Lec 18: Domain Modeling

CS220: Programming Principles

Sang Kil Cha
Domain-Driven Design
Disclaimer

This lecture is mainly inspired by https://www.slideshare.net/ScottWlaschin/domain-driven-design-with-the-f-type-system-functional-londoners-2014.
Motivation: S/W Engineering is Difficult

As the code size gets larger, it gets harder and harder to understand.
What is DDD?

DDD is an approach to developing complex software in which we focus on the core domain and domain logic rather than technology.

Domain is a sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.¹

¹Eric Evans, Domain-Driven Design Reference.
Designing a Card Game?

```ocaml
module CardGame = 
  type Suit = Club | Diamond | Spade | Heart
  type Rank = Ace | Two | Three | ... | Jack | Queen | King
  type Card = Suit * Rank
  type Hand = Card list
  type Deck = Card list
  type Player = { Name: string; Hand: Hand }
  type Game = { Deck: Deck; Players: Player list }
  type Deal = Deck -> Deck * Card
  type PickupCard = Hand -> Card -> Hand
```

DDD with Rich Type System

F#'s rich type system helps in designing your program in a very abstract manner.

```fsharp
type PlayerInfo = {
    FirstName: string // Can have maximum 50 chars.
    MiddleName: string // This is optional
    LastName: string // Can have maximum 50 chars.
    EmailAddress: string // Should follow email format.
    IsEmailVerified: bool
}
```
Optional?

```typescript
type PlayerInfo = {
  FirstName: string
  MiddleName: string option
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
}
```
Constrained String

```
type String50 = String50 of string

module String50 =
    let create s = // ...

type PlayerInfo = {
    FirstName: String50
    MiddleName: string option
    LastName: String50

    EmailAddress: string
    IsEmailVerified: bool
}
```
Constrained String (cont’d)

type EmailAddress = EmailAddress of string

module EmailAddress =
    let create s = // ...

type PlayerInfo = {
    FirstName: String 50
    MiddleName: string option
    LastName: String 50

    EmailAddress: EmailAddress
    IsEmailVerified: bool
}
type PlayerName = {
    FirstName: String\text{50}
    MiddleName: string\text{ option}
    LastName: String\text{50}
}
type Email = {
    EmailAddress: EmailAddress
    IsEmailVerified: bool
}
type PlayerInfo = {
    Name: PlayerName
    Email: Email
}
Flag variable

```csharp
type Email = {
    EmailAddress: EmailAddress
    IsEmailVerified: bool
}
```

If the email address changes, then the verified flag should set to false. To make the two fields synchronized, one needs to carefully implement the code, which can be error-prone.
Encode Domain Logic into Types

```plaintext
type VerifiedEmail = Verified of EmailAddress

type Email =
| Unverified of EmailAddress
| Verified of VerifiedEmailAddress

/// Just need to write a function
val verify: EmailAddress -> VerifiedEmail option
```
Changed Domain?

We can easily adopt new requirements with the help of strong type system.

```typescript
type PlayerInfo = {
  Name: PlayerName
  Email: Email
  Address: PostalAddress // Added!
}
```
The Key Take-Away

How many lines of functions did we write so far? The domain has evolved as we write mostly *types*. F# is the perfect language to do DDD, which is a well-known software engineering approach for building complex systems.