codecentric

From Vision to Code

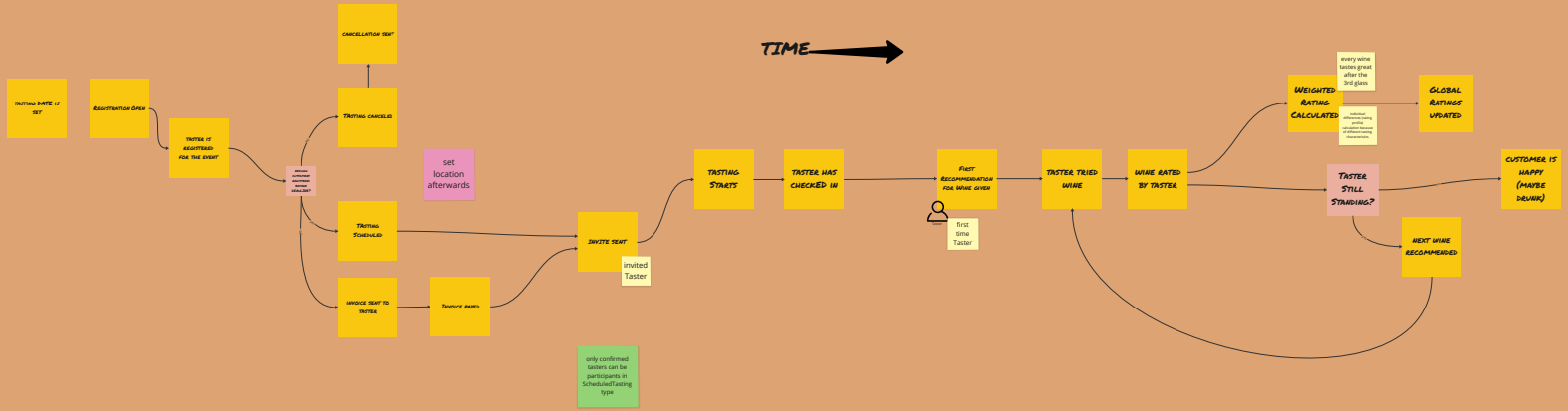
Domain Modelling Live

Marco
as the Dev

Ferdi
as the PO

Algorithmic Aromas Tasting Co.

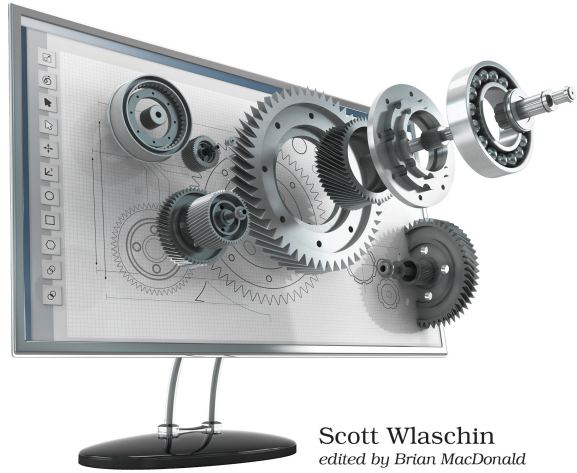

Sancerre
2010
LUCIEN CROCHET


TIME
TASTING DATE IS SET
REGISTRATION OPEN
TASTER IS REGISTERED FOR THE EVENT
CANCELLATION SENT
TASTING CANCELED
TASTING SCHEDULED
INVOICE SENT TO TASTER
INVOICE PAID
set location afterwards
INVITE SENT
invited Taster
only confirmed tasters can be participants in ScheduledTasting type
TASTING STARTS
TASTER HAS CHECKED IN
FIRST RECOMMENDATION FOR WINE GIVEN
first time Taster
TASTER TRIED WINE
WINE RATED BY TASTER
WEIGHTED RATING CALCULATED
every wine tastes great after the 3rd glass
GLOBAL RATINGS UPDATED
TASTER STILL STANDING?
CUSTOMER IS HAPPY (MAYBE DRUNK)
NEXT WINE RECOMMENDED

The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#



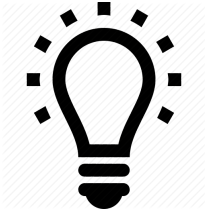
Scott Wlaschin

edited by Brian MacDonald

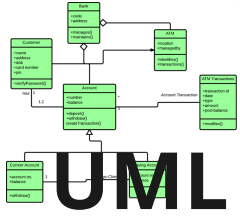
Business Development Gap

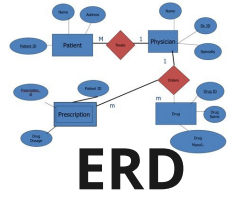
Gap

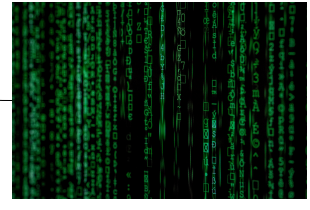
Classical Modelling Tools



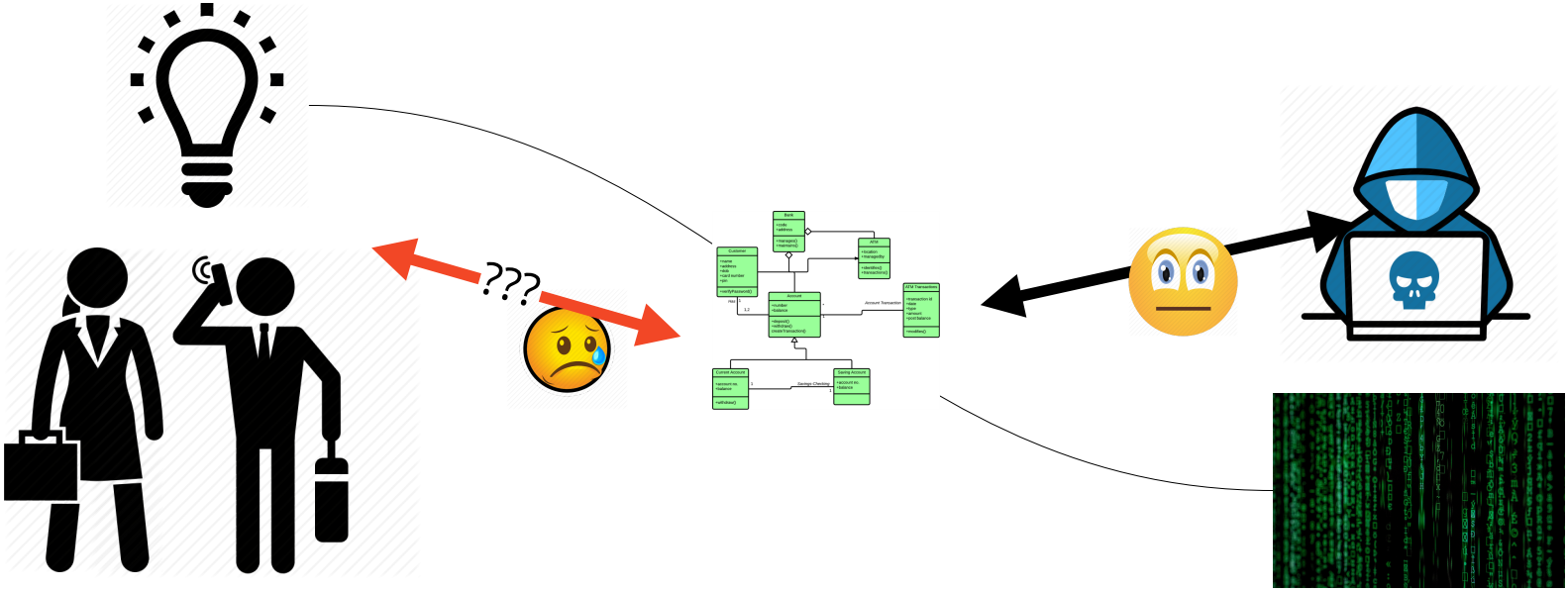
Gap



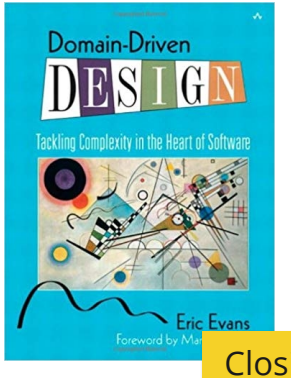




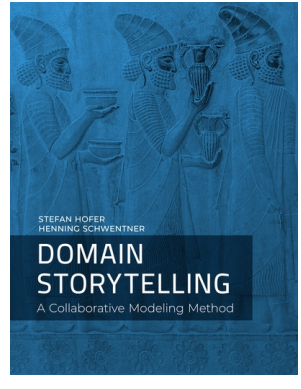
Classical Modelling Tools

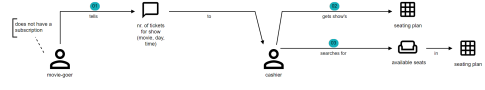


Collaborative Modelling Tools

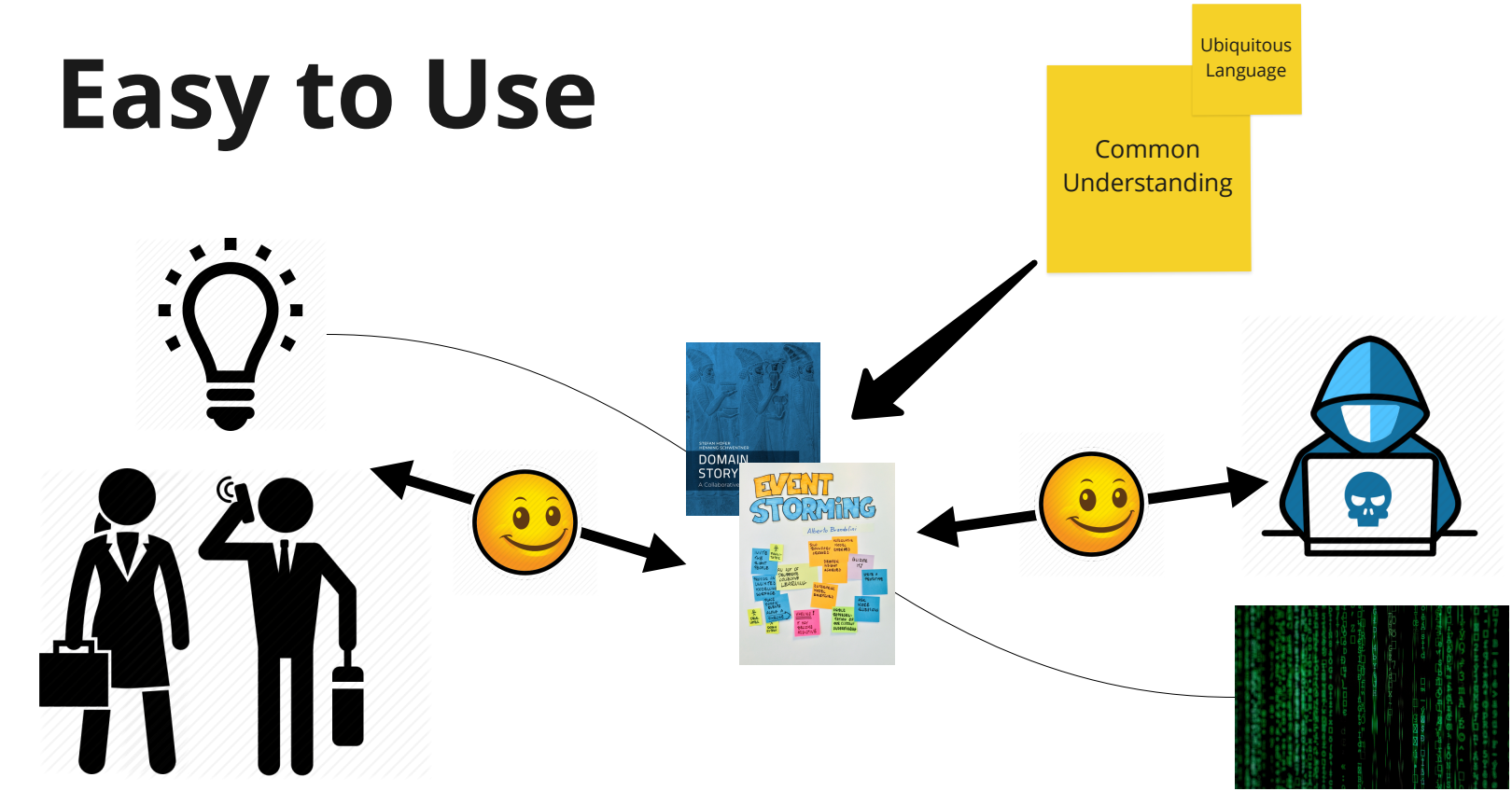


Closing the Gap

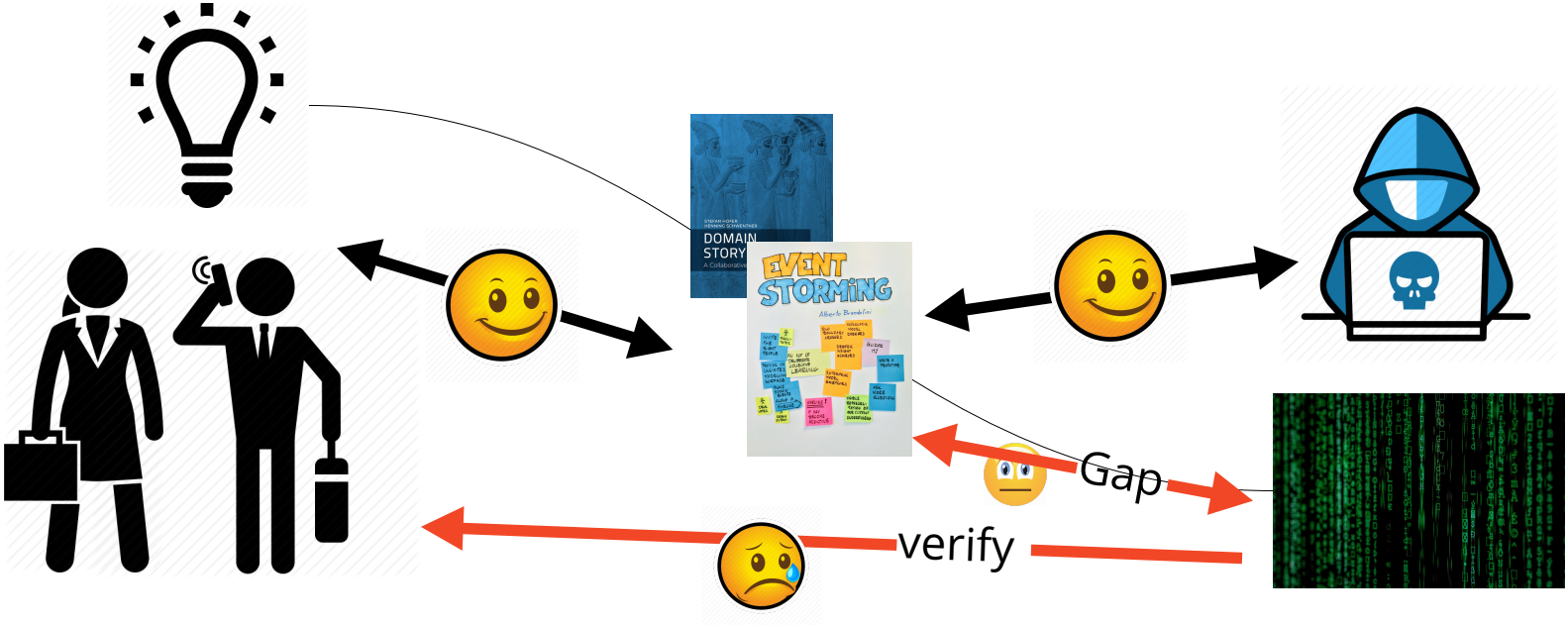




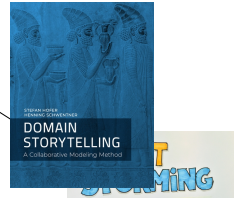
Easy to Use

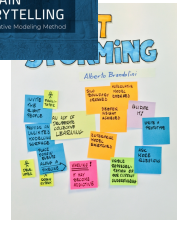

Ubiquitous Language
Common Understanding
DOMAIN STORY
EVENT STORMING

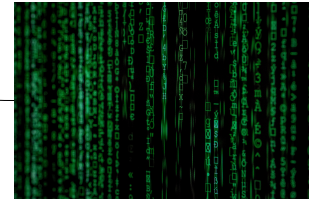
Modelling Tools



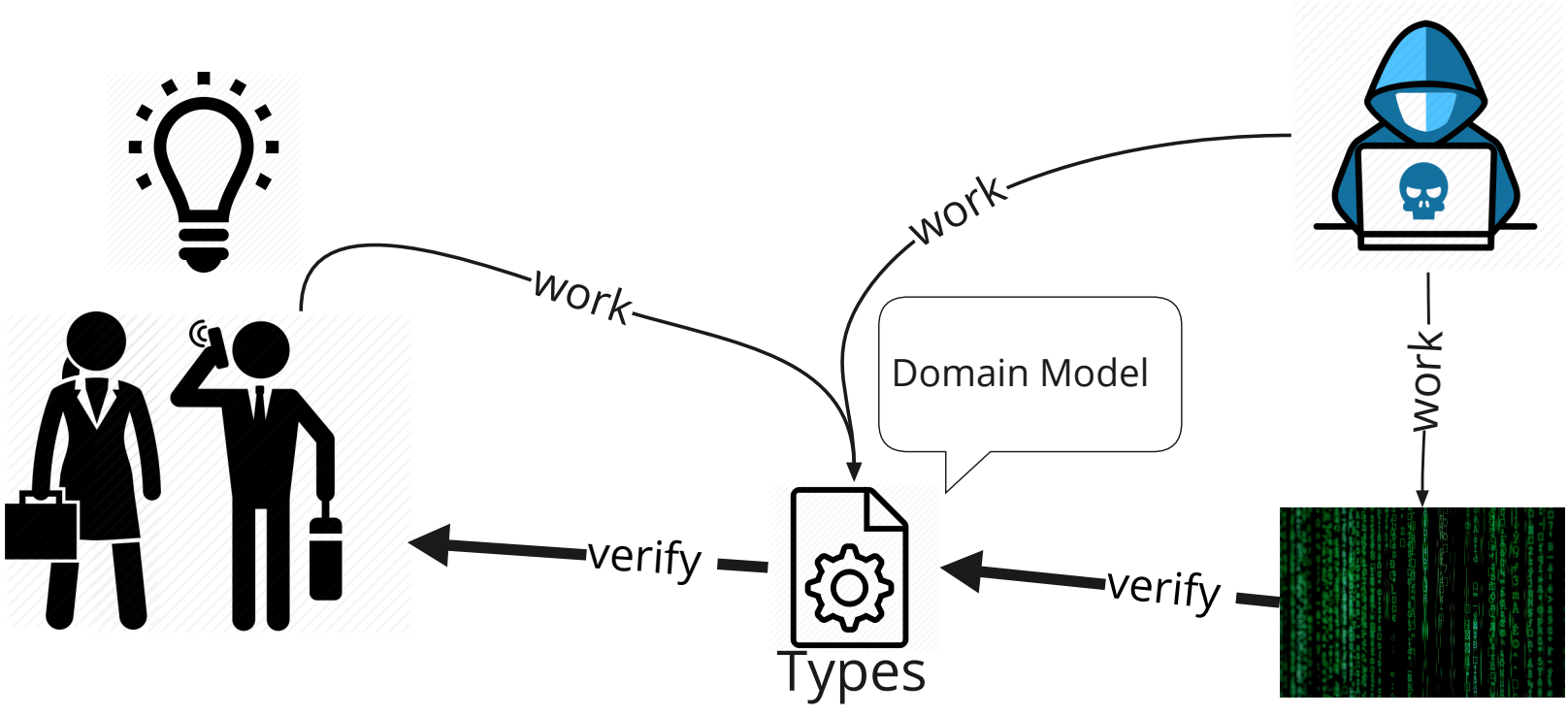
(Functional) Domain Modelling







Domain Modelling with Types



Goals

- common understanding
- explicit, specific
- starting point for implementation


Let's Start

Wrap Up

Domain Modelling

TLDR;

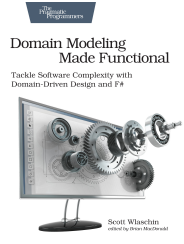
What is it?

A way of working with business people directly on the code ... without having them running away screaming

What happens before?

Advantages?

* it's real!
* not just a fancy pic
* everyone can read it



How to?

Use just the TYPES to model a domain

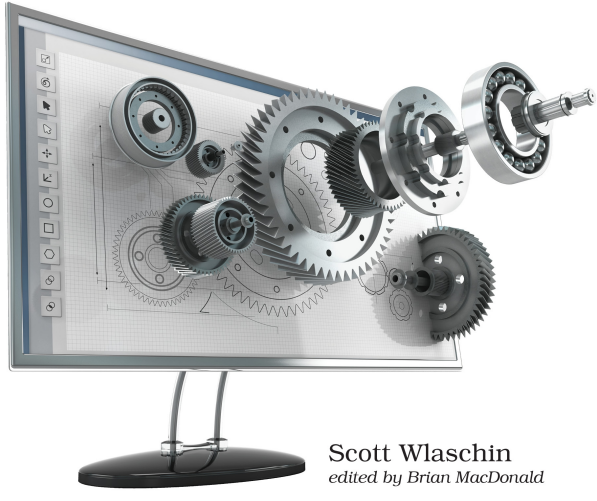
You need a really STRONG Type System

Encode your Domain as Types

The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

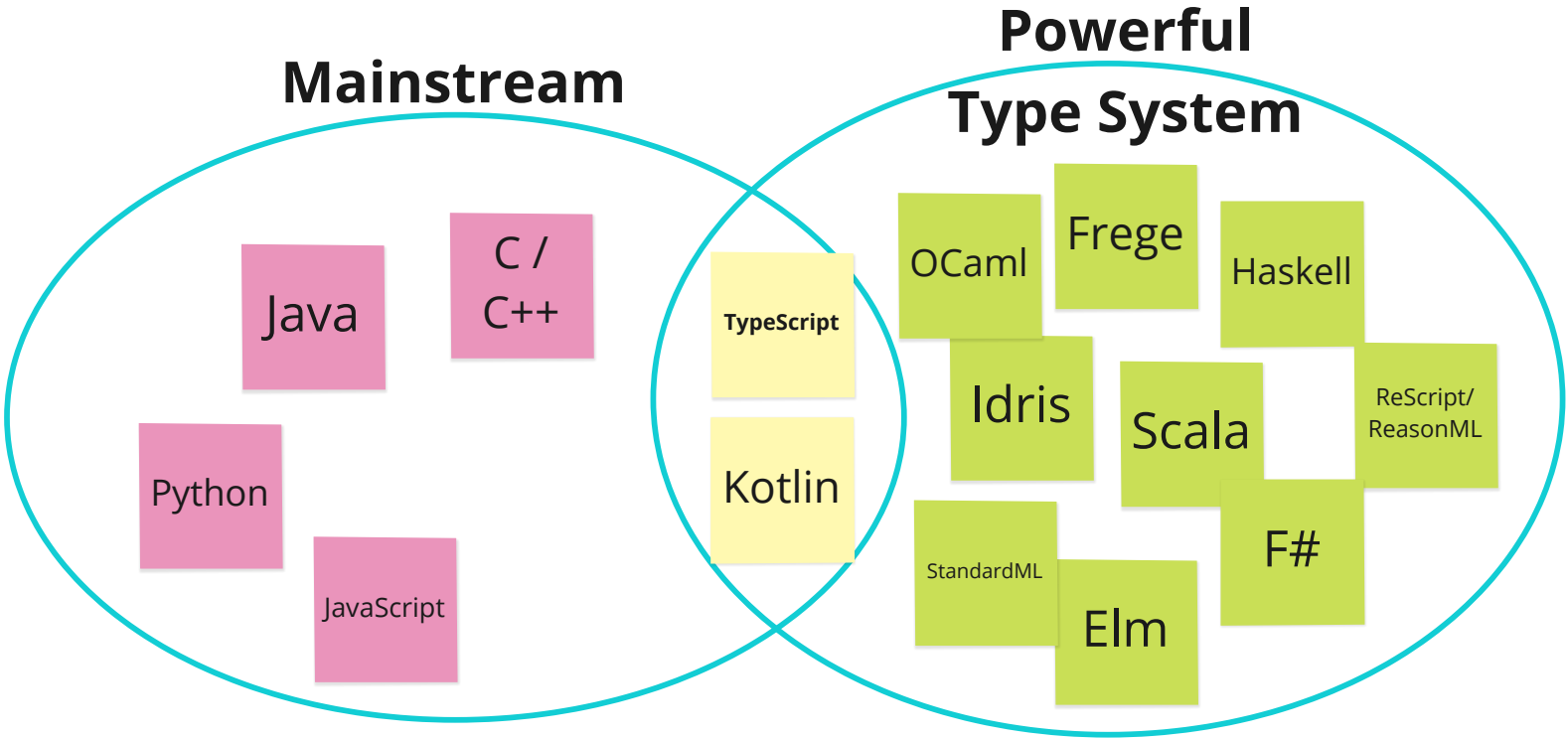


Scott Wlaschin
edited by Brian MacDonald

```
type UnitQuantity = UnitQuantity of int

type KilogramQuantity = KilogramQuantity of decimal

type OrderQuantity =
  | Unit of UnitQuantity
  | Kilos of KilogramQuantity

type Order = {
  Id : OrderId // id for entity
  CustomerId : CustomerId // customer reference
  ShippingAddress : ShippingAddress
  BillingAddress : BillingAddress
  OrderLines : OrderLine list
  AmountToBill : BillingAmount
}
and OrderLine = {
  Id : OrderLineId // id for entity
  OrderId : OrderId
  ProductCode : ProductCode
  OrderQuantity : OrderQuantity
  Price : Price
}
```

F#

Languages


Mainstream
Powerful
Type System
Java
C /
C++
Python
JavaScript
TypeScript
Kotlin
OCaml
Frege
Haskell
Idris
Scala
ReScript/
ReasonML
StandardML
F#
Elm

Patterns

Explicit Options

```
type Wine = { ...;  onSale: boolean;  };
```

```
type Wine = { ...;  onSale: OnSale; };
type OnSale =
  | "not on sale"
  | "stock sale"
```

Invariants enforcement

```
type Wine = { ...;  onSale: boolean;  campaignStart: Date;  campaignEnd: Date;  stock: number; };
```

Union Type

```
type Wine = { ...;  onSale: OnSale; };
type OnSale =
  | { saleType: "not on sale" }
  | { saleType: "stock sale"; stock: number }
  | { saleType: "campaign"; from: Date; to: Date };
```

Invariants enforcement

```
const myWine = {
   ...;
   onSale: "true";
   campaignStart: 2024-01-01;
   campaignEnd: null;
   stock: 10;
};
```

Invalid, no Compile Error

Compile Error

```
const saleValue = { saleType: "stock sale"; stock: 10 , from: "2023-01-01"}
```

```
const saleValue = { saleType: "no sale"; from: "2023-01-01", end: "2023-01-02"}
```

```
const saleValue = { saleType: "stock sale"; stock: 10 }
```

valid

Explicit Data

number → 1 | 2 | 3| 4 | 5

Wine[] → [Wine, Wine, Wine]

```
type RecommendWine2 = (wineCellar: WineCellar) => WineRecommendation;
type WineRecommendation = [Wine, Wine, Wine];
```

Avoid Side Effects

```
type RateWine = (rating: WineRating, profile: TasterRatingProfile)
                => void
```

```
type RateWine = (rating: WineRating, profile: TasterRatingProfile)
                => TasterRatingProfile
```

Expressive Workflow

```
type RateWine = (..., profile: TasterRatingProfile) => TasterRatingProfile;

type RecommendNextWine = (..., profile: TasterRatingProfile) =>...
```

```
type RateWine = (..., profile: TasterRatingProfile) => UpdatedTasterRatingProfile;

type RecommendNextWine = (..., profile: UpdatedTasterRatingProfile) =>...
```

Outside-In

Start with the Result

Behaviour first

Details later

```
type RecommendWine = (wines: Wine[]) => Wine;
```

Omit Details

```
type Wine = unknown;
```


```
type Wine = {
 color: WineColor;
  grape: Grape;
  name: Name;
  year: Year;
  country: Country;
};
```

Explicit Behaviour

```
type RecommendWine =
 (wineCellar: WineCellar, options: { profile?: TasterRatingProfile }) => WineRecommendation;
```


```
type RecommendNextWine = (wineCellar: WineCellar, profile: TasterRatingProfile) => ...;
type RecommendFirstWine = (wineCellar: WineCellar) => ..
```

Use Business Terms

Wine[] → WineCellar

Rating → StarRating

```
type Wine = {
  grape: string;
 ...
};
```

```
type Wine = {
  grape: Grape;
 ...
};
type Grape = string;
```

Try this at Home!

- Be Precise and Explicit
- Don't be too **fancy** with code
- Listen carefully to domain experts