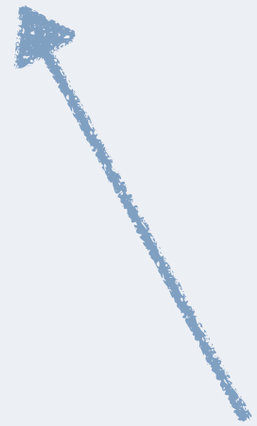
Redonner du corps au métier avec la programmation fonctionnelle

« Making illegal states unrepresentable »

Du code métier spécifique

Expert métier

Développeur



Modèle du domaine

Ubiquitous language

Exprimé dans le code

```
open Voteer.Core.Common

type OpenPoll =
    PendingPoll → DomainResponse<OpenedPoll>

type CreatePoll =
    CreationForm → DomainResponse<PendingPoll>
```

Stratégie

Versus

Tactique

Agrégat

Entité

Architecture !

Objet valeur

Service

Entrepôt

Options d’architecture

CRUD	«DDD» Tactique	API Rest	Message broker
CQRS	Event sourcing	gRPC…	Protocol
Fonctionnel	Objet	Dynamique	Static

Options d’architecture

CRUD	«DDD» Tactique	API Rest	Message broker
CQRS	Event sourcing	gRPC…	Protocol
Fonctionnel	Objet	Dynamique	Static

Quels problèmes souhaite-t-on résoudre ?

```
class Patient {
  public int IdPatient;
  public string Prénom;
  public string NomDeNaissance;
  public string NomDUsage;

  public string AdresseEmail;
  public bool EmailEstVerifié;
}
```

- Quelles valeurs sont optionnelles ?
- Quelle est la logique métier ?
- Quel est l’état valide de cet objet ?
- Utilisation exclusive des types primitifs 😕

```
type Patient = {
  IdPatient: int
  Prénom: string
  NomDeNaissance: string
  NomDUsage: string

  AdresseEmail: string
  EmailEstVérifié: bool
}
```

```
type Patient = {
  IdPatient: int
  Prénom: string
  NomDeNaissance: string
  NomDUsage: string option

  AdresseEmail: string
  EmailEstVérifié: bool
}
```

Les types somme

```
type Option = ?
```

Les types somme

```
type Option =
| SomeString of string
| None
```

Les types somme

```
type Option =
| SomeString of string
| None
```

```
type OptionInt =
| SomeInt of int
| None
```

```
...
```

Les types somme

```
type Option<'T> =
| Some of 'T
| None
```

```
let defaultValue value optional =
   match optional with
   | Some x → x
   | None → value

let monOption = Some "Okiwi"
printfn "Hello %s" (defaultValue "World" monOption)
```

```
type Patient = {
  IdPatient: int
  Prénom: string
  NomDeNaissance: string
  NomDUsage: string option

  AdresseEmail: string
  EmailEstVérifié: bool
}
```

```fsharp
type AdresseEmail =
| Vérifiée of string
| NonVérifiée of string * Guid

type Patient = {
  IdPrénom: int
  Prénom: string
  NomDeNaissance: string
  NomDUsage: string option

  AdresseEmail: AdresseEmail
}
```

```
type AdresseEmail = ??

let vérifieAdresse (??) token: ?? option =
  ???


type Patient = {
  ...

  AdresseEmail: AdresseEmail
}
```

```fsharp
type AdresseVérifiée = AdresseVérifiée of string
type AdresseNonVérifiée = AdresseNonVérifiée of string * Guid

type AdresseEmail =
| Vérifiée of AdresseVérifiée
| NonVérifiée of AdresseNonVérifiée
```

```fsharp
let vérifieAdresse (AdresseNonVérifiée (v, t)) token:
AdresseVérifiée option =
  if t = token then Some (AdresseVérifiée v) else None

type Patient = {
  ...

  AdresseEmail: AdresseEmail
}
```

```fsharp
module AdresseEmail =
  type AdresseVérifiée = AdresseVérifiée of string
  type AdresseNonVérifiée = AdresseNonVérifiée of string * Guid

  type T =
  | Vérifiée of AdresseVérifiée
  | NonVérifiée of AdresseNonVérifiée

  let crée (v: string): AdresseNonVérifiée option =
    if v.Contains("@")
      then Some (AdresseNonVérifiée (v, Guid.NewGuid()))
      else None

  let vérifieAdresse (AdresseNonVérifiée (v, t)) token:
AdresseVérifiée option =
    if t = token then Some (AdresseVérifiée v) else None

type Patient = {
  ...

  AdresseEmail: AdresseEmail.T
}
```

```fsharp
type Patient = {
    IdPatient: int
    Prénom: string
    NomDeNaissance: string
    NomDUsage: string option

    AdresseEmail: AdresseEmail.T
}
```

```fsharp
type Patient = {
    IdPatient: int
    Prénom: string
    NomDeNaissance: string
    NomDUsage: string option

    AdresseEmail: AdresseEmail.T
}

type RécupérePatient = int -> Patient
```

```fsharp
type IdPatient = IdPatient of int

type Patient = {
  IdPatient: IdPatient
  Prénom: string
  NomDeNaissance: string
  NomDUsage: string option

  AdresseEmail: AdresseEmail.T
}

type RécupérePatient = IdPatient -> Patient
```

Récapitulatif

- Enrober les types primitifs dans des types porteurs de sémantique
- Utiliser les types algébriques pour modéliser les contraintes métiers
- Ne pas permettre la création d’objets dans un état invalide
- `null` n’est pas notre ami, et est à proscrire dans la couche métier — la validation peut être faite dans les couche d’I/O

Bénéfices

- Réduction de la boucle de feedback grâce au compilateur
- Utilisation du *type checker* comme d'un assistant pour la vérification du code — « TDD shows presence of errors, Type Driven Development shows absence of errors »
- Effet de documentation par l'augmentation de l'expressivité du code — « Code as documentation »
- Augmentation de la confiance

« Re: FP & TDD. More important than tests vs types is the principle of double checking. If you say something twice in independently derived ways, you're more likely to be correct than if you just say it once. Tests are a form of double checking. So are types. »

-Kent Beck

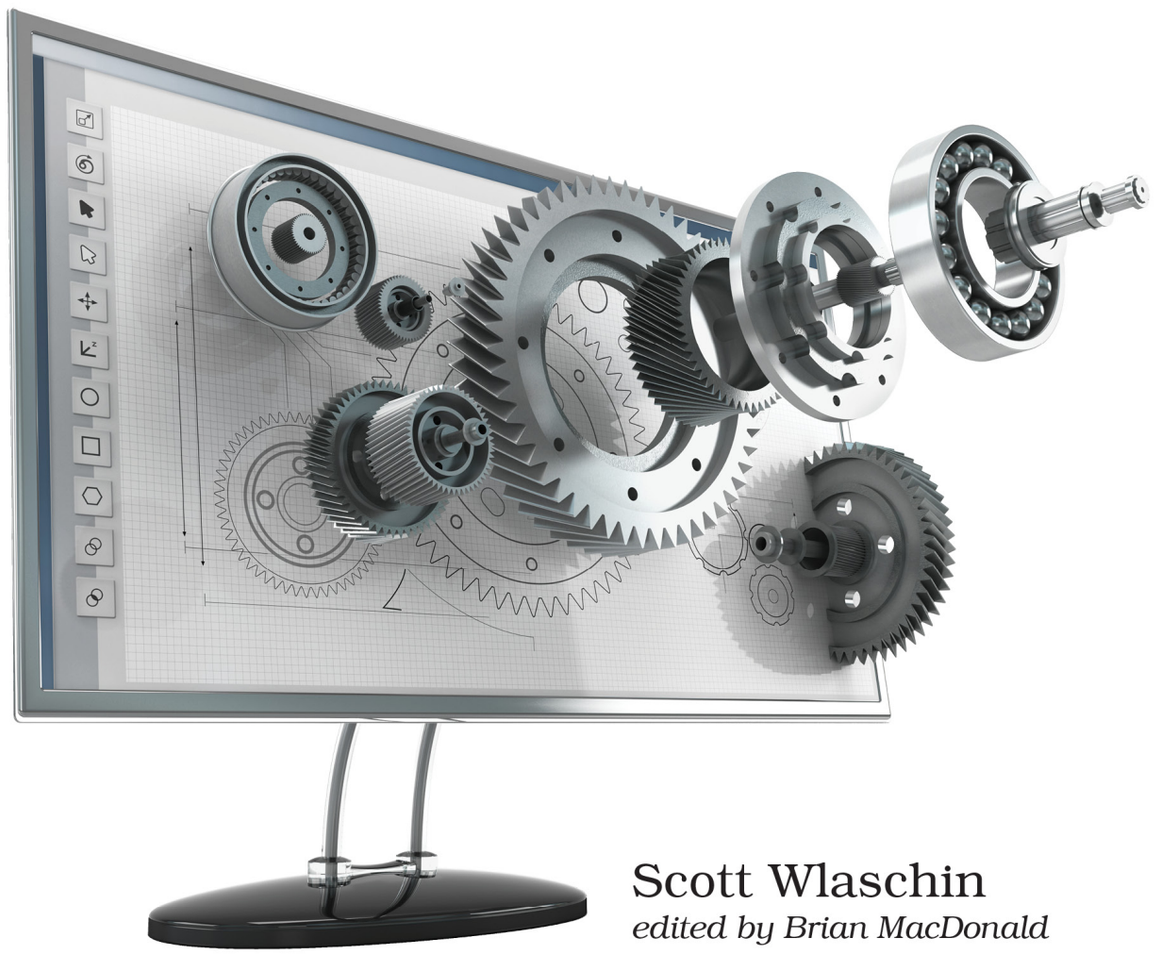
Où puis-je tester tout ça ???

- Forcément, en Haskell, Elm, OCaml, Reason, …
- En F# (qui est déjà présent dans toutes les créméries .Net !)
- Mais aussi en Scala, Kotlin, TypeScript, Rust, Swift, et sûrement plus dans le futur !

The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#



Scott Wlaschin

edited by Brian MacDonald

Merci !

Josselin Auguste aka. @josselinauguste