Helps others do the right thing when in a rush

## Redefine base operations

`Quantity + Quantity` works, but returns a `number`

## Remove Duplication

When you get the type, you *know* it was validated

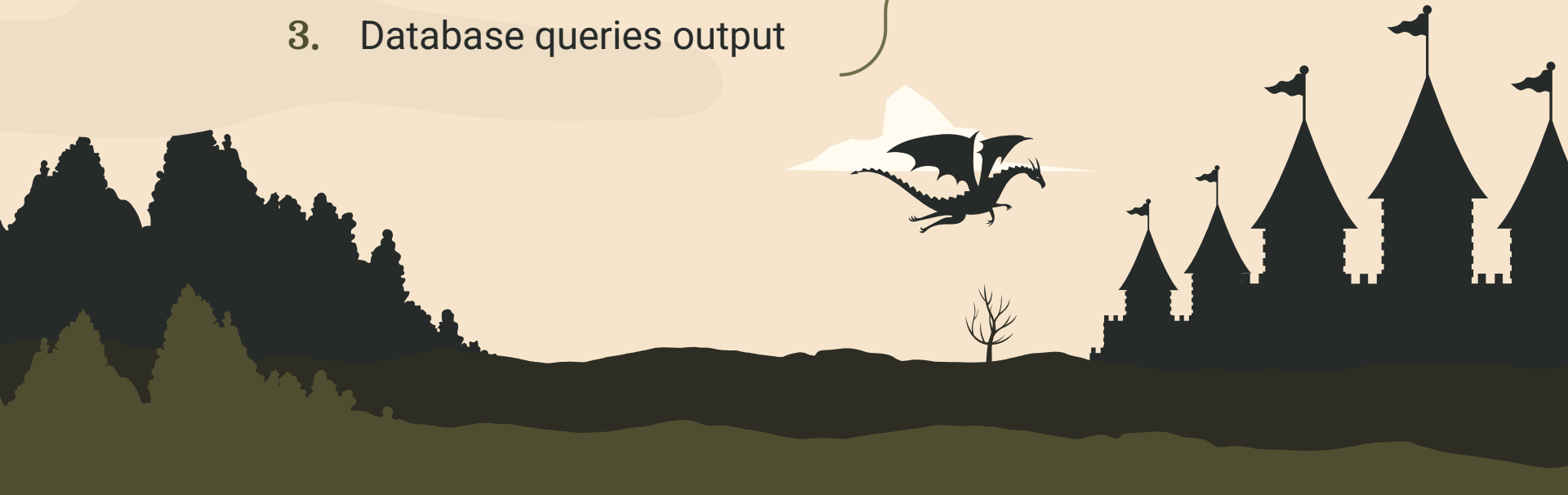
## Need to teach it

You need to change the way your team do things

# Beware of external data

1. User Inputs
2. Third-party API responses
3. Database queries output

**unknown**



# Delegate this work to Zod

```
const Phone = z.string()
    .brand("Phone")
    .refine(validatePhone)

type Phone = z.infer<typeof Phone>
```

```
const validPhone = Phone.parse(incomingData)
```

zod.dev

You are ready for battle!

# Discriminated Unions & Brand Types

## Fewer states

Impossible scenarios made unrepresentable

## Pit of Success

Easy to write maintainable code

## Faster than tests

Catch bugs at compilation time

## No duplication

Express your Domain in a single source of truth

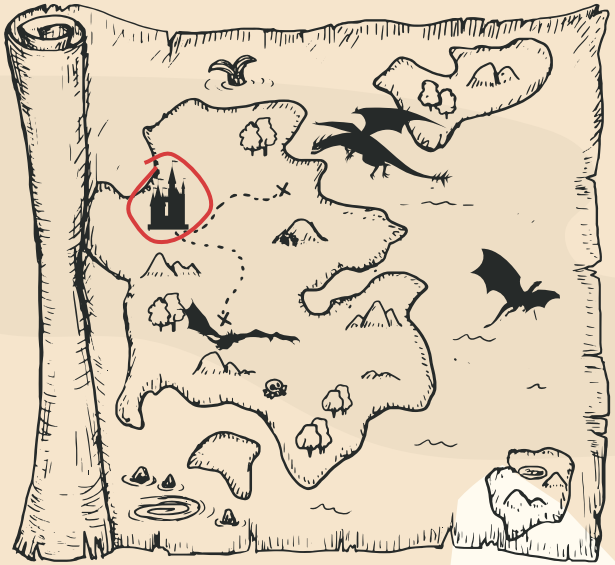
## Native TS code

You don't need an external lib *(but use Zod)*

## Secure by Design

Bonus side-effect of expressing your Domain

# The journey is only beginning...



## Pattern Matching

ts-pattern — to make client code easier to read and compose

## State Machines

xstate — to manage transitions between possible states

# References, for curious minds


1. ts-reset - Matt Pocock
2. Domain Modeling Made Functional - Scott Wlaschin
3. Secure by Design - Dan Bergh Johnsson, Daniel Deogun, Daniel Sawano
4. Software Engineering at Google - Titus Winters, Tom Manshreck, Hyrum K. Wright
5. Domain Data Modeling using TypeScript Aliases, Brand Types and Value Objects - Egghead course by Tomasz Ducin

# Farewell

Let’s keep in touch!

Nicolas Carlo

@nicoespeon