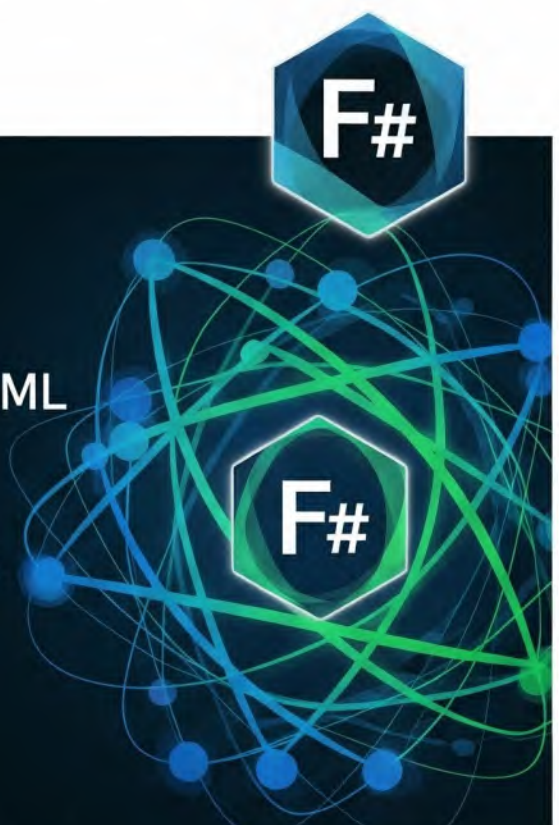
F# Programming: Professional

(Integrative-Generative AI Edition)



Contents

- Domain Modeling with F#
- Working with Data Science & ML
- Web Development in F#
- Testing & Quality
- Deployment & Best Practices

Student Price Book Center

คำนำ

การพัฒนาซอฟต์แวร์ในยุคปัจจุบันต้องการนักพัฒนาที่ไม่เพียงเชี่ยวชาญด้านโค้ดดิ้งเท่านั้น แต่ยังต้องเข้าใจเชิงลึกถึงแนวคิดการออกแบบ ระบบ และกระบวนการทำงานในโลกธุรกิจจริง **F#** เป็นหนึ่งในภาษาโปรแกรมเชิงฟังก์ชันที่ตอบโจทย์การพัฒนาแอปพลิเคชันที่ซับซ้อนและมีคุณภาพสูง ด้วยคุณสมบัติเด่นด้าน immutability, pattern matching, และ type system ที่เข้มงวด ทำให้นักพัฒนาสามารถสร้างโค้ดที่กระชับ อ่านง่าย และรองรับการขยายในอนาคต

หนังสือเล่มนี้ถูกออกแบบมาเพื่อช่วยให้นักพัฒนาระดับมืออาชีพสามารถนำ **F#** ไปประยุกต์ใช้ในหลากหลายด้าน ตั้งแต่การสร้าง **Domain Model** ที่ซับซ้อน การประมวลผลและวิเคราะห์ข้อมูลขนาดใหญ่ การพัฒนาเว็บแอปพลิเคชัน การทำงานด้านวิทยาศาสตร์ข้อมูลและ Machine Learning ไปจนถึงการทดสอบและการส่งมอบซอฟต์แวร์อย่างเป็นระบบ แต่ละบทถูกจัดทำให้ผู้อ่านได้เรียนรู้ทั้งหลักการเชิงทฤษฎีและตัวอย่างเชิงปฏิบัติ พร้อมแนะนำแนวทางที่เหมาะสมและ Best Practices ที่นักพัฒนาควรทราบ

บทที่ 18 จะพาผู้อ่านเจาะลึก **Domain Modeling with F#** ด้วยแนวคิด **Domain-Driven Design (DDD)** การใช้ **Discriminated Union (DU)** ในการกำหนด Business Rules และการประยุกต์ **Railway Oriented Programming (ROP)** เพื่อจัดการความสำเร็จและข้อผิดพลาดของระบบอย่างเป็นระบบ ผู้อ่านจะได้เข้าใจทั้งแนวคิดเชิงทฤษฎีและเทคนิคเชิงปฏิบัติในการสร้างแบบจำลองเชิงโดเมนที่สะท้อนความต้องการทางธุรกิจอย่างแท้จริง

ต่อด้วยบทที่ 19 ซึ่งเน้นการทำงานด้าน **Data Science และ Machine Learning** ด้วย F# การเชื่อมต่อกับ **ML.NET** การจัดการและประมวลผลข้อมูลขนาดใหญ่ รวมถึงการวิเคราะห์เชิงสถิติและการสร้างโมเดลเชิง Machine Learning โดยมีตัวอย่างบูรณาการที่ช่วยให้ผู้อ่านเข้าใจการประยุกต์ใช้ F# ในโลกของ Big Data และการวิเคราะห์ข้อมูลเชิงลึกได้อย่างครบถ้วน

บทที่ 20 มุ่งเน้น **Web Development in F#** โดยแนะนำการสร้างเว็บแอปพลิเคชันด้วย **Giraffe** และ **Saturn Framework** ซึ่งสนับสนุนแนวคิด functional-first บน ASP.NET Core พร้อมการพัฒนา **REST API** ที่มีความปลอดภัยและยืดหยุ่น การผสมผสานระหว่างโครงสร้างเชิงฟังก์ชันและสถาปัตยกรรมเว็บสมัยใหม่ช่วยให้นักพัฒนาสามารถสร้างเว็บแอปพลิเคชันที่มีประสิทธิภาพและบำรุงรักษาได้ง่าย

บทที่ 21 ครอบคลุมเรื่อง **Testing & Quality** ตั้งแต่ **Unit Testing** ด้วย xUnit และ Expecto, **Property-based Testing** ด้วย FsCheck ไปจนถึงแนวทาง **Test-Driven Development (TDD)** ผู้อ่านจะได้เรียนรู้วิธีการสร้างเทสต์ที่มั่นคง ตรวจสอบได้ และออกแบบโค้ดที่มีคุณภาพสูง เพื่อให้ซอฟต์แวร์สามารถทำงานได้อย่างเชื่อถือได้และลดความเสี่ยงจากข้อผิดพลาดในระยะยาว

บทสุดท้ายของเล่มคือบทที่ 22 เกี่ยวกับ **Deployment & Best Practices** ครอบคลุมแนวทางการทำ **CI/CD** บน .NET F#, การเฝ้าติดตามระบบด้วย **Logging & Monitoring** ผ่าน Serilog และ

NLog, การประยุกต์ **Clean Code และ SOLID Principles** ในมุม Functional Programming รวมถึงการ **Deploy API/Services ไปยัง Cloud** เช่น Azure, AWS และ Docker บทนี้ช่วยให้นักพัฒนาสามารถส่งมอบซอฟต์แวร์ได้อย่างมั่นใจ ปลอดภัย และมีคุณภาพสูง

ตลอดเล่ม หนังสือเน้นทั้ง **เชิงลึกและเชิงปฏิบัติ** เพื่อให้ผู้อ่านสามารถต่อยอดความรู้และสร้างซอฟต์แวร์คุณภาพสูงได้ด้วย F# ไม่ว่าจะเป็นการออกแบบระบบ การพัฒนาเว็บ การวิเคราะห์ข้อมูล หรือการส่งมอบระบบไปยังสภาพแวดล้อมจริง ด้วยโครงสร้างและตัวอย่างที่ครบถ้วน เล่มนี้จึงเป็นคู่มือระดับมืออาชีพที่เหมาะสำหรับนักพัฒนาที่ต้องการก้าวสู่การใช้งาน F# อย่างเต็มศักยภาพ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 18

Domain Modeling with F#

(Domain Modeling with F#)

เนื้อหา

- Domain Modeling with F#
- Domain Modeling with F# ให้ละเอียดเชิงลึก
- Domain-Driven Design (DDD) เชิงลึก
- การใช้ Discriminated Union (DU) กับ Business Rules
- Railway Oriented Programming (ROP) ใน F#

การพัฒนาซอฟต์แวร์ในยุคปัจจุบันเน้นมากกว่าการสร้างระบบที่ทำงานได้ แต่ยังต้องสะท้อนปัญหาและความซับซ้อนของโลกธุรกิจให้ถูกต้อง แนวคิด Domain-Driven Design (DDD) จึงเข้ามามีบทบาทสำคัญในการช่วยให้นักพัฒนาสามารถสร้างแบบจำลองเชิงโดเมนที่ตรงกับความต้องการจริงของผู้ใช้งาน และทำให้โค้ดที่ได้มีความชัดเจนและยืดหยุ่นต่อการเปลี่ยนแปลงในอนาคต

F# เป็นภาษาที่มีพื้นฐานเชิงฟังก์ชัน จึงเหมาะสมอย่างยิ่งสำหรับการประยุกต์ใช้กับ DDD เพราะโครงสร้างภาษามีความกระชับและส่งเสริมการเขียนโค้ดที่มุ่งเน้นการอธิบายข้อมูลและข้อกำหนดเชิงธุรกิจโดยตรง การพัฒนาแบบนี้ช่วยให้แบบจำลองสะท้อนแนวคิดของธุรกิจได้ดีกว่า ลดการซ่อนรายละเอียดที่ไม่จำเป็น และเพิ่มความสามารถในการสื่อสารระหว่างนักพัฒนากับผู้เชี่ยวชาญทางธุรกิจ

หนึ่งในคุณสมบัติเด่นของ F# คือ Discriminated Union (DU) ซึ่งช่วยให้นักพัฒนาสามารถนิยามสถานการณ์และกรณีต่าง ๆ ได้อย่างเป็นระบบ คุณสมบัตินี้เหมาะอย่างยิ่งในการกำหนด Business Rules ที่มักจะมีหลายเงื่อนไขและผลลัพธ์ที่แตกต่างกัน ตัวอย่างเช่น สถานะของคำสั่งซื้อหรือการตรวจสอบความถูกต้องของข้อมูล ซึ่งเมื่อใช้ DU จะช่วยป้องกันการเขียนโค้ดที่ไม่ตรงตามตรรกะที่ต้องการ

นอกจากนั้น แนวคิด Railway Oriented Programming (ROP) ยังเป็นอีกหนึ่งเครื่องมือที่สอดคล้องกับ DDD และ DU โดย ROP เปรียบโครงสร้างของโปรแกรมเหมือนรางรถไฟสองเส้น เส้นหนึ่งแทนความสำเร็จ อีกเส้นหนึ่งแทนข้อผิดพลาด การเขียนโค้ดในลักษณะนี้ช่วยให้นักพัฒนารับมือกับข้อผิดพลาดได้อย่างมีระบบ โดยไม่ทำให้โครงสร้างซับซ้อนหรือยากต่อการติดตาม

การผสาน DDD, DU และ ROP เข้าด้วยกันทำให้การออกแบบระบบมีความชัดเจนและสื่อสารได้ง่าย ทั้งในมุมมองของการเขียนโค้ดและการอธิบายกับผู้ใช้งานหรือผู้เชี่ยวชาญธุรกิจ ทุกเงื่อนไขและ

กฎเกณฑ์ถูกแสดงออกมาในรูปแบบที่เป็นธรรมชาติและตรวจสอบได้ง่าย ซึ่งช่วยลดปัญหาการตีความที่ไม่ตรงกันระหว่างทีมงานและผู้ใช้ระบบ

เมื่อพิจารณาในเชิงคุณภาพของซอฟต์แวร์ แนวทางนี้ยังช่วยสนับสนุนการทดสอบและการบำรุงรักษาระบบในระยะยาว เนื่องจากแบบจำลองที่ได้จะมีความถูกต้องเชิงตรรกะตั้งแต่การออกแบบ การใช้ DU และ ROP ยังช่วยลดข้อผิดพลาดที่อาจเกิดขึ้นในเชิงการทำงานจริง ทำให้ซอฟต์แวร์มีความน่าเชื่อถือและยืดหยุ่นต่อการพัฒนาในอนาคต

บทนี้จะพาผู้อ่านไปทำความเข้าใจการสร้าง Domain Model ด้วย F# อย่างเป็นขั้นตอน ตั้งแต่หลักการของ Domain-Driven Design ไปจนถึงการใช้ Discriminated Union เพื่อกำหนด Business Rules และปิดท้ายด้วยการประยุกต์ Railway Oriented Programming เพื่อรองรับทั้งความสำเร็จและความล้มเหลวได้อย่างสง่างาม การบูรณาการแนวคิดเหล่านี้จะช่วยให้ผู้อ่านเข้าใจการสร้างซอฟต์แวร์ที่มีคุณภาพทั้งในเชิงโครงสร้างและเชิงธุรกิจ

Domain Modeling with F#

- Domain-Driven Design (DDD)
- การใช้ Discriminated Union กับ Business Rules
- Railway Oriented Programming

ต่อไปนี้คือ “บทที่ 18: Domain Modeling with F#” แบบลงลึก ครอบคลุมแนวคิด DDD, การใช้ Discriminated Union (DU) กับกฎธุรกิจ, และ Railway Oriented Programming (ROP) พร้อมตัวอย่างโค้ด F# แบบครบยกโมดูล นำไปใช้จริงได้ทันทีใน .fsx หรือโปรเจกต์ F# ใด ๆ

1) ภาพรวม Domain-Driven Design (DDD)

เป้าหมาย

- ทำให้แบบจำลองซอฟต์แวร์ **สอดคล้องกับโดเมนจริง** (Ubiquitous Language)
- แตกปัญหาใหญ่เป็นส่วน ๆ ที่เป็น **Bounded Context** ชัดเจน
- วางขอบเขต **ธุรกรรมและอินวาเรียนต์** ด้วย **Aggregate** ที่มี **Aggregate Root**
- แยกหน้าที่: **Entity** (มีอัตลักษณ์), **Value Object** (ไม่มีอัตลักษณ์, เปรียบเทียบด้วยค่า), **Domain Service**, **Application Service**, **Repository**, **Factory/Policy**
- ใช้ **Event** เพื่อสื่อสารการเปลี่ยนสถานะ/เหตุการณ์ในโดเมน

กลยุทธ์สำคัญ

- **Strategic DDD**: แบ่ง Bounded Context, ทำ Context Map (Upstream/Downstream, Anti-Corruption Layer)
- **Tactical DDD**: Entity/Value Object/Aggregate/Repository/Domain Event/Policy

หลักการออกแบบที่สัมพันธ์กับ F#

- **Immutability เป็นค่าเริ่มต้น** ช่วยลดบั๊กเชิงสถานะ

- **ชนิดข้อมูลที่ชัดเจน** (record, DU) ใช้ “make illegal states unrepresentable”
- **Pattern Matching** เพื่อบังคับเส้นทางกฎธุรกิจ (state machine)
- **Smart Constructors** บังคับอินวาเรียนต์ ณ จุดสร้าง

---

2) Map DDD ⟶ F# อย่างเป็นระบบ

แนวคิด DDD	F# Mapping
Value Object	type X = private X of ... + โมดูลสร้างแบบตรวจสอบ
Entity	record + Id ที่ไม่ซ้ำ (เช่น GUID/ULID)
Aggregate	DU ตาม “สถานะ” + record รายละเอียดของสถานะแต่ละแบบ
Domain Rules	Pattern matching + ฟังก์ชันเพียว
Repository	อินเตอร์เฟซ/เรคอร์ดของฟังก์ชัน get/save (ใช้ Async/Task)
Domain Events	DU ของเหตุการณ์
Invariants	Smart constructors + transition functions ที่คืน Result

เทคนิคเสริม:

- Option<'T> สำหรับ field ที่อาจไม่มี
- Result<'T,'E> สำหรับเส้นทางสำเร็จ/ล้มเหลว
- Units of Measure/Single-case DU เพื่อ “พิมพ์แข็ง” ป้องกันสลับหน่วย/ชนิด
- แยกโมดูลย่อย: ValueObjects, Domain, Rules, Events, Repo, AppService

---

3) โดเมนตัวอย่าง: ออเดอร์อีคอมเมิร์ซ

สิ่งที่ต้องการ

- **Value Objects**: Email, Quantity>0, Money≥0, NonEmptyString
- **Aggregate**: Order ที่จำลองเป็น “สถานะ” ด้วย DU (Draft ⟶ PendingPayment ⟶ Paid ⟶ Shipped/Cancelled)
- **Rules**:
  - เพิ่มสินค้าได้เฉพาะตอน Draft
  - Submit ต้องมีรายการ ≥ 1 และยอดรวม > 0
  - ชำระเงินได้เฉพาะ PendingPayment
  - ส่งของได้เฉพาะ Paid
- **Events**: OrderCreated, ItemAdded, OrderSubmitted, PaymentRecorded, OrderShipped, OrderCancelled

ด้านล่างเป็น **โค้ดเดียวครบ** (ย่อ/สื่อสารชัดเจน) ที่คุณสามารถวางในไฟล์ DomainModel.fsx แล้วรันด้วย dotnet fsi DomainModel.fsx:

```
// ================================================
// Chapter 18: Domain Modeling with F# (DDD + DU + ROP)
// ================================================

open System

// ---------- Common: NonEmptyList, Result/AsyncResult helpers ----------
type NonEmptyList<'a> = { Head:'a; Tail:'a list } with
  member x.ToList() = x.Head::x.Tail
module NonEmptyList =
  let create (x:'a) (xs:'a list) = { Head=x; Tail=xs }
  let ofList = function | h::t -> Some { Head=h; Tail=t } | [] -> None
  let append a b = a.ToList() @ b.ToList() |> ofList |> Option.get

type DomainError =
  | ValidationError of NonEmptyList<string>
  | RuleViolation of string
  | NotFound of string
  | ConcurrencyConflict
  | Unauthorized

// Result CE
type ResultBuilder() =
  member _.Bind(m,f) = Result.bind f m
  member _.Return x = Ok x
  member _.ReturnFrom m = m
  member _.Zero() = Ok ()
  member _.Combine(a,b) = Result.bind (fun _ -> b) a
  member _.Delay f = f()
let result = ResultBuilder()

module Result =
```

```
  let map2 f r1 r2 =
    match r1,r2 with
    | Ok a, Ok b -> Ok (f a b)
    | Error e, Ok _ -> Error e
    | Ok _, Error e -> Error e
    | Error e1, Error e2 ->
        // รวมข้อความ error แบบง่าย
        match e1,e2 with
        | ValidationError ne1, ValidationError ne2 -> ValidationError (NonEmptyList.append ne1
ne2) |> Error
        | _ -> e1 |> Error

module AsyncResult =
  let map f ar = async { let! r = ar in return Result.map f r }
  let bind f ar = async { let! r = ar in match r with Ok x -> f x | Error e -> async { return Error e }
}
  let ofResult r = async { return r }
  let ofAsync a = async { let! x = a in return Ok x }
  let mapError f ar = async { let! r = ar in return Result.mapError f r }

// ---------- Value Objects & Smart Constructors ----------
type EmailAddress = private EmailAddress of string
module EmailAddress =
  let value (EmailAddress s) = s
  let create (s:string) =
    if String.IsNullOrWhiteSpace s then ValidationError (NonEmptyList.create "Email is required"
[]) |> Error
    elif s.Contains("@") |> not then ValidationError (NonEmptyList.create "Email must contain @"
[]) |> Error
    else Ok (EmailAddress (s.Trim().ToLowerInvariant()))

type NonEmptyString = private NonEmptyString of string
module NonEmptyString =
  let value (NonEmptyString s) = s
```

```
  let create (s:string) =
    if String.IsNullOrWhiteSpace s then ValidationError (NonEmptyList.create "Value cannot be
empty" []) |> Error
    else Ok (NonEmptyString (s.Trim()))

type Quantity = private Quantity of int
module Quantity =
  let value (Quantity q) = q
  let create (q:int) =
    if q <= 0 then ValidationError (NonEmptyList.create "Quantity must be > 0" []) |> Error
    else Ok (Quantity q)

type Currency = USD | THB | EUR
type Money = private Money of amount:decimal * currency:Currency
module Money =
  let value (Money (a,c)) = a,c
  let create (c:Currency) (a:decimal) =
    if a < 0m then ValidationError (NonEmptyList.create "Money cannot be negative" []) |> Error
    else Ok (Money (decimal.Round(a,2), c))
  let add (Money(a,c)) (Money(b,c2)) =
    if c<>c2 then invalidArg "currency" "Currency mismatch"
    Money(a+b,c)
  let mul (Money(a,c)) (k:int) = Money(a*decimal k, c)
  let zero c = Money(0m,c)

type CustomerId = CustomerId of Guid
type OrderId = OrderId of Guid
type ProductId = ProductId of string
type Sku = Sku of string

// ---------- Domain: Order Aggregate as State DU ----------
type OrderItem = {
  Sku: Sku
  ProductId: ProductId
```

```
  Name: NonEmptyString
  UnitPrice: Money
  Qty: Quantity
}

module OrderItem =
  let lineTotal (x:OrderItem) =
    let (q) = Quantity.value x.Qty
    Money.mul x.UnitPrice q

type PaymentInfo = {
  Method: string
  PaidAt: DateTimeOffset
  Amount: Money
  TxId: NonEmptyString
}

type Shipment = {
  Carrier: NonEmptyString
  TrackingNo: NonEmptyString
  ShippedAt: DateTimeOffset
}

type DraftOrder = {
  Id: OrderId
  Customer: CustomerId
  Currency: Currency
  Items: OrderItem list
  CreatedAt: DateTimeOffset
}

type PendingPaymentOrder = {
  Id: OrderId
  Customer: CustomerId
```

```
  Items: OrderItem list
  Total: Money
  CreatedAt: DateTimeOffset
  SubmittedAt: DateTimeOffset
}

type PaidOrder = {
  Id: OrderId
  Customer: CustomerId
  Items: OrderItem list
  Total: Money
  Payment: PaymentInfo
  CreatedAt: DateTimeOffset
  SubmittedAt: DateTimeOffset
}

type ShippedOrder = {
  Id: OrderId
  Customer: CustomerId
  Items: OrderItem list
  Total: Money
  Payment: PaymentInfo
  Shipment: Shipment
  CreatedAt: DateTimeOffset
  SubmittedAt: DateTimeOffset
}

type CancelledOrder = {
  Id: OrderId
  Customer: CustomerId
  Reason: NonEmptyString
  CancelledAt: DateTimeOffset
  CreatedAt: DateTimeOffset
}
```

```
type Order =
  | Draft of DraftOrder
  | PendingPayment of PendingPaymentOrder
  | Paid of PaidOrder
  | Shipped of ShippedOrder
  | Cancelled of CancelledOrder

// ---------- Domain Events ----------
type OrderEvent =
  | OrderCreated of OrderId * CustomerId
  | ItemAdded of OrderId * Sku * Quantity
  | OrderSubmitted of OrderId * Money
  | PaymentRecorded of OrderId * NonEmptyString * Money
  | OrderShipped of OrderId * NonEmptyString
  | OrderCancelled of OrderId * NonEmptyString

// ---------- Business Rules & Transitions (ROP: Result) ----------
module Rules =
  let private sumTotal (currency:Currency) (items:OrderItem list) =
    items |> List.fold (fun acc it -> Money.add acc (OrderItem.lineTotal it)) (Money.zero currency)

  let createDraft (currency:Currency) (customer:CustomerId) : Order * OrderEvent list =
    let order = Draft {
      Id = OrderId (Guid.NewGuid())
      Customer = customer
      Currency = currency
      Items = []
      CreatedAt = DateTimeOffset.UtcNow
    }
    match order with
    | Draft d -> order, [ OrderCreated (d.Id, d.Customer) ]
    | _ -> failwith "impossible"
```

```
  let addItem (sku:Sku) (prodId:ProductId) (name:NonEmptyString) (unitPrice:Money)
(qty:Quantity)
              (order:Order) : Result<(Order * OrderEvent list), DomainError> =
    match order with
    | Draft d ->
        let item = { Sku=sku; ProductId=prodId; Name=name; UnitPrice=unitPrice; Qty=qty }
        let items = d.Items @ [item]
        let updated = Draft { d with Items = items }
        Ok (updated, [ ItemAdded (d.Id, sku, qty) ])
    | _ -> Error (RuleViolation "Items can be added only in Draft state")

  let submit (order:Order) : Result<(Order * OrderEvent list), DomainError> =
    match order with
    | Draft d ->
        if List.isEmpty d.Items then Error (RuleViolation "Cannot submit empty order")
        else
          let total = sumTotal d.Currency d.Items
          let submitted = PendingPayment {
            Id=d.Id; Customer=d.Customer; Items=d.Items; Total=total
            CreatedAt=d.CreatedAt; SubmittedAt=DateTimeOffset.UtcNow
          }
          Ok (submitted, [ OrderSubmitted (d.Id, total) ])
    | _ -> Error (RuleViolation "Only Draft order can be submitted")

  let recordPayment (payment:PaymentInfo) (order:Order)
      : Result<(Order * OrderEvent list), DomainError> =
    match order with
    | PendingPayment p ->
        let (amtCur, payCur) =
          let _, c1 = Money.value p.Total
          let _, c2 = Money.value payment.Amount
          c1, c2
        if amtCur <> payCur then Error (RuleViolation "Payment currency mismatch")
```

```
        elif payment.Amount <> p.Total then Error (RuleViolation "Payment amount must equal
order total")
        else
          let paid = Paid {
            Id=p.Id; Customer=p.Customer; Items=p.Items; Total=p.Total; Payment=payment
            CreatedAt=p.CreatedAt; SubmittedAt=p.SubmittedAt
          }
          Ok (paid, [ PaymentRecorded (p.Id, payment.TxId, payment.Amount) ])
    | _ -> Error (RuleViolation "Payment allowed only when PendingPayment")

  let ship (shipment:Shipment) (order:Order) : Result<(Order * OrderEvent list), DomainError> =
    match order with
    | Paid p ->
        let shipped = Shipped {
          Id=p.Id; Customer=p.Customer; Items=p.Items; Total=p.Total
          Payment=p.Payment; Shipment=shipment
          CreatedAt=p.CreatedAt; SubmittedAt=p.SubmittedAt
        }
        Ok (shipped, [ OrderShipped (p.Id, shipment.TrackingNo) ])
    | _ -> Error (RuleViolation "Shipping allowed only when Paid")

  let cancel (reason:NonEmptyString) (order:Order) : Result<(Order * OrderEvent list),
DomainError> =
    match order with
    | Draft d ->
        let cancelled = Cancelled {
          Id=d.Id; Customer=d.Customer; Reason=reason
          CancelledAt=DateTimeOffset.UtcNow; CreatedAt=d.CreatedAt
        }
        Ok (cancelled, [ OrderCancelled (d.Id, reason) ])
    | PendingPayment p ->
        // สมมติ นโยบาย: ยกเลิกได้จนกว่าจะจ่ายเงินสำเร็จ
        let cancelled = Cancelled {
          Id=p.Id; Customer=p.Customer; Reason=reason
```

```
            CancelledAt=DateTimeOffset.UtcNow; CreatedAt=p.CreatedAt
        }
        Ok (cancelled, [ OrderCancelled (p.Id, reason) ])
    | _ -> Error (RuleViolation "Cannot cancel after payment")

// ---------- Repository abstraction (ตัวอย่างง่าย) ----------
type ETag = int // ตัวอย่างเลขเวอร์ชัน (optimistic concurrency)

type OrderRepository = {
  Get : OrderId -> Async<Option<Order * ETag>>
  Save : OrderId * (Order * ETag option) -> Async<Result<ETag,DomainError>>
}

// In-memory example (สำหรับเดโม/ทดสอบ)
module InMemoryRepo =
  let create () =
    let store = System.Collections.Concurrent.ConcurrentDictionary<OrderId, struct
(Order*ETag)>()
    {
      Get = fun id -> async {
        match store.TryGetValue id with
        | true, struct (o,tag) -> return Some (o, tag)
        | _ -> return None
      }
      Save = fun (id,(order,etagOpt)) -> async {
        match etagOpt with
        | None ->
            let tag = 1
            store[id] <- struct (order, tag)
            return Ok tag
        | Some etag ->
            match store.TryGetValue id with
            | true, struct (_, current) when current = etag ->
                let newTag = etag + 1
```

```
                store[id] <- struct (order, newTag)
                return Ok newTag
            | true, _ -> return Error ConcurrencyConflict
            | _ ->
                // ไม่มีข้อมูลเดิมแต่ดันส่ง etag มาด้วย ⟶ ถือว่า conflict
                return Error ConcurrencyConflict
        }
    }

// ---------- Application Service (ใช้ ROP pipeline) ----------
module App =
  open Rules

  // DTO สำหรับคำสั่งสร้าง/ส่งออเดอร์
  type AddItemDto = { Sku:string; ProductId:string; Name:string; UnitPrice:decimal; Qty:int }
  type CreateAndSubmitDto = {
    CustomerId: Guid
    Currency: Currency
    Items: AddItemDto list
  }

  let private toOrderItem (currency:Currency) (dto:AddItemDto) :
Result<OrderItem,DomainError> =
    result {
      let! name = NonEmptyString.create dto.Name
      let! qty  = Quantity.create dto.Qty
      let! price = Money.create currency dto.UnitPrice
      return {
        Sku = Sku dto.Sku
        ProductId = ProductId dto.ProductId
        Name = name
        UnitPrice = price
        Qty = qty
      }
```

```
    }

  let createSubmit (repo:OrderRepository) (cmd:CreateAndSubmitDto)
    : Async<Result<OrderId * ETag * OrderEvent list, DomainError>> = async {
    // 1) สร้าง Draft + Event
    let order0, ev0 =
      Rules.createDraft cmd.Currency (CustomerId cmd.CustomerId)

    // 2) แปลงและเพิ่มรายการ
    let addAll (order:Order, evs:OrderEvent list) (dto:AddItemDto) =
      result {
        let! item = toOrderItem cmd.Currency dto
        let! (o,e) = Rules.addItem item.Sku item.ProductId item.Name item.UnitPrice item.Qty
order
        return (o, evs @ e)
      }

    // fold ด้วย Result
    let itemsResult =
      cmd.Items
      |> List.fold (fun acc dto -> Result.bind (fun st -> addAll st dto) acc) (Ok (order0, ev0))

    match itemsResult with
    | Error e -> return Error e
    | Ok (order1, evs1) ->
        // 3) Submit
        match Rules.submit order1 with
        | Error e -> return Error e
        | Ok (order2, evs2) ->
            // 4) Save
            let (orderId) =
              match order2 with
              | PendingPayment x -> x.Id
              | _ -> failwith "impossible"
```

```
            let! saveRes = repo.Save (orderId, (order2, None))
            match saveRes with
            | Ok etag -> return Ok (orderId, etag, evs1 @ evs2)
            | Error e -> return Error e
  }

// ---------- Demo (รันใน fsx เพื่อตรวจเส้นทางหลัก) ----------
module Demo =
  open App
  open Rules

  let run () = async {
    let repo = InMemoryRepo.create()
    let cmd = {
      CreateAndSubmitDto.CustomerId = Guid.NewGuid()
      Currency = THB
      Items = [
        { Sku="ABC-001"; ProductId="P-001"; Name="USB Cable"; UnitPrice=120m; Qty=2 }
        { Sku="ABC-002"; ProductId="P-XYZ"; Name="Power Adapter"; UnitPrice=350m; Qty=1 }
      ]
    }
    let! res = App.createSubmit repo cmd
    match res with
    | Ok (orderId, _, events) ->
        printfn "Order %A submitted.\nEvents:" orderId
        events |> List.iter (printfn " - %A")
    | Error err ->
        printfn "ERROR: %A" err
  }

// Uncomment เพื่อทดลองเมื่อรันด้วย dotnet fsi
// Demo.run() |> Async.RunSynchronously
```

จุดเด่นเชิงโมเดล

- สถานะออเดอร์เป็น **DU** (Order = Draft | PendingPayment | Paid | Shipped | Cancelled) ⟶ โค้ดบังคับให้เรียกฟังก์ชันผ่านเส้นทางที่ถูกต้อง
- อินวาเรียนต์สำคัญ (เช่น quantity > 0, money ≥ 0) ถูกบังคับผ่าน **Smart constructors**
- กฎธุรกิจฝังอยู่ในฟังก์ชัน transition (addItem, submit, recordPayment, ship, cancel) และคืนค่าเป็น Result (แนว ROP)
- เหตุการณ์ถูกผลิตเคียงข้างการเปลี่ยนสถานะ เพื่อนำไปใช้กับ event store/notification/analytics ภายหลัง

---

4) การใช้ Discriminated Union กับกฎธุรกิจ (Business Rules)

แนวคิดหลัก

1. **Model State Machine ด้วย DU**: ทำ illegal transitions ให้เขียนไม่ติด
2. **State-specific Data**: เก็บข้อมูลที่ “มีความหมายเฉพาะในสถานะนั้น” ไว้ใน record ของสถานะนั้นเท่านั้น (เช่น SubmittedAt อยู่หลัง submit เป็นต้น)
3. **Invariant Enforcement**: Smart constructors + transitions ตรวจสอบก่อนข้ามสถานะ
4. **Pattern Matching**: โค้ดอ่านตรง ๆ ว่า “จากสถานะ X ไป Y ได้ไหม”

ตัวอย่างกฎเพิ่มเติม

- “สกุลเงินของ payment ต้องตรงกับ order”
- “จำนวนเงินที่จ่ายต้องเท่ากับยอดรวม”
- “ยกเลิกได้เฉพาะ Draft/PendingPayment”

ดูในฟังก์ชัน recordPayment, cancel ในโค้ดด้านบน—ทั้งหมดแสดงผ่าน match-case ที่ชัดเจน

---

5) Railway Oriented Programming (ROP)

ภาพรวม

- มองฟังก์ชันเป็น “รางสองเส้น” (สำเร็จ/ล้มเหลว): Result<'T,'E>
- ต่อท่อ (compose) ด้วย bind (หรือ CE result { ... }) เพื่อให้เมื่อพลาดที่จุดใด จะหยุดและส่ง error ออกมา
- เมื่อเกี่ยวกับ I/O ใช้ Async<Result<_,_>> (หรือ Task<Result<_,_>>) แล้วมี combinator AsyncResult.bind/map
- **Validation แบบสะสมหลายข้อผิดพลาด** ใช้ Applicative (map2, apply) และ NonEmptyList<string> เพื่อรวม error

ตัวอย่าง Pipeline

ใน App.createSubmit:

1. แปลง DTO ⟶ Value Objects (smart constructors)

2. เพิ่ม items ลง order (fold ด้วย Result.bind)
3. submit และคำนวณ Total
4. บันทึกด้วย repo (I/O) ⟶ Async<Result<_,_>>
   ทั้งเส้นจะล้มเหลวทันทีถ้าจุดใดจุดหนึ่งผิด (fail-fast)

ถ้าต้องการ “สะสม error” ของ input ทั้งหมดก่อนหยุด (เช่น validate ฟอร์ม), ใช้แนว Applicative แยกต่างหากสำหรับช่วง validate แล้วจึงเข้า pipeline ROP ภายหลัง

---

6) การออกแบบเพื่อ “Illegal States Unrepresentable”

ตัวอย่าง:

- Quantity เป็นชนิดเฉพาะที่รับรองค่าบวกเสมอ
- Money บังคับไม่ติดลบและตรึงสกุลเงินไว้ในค่า
- Order แยกสถานะด้วย DU จึง “เรียก ship() กับ Draft ไม่ได้” (คอมไพเลอร์/โค้ดบังคับผ่าน pattern matching)

ผล: ลดจำนวน branch/check ที่กระจายมั่ว ๆ และลดบั๊กชั้น logic

---

7) Testing & Property-Based Testing (แนะนำแนวทาง)

- **Unit Test** ระดับ transition:
  - submit Draft(order with items) ต้อง Ok PendingPayment
  - submit Draft(empty items) ต้อง Error RuleViolation
- **Property-Based** (FsCheck):
  - สุ่ม Qty>0 และ UnitPrice≥0 แล้วตรวจว่า sumTotal ไม่ลบ
  - state machine: ไม่ควรมีลูป/ย้อนไปสถานะก่อนหน้าโดยไม่มีเหตุผล
- **Event assertions**: ตรวจรายการ OrderEvent ที่ควรถูกปล่อยทุกครั้ง

---

8) Integration ระหว่าง Bounded Context

- ใช้ **DTO + Anti-Corruption Layer (ACL)** แยกชนิด/รูปแบบข้อมูลภายนอกออกจาก Value Objects ภายใน
- mapping ภายนอก ⟶ ภายใน ต้องผ่าน Smart constructors เสมอ (ป้องกันข้อมูลเสีย/ไม่ตรงสเปค)
- พิจารณา **Domain Events ⟶ Integration Events** ถ้าต้องข้ามคอนเท็กซ์ผ่าน message bus

---

9) แนวปฏิบัติแนะนำ & กับดัก

ทำ

- เริ่มด้วย Ubiquitous Language: นิยามคำหลักของโดเมนให้ทีมเข้าใจตรงกัน

- ทำ Event Storming เพื่อค้นหา Aggregate/Boundary/Policy
- ใช้ DU กับ state machine, Smart constructors กับค่าที่ต้องปลอดภัย
- แยก App Service (จัด orchestration/I/O) ออกจาก Domain pure functions

เลี่ยง

- ใช้ record ใหญ่ตัวเดียวใส่ทุก field ทุกสถานะ (จะเปิดช่องผิดกฎง่าย)
- ทำ Validation หลังสุด (ช้าไป—ควรที่ขอบ)
- ใช้ string/raw primitive ไปทั่ว (ทำให้ type safety หายหมด)

---

10) ขยายผลต่อ

- ใส่ **Units of Measure** ให้ Money (เช่น ฿, $, €) หรือใช้ DU + policy แปลงสกุลเงิน
- เปลี่ยน Repository ให้ใช้ฐานข้อมูลจริง (EF Core, Dapper) แต่ยังคงสัญญา (OrderRepository)
- เพิ่ม **Policy/Promotion** เป็น **Domain Service** เพียว ๆ รับ OrderItem list ⟶ Discount (Result)
- เติม **Concurrency** จริงจังด้วย ETag/version จากฐานข้อมูล
- ครอบ pipeline ด้วย **Retry/Idempotency** ในชั้น Application Service

---

สรุปสั้น

- F# ให้เครื่องมือที่ทรงพลังสำหรับ DDD: **DU + Records + Immutability + Pattern Matching + Smart Constructors**
- ใช้ **ROP** (Result/AsyncResult) สร้าง pipeline ที่ชัดเจน ปลอดภัย และอ่านง่าย
- แบบจำลองที่ดีทำให้ “สถานะ/การเปลี่ยนสถานะที่ผิด” เขียนไม่ติดตั้งแต่ต้นทาง ⟶ โค้ดเสถียรและสื่อสารโดเมนได้ตรงกว่าเดิม

Domain Modeling with F# ให้ละเอียดเชิงลึก

Domain Modeling with F# ให้ละเอียดเชิงลึกที่สุด ครอบคลุมทั้ง **ทฤษฎี ⟶ กลยุทธ์ ⟶ Tactical Design ⟶ Mapping กับ F# ⟶ ตัวอย่างเชิงลึก**

บทที่ 18: Domain Modeling with F#

1) Domain-Driven Design (DDD) เชิงลึก

1.1 ปัญหาที่ DDD แก้ไข

- แอปพลิเคชันธุรกิจส่วนใหญ่ “ซับซ้อนเพราะโดเมน” (Business complexity) ไม่ใช่เพราะเทคโนโลยี

- โค้ดมักเต็มไปด้วย if/else กระจัดกระจาย จนไม่สะท้อน Business Rules
- “ข้อมูลผิด” หรือ “สถานะไม่ถูกต้อง” (Illegal State) หลุดเข้าระบบได้ง่าย

➡ DDD เน้น “แบบจำลองที่ตรงกับโดเมนจริง” และทำให้ **Ubiquitous Language** (ภาษาเดียวกันทั้งทีม dev + domain expert) ถูกฝังในโค้ด

1.2 Strategic Design

- **Bounded Context**: ขอบเขตโดเมนย่อยที่มี **คำศัพท์และโมเดล** ของตัวเอง (เช่น Sales, Billing, Shipping)
- **Context Map**: วิธีที่ Context ต่าง ๆ สื่อสารกัน (Upstream, Downstream, ACL)
- **Anti-Corruption Layer (ACL)**: ป้องกันโมเดลภายในไม่ปนเปื้อนด้วยโมเดลภายนอก

1.3 Tactical Design

- **Entity**: มี Identity ถาวร (เช่น Customer, Order)
- **Value Object**: ไม่มี Identity เทียบด้วยค่า (เช่น Email, Money, Quantity)
- **Aggregate**: กลุ่มของ Entity/Value Object ที่รักษา Invariant ร่วมกัน มี Aggregate Root
- **Domain Event**: เหตุการณ์ที่เกิดในโดเมน (เช่น OrderSubmitted, PaymentRecorded)
- **Repository**: Interface สำหรับ persistence (ไม่โยน infrastructure มาปนใน domain)
- **Factory / Policy / Service**: สำหรับสร้าง/คำนวณสิ่งที่ซับซ้อน

1.4 Mapping DDD ⟶ F#

- **Entity**: record + Id (เช่น Guid)
- **Value Object**: single-case DU + smart constructor
- **Aggregate**: ใช้ **Discriminated Union** (DU) เป็น State Machine
- **Domain Rules**: pure function + pattern matching
- **Domain Event**: DU ของเหตุการณ์
- **Repository**: interface เป็น record ของฟังก์ชัน get/save

---

2) Discriminated Union (DU) กับ Business Rules

2.1 State Machine

- ใช้ DU สื่อ “สถานะที่เป็นไปได้”

```
type Order =
  | Draft of DraftOrder
  | PendingPayment of PendingPaymentOrder
```

```
  | Paid of PaidOrder
  | Shipped of ShippedOrder
  | Cancelled of CancelledOrder
```

2.2 กฎธุรกิจฝังใน Transition

- submit : Order -> Result<Order,Error>
- recordPayment : Order -> Payment -> Result<Order,Error>
- ship : Order -> Shipment -> Result<Order,Error>

2.3 "Make Illegal States Unrepresentable"

- ถ้า Order อยู่ใน Draft ⟶ ไม่มี SubmittedAt field
- ถ้า Order อยู่ใน Paid ⟶ ต้องมี PaymentInfo เสมอ
- Compiler จะบังคับให้โค้ดเรา "match ครบทุกสถานะ" ⟶ ลด bug logic

---

3) Railway Oriented Programming (ROP)

3.1 ปัญหาการเขียนโค้ดแบบ Imperative

```
if (!ValidateEmail(email)) return Error("Invalid")
if (!ValidateQty(qty)) return Error("Invalid")
var result = SaveOrder(...)
if (!result.Ok) return Error("DB failed")
return Success(result.Order)
```

➡ if/else ซ้อนกันเยอะ, error-handling ปน logic

3.2 แนวคิด ROP

- ใช้ Result<'T,'E> = Ok หรือ Error
- เขียนฟังก์ชันแบบ **"ท่อ" (pipeline)**:

```
let placeOrder dto =
  createDraft dto
  |> Result.bind (addItems dto.Items)
  |> Result.bind submit
  |> Result.bind save
```

- **ข้อดี**: อ่านเหมือน flow ธรรมชาติ, Error หยุดท่ออัตโนมัติ

3.3 Composition

- **bind**: ฟังก์ชัน Result<'a,'e> -> ('a -> Result<'b,'e>) -> Result<'b,'e>

- **map**: ฟังก์ชัน Result<'a,'e> -> ('a -> 'b) -> Result<'b,'e>

ใน F#:

```
result {
  let! email = EmailAddress.create dto.Email
  let! qty   = Quantity.create dto.Qty
  let! order = submit draft
  return order
}
```

➡ ใช้ Computation Expression (result { ... }) เขียนเป็น imperatively แต่ยัง pure

---

3.4 Async + Result

- เมื่อมี I/O (DB, API) ⟶ ใช้ Async<Result<'T,'E>>
- Combinator AsyncResult.bind/map ช่วยผูกหลายขั้นตอนแบบ async

---

4) ตัวอย่างเชิงลึก: กรณี Order

4.1 Smart Constructors

```
type Quantity = private Quantity of int
module Quantity =
  let create q =
    if q <= 0 then Error "Qty must be >0" else Ok (Quantity q)
```

4.2 Transition Function

```
let submit order =
  match order with
  | Draft d when d.Items |> List.isEmpty -> Error "Empty order"
  | Draft d ->
      let total = d.Items |> List.sumBy (fun i -> i.Price)
      Ok (PendingPayment { Id=d.Id; Items=d.Items; Total=total })
  | _ -> Error "Only Draft can be submitted"
```

4.3 Pipeline (ROP)

```
let placeOrder dto =
  result {
    let! draft = createDraft dto.CustomerId
    let! order1 = addItems dto.Items draft
    let! order2 = submit order1
```

```
    return order2
}
```

---

5) เชื่อมโยง DDD + DU + ROP

1. **DDD** ⟶ โครงสร้างโมเดล (Entity, VO, Aggregate, Event)
2. **DU** ⟶ ใช้แทน State Machine ของ Aggregate
3. **ROP** ⟶ ใช้แทน Flow การทำงาน ที่คืนค่าเป็น Result, Fail-fast แต่โค้ดอ่านง่าย

---

6) Advanced Topics

- **Validation Accumulation**: ใช้ Applicative เพื่อสะสม error หลายข้อ
- **CQRS + Event Sourcing**: ใช้ Domain Event (DU) เป็น source of truth
- **Property-Based Testing (FsCheck)**: ตรวจว่า state machine ไม่มีการ transition ที่ผิด
- **Interop กับ C#**: F# domain model ยังถูกใช้จาก C# ได้ แต่จะได้ type safety เพิ่มขึ้น

☐ **สรุปเชิงลึก**

- DDD = ปรับโครงสร้างระบบให้สะท้อนโดเมนจริง
- DU = ทำให้ “State Machine” ของธุรกิจปลอดภัย, illegal states unrepresentable
- ROP = ใช้ Result/AsyncResult ทำ pipeline การทำงานที่ชัดเจนและง่ายต่อการทดสอบ

Domain-Driven Design (DDD) เชิงลึก

1) แนวคิดหลักของ DDD

DDD ไม่ใช่แค่เทคนิคการโค้ด แต่คือ **แนวคิดในการออกแบบระบบซอฟต์แวร์** โดยให้ความสำคัญกับ “โดเมนธุรกิจ” มากกว่า “เทคโนโลยี”

- **Domain** = พื้นที่ของปัญหา (Problem Space) เช่น E-Commerce, Banking, Healthcare
- **Ubiquitous Language** = ภาษาเดียวที่ทีม Dev + Domain Expert ใช้ร่วมกัน
- **Model** = โครงสร้างแนวคิดที่สื่อถึงโดเมนในเชิงซอฟต์แวร์

☐ เป้าหมายคือ “ทำให้โค้ด = โมเดลธุรกิจ” (ไม่ใช่โค้ดเป็นเพียง CRUD DB)

---

2) Strategic Design

2.1 Bounded Context

- ขอบเขตที่ชัดเจนของโมเดลและคำศัพท์
- ตัวอย่าง:
    - **Sales Context** ⟶ “Order = สิ่งที่ลูกค้าสั่ง”