Lec 24: Domain Modeling

CS220: Programming Principles

Sang Kil Cha

Domain-Driven Design

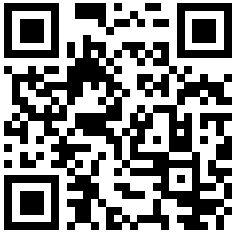
Disclaimer

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

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don’t accept late responses.



Motivation: S/W Engineering is Difficult

If your code is not well-structured, it becomes a nightmare to maintain and extend it.
How can we structure our code in a way that it is easy to understand and maintain?

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]

[1] Eric Evans, Domain-Driven Design Reference.

Do Not Focus on Technology

- DDD is ***not about technology***.
- DDD is about understanding the domain.
- DDD is about understanding the business.

By understanding the domain, we can build our program in a way that it reflects the domain, which makes it easier to understand and maintain.

Ubiquitous Language

DDD requires that users (i.e., customers) and developers share a common language to describe the domain. This is so-called ***ubiquitous language***.

For example, suppose we are building a system for a bank. The term ***account*** should have the same meaning for both the users and the developers. Is it a checking account, a savings account, or both? How about joint accounts? When we talk about ***account***, we should all have the same understanding.

F#’s Rich Type System

F#’s rich type system is a perfect fit for DDD. We can encode domain logic into types, which makes it easier to understand and maintain the code.

Designing a Card Game

First, let's design a card game. What are the components of a card game?

- A card game consists of a deck of cards, a set of players, and a set of rules.
- A deck of cards consists of a set of cards.
- A card consists of a suit and a rank.
- A player consists of a name and a hand of cards.
- A hand of cards consists of a set of cards.

Can we describe this domain in F#?

Designing a Card Game

Start by defining the types.

```
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
```

Types are Self-Explanatory

The types are “mostly” self-explanatory even for non-programmers, i.e., customers or domain experts. In fact, we can make it even clearer by leveraging comments.

```
module CardGame =
  /// Represents a suit of a card.
  type Suit = Club | Diamond | Spade | Heart

  /// Represents a rank of a card.
  type Rank = Ace | Two | Three | ... | Jack | Queen | King

  ...
```

Discuss with Domain Experts

Using these types, we can discuss with customers (domain experts) to make sure that we understand the domain correctly. This is a very important step in DDD.

We haven't implemented any functions/classes yet. We are just discussing the type definitions, which describe the domain.

Discuss with Domain Experts

Using these types, we can discuss with customers (domain experts) to make sure that we understand the domain correctly. This is a very important step in DDD.

We haven't implemented any functions/classes yet. We are just discussing the type definitions, which describe the domain.

Rich type system enables DDD!

Example: PlayerInfo

Before

```
type PlayerInfo = {

  FirstName: string

  MiddleName: string

  LastName: string

  EmailAddress: string

  IsEmailVerified: bool
}
```

After

```
type PlayerInfo = {
  /// Can have maximum 50 chars.
  FirstName: string
  /// This should be optional.
  MiddleName: string
  /// Can have maximum 50 chars.
  LastName: string
  /// Should follow email format.
  EmailAddress: string
  /// True only if the email is verified.
  IsEmailVerified: bool
}
```

Make a Field Optional

```
type PlayerInfo = {
  FirstName: string
  /// Middle name is optional.
  MiddleName: string option
  LastName: string
  EmailAddress: string
  IsEmailVerified: bool
}
```

Constrained String

```
(* This can be later changed to a class if needed. *)
type StringMaxFiftyChars = StringMaxFiftyChars of string

module StringMaxFiftyChars = (* N.B. no need to have this module for now *)
  let create s = failwith "TODO"
```

```
type PlayerInfo = {
  FirstName: StringMaxFiftyChars
  MiddleName: string option
  LastName: StringMaxFiftyChars
  EmailAddress: string
  IsEmailVerified: bool
}
```

Email Address Type

```
type EmailAddress = EmailAddress of string

module EmailAddress = (* no need to have this for now *)
  let create s = failwith "TODO"
```

```
type PlayerInfo = {
  FirstName: StringMaxFiftyChars
  MiddleName: string option
  LastName: StringMaxFiftyChars
  EmailAddress: EmailAddress
  IsEmailVerified: bool
}
```

Divide Types by (Sub)Domain

```
type PlayerName = {
  FirstName: StringMaxFiftyChars
  MiddleName: string option
  LastName: StringMaxFiftyChars
}
```

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

```
type PlayerInfo = {
  Name: PlayerName
  Email: Email
}
```

Flag variable

```
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

```
type VerifiedEmailAddress = Verified of EmailAddress

type Email =
  | Unverified of EmailAddress
  | Verified of VerifiedEmailAddress

module Email =
  // verify: EmailAddress -> VerifiedEmailAddress option
  let verify (email: EmailAddress): VerifiedEmailAddress option =
    failwith "TODO"
```

The Key Take-Away

How many lines of functions did we write so far? The domain model has evolved as we write mostly ***types*** without considering the actual implementation. F# is the perfect language to do DDD, which is a well-known software engineering principle for building complex systems.

This way of developing software is also known as ***type-driven development*** or ***typeful programming***.

Changed Domain?

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

```
type PlayerInfo = {
  Name: PlayerName
  Email: Email
  Address: PostalAddress // Added!
}
```

In-Class Activity #24

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

Problem

Let's practice DDD by designing a model for the following domain. You want to build a system for e-commerce. The system should allow registered customers to get 10% discount when they spend more than 100000 KRW.

We can start by defining a simple `Customer` type:

```
type Customer = {
  Id: string
  IsRegistered: bool
}
```

What's Wrong with the Type?

Having two boolean fields can be error-prone. Can we explicitly encode the registration status? Try to design a better type by yourself. You should also implement the `calculateTotal` function, which takes in a customer and their total spending as input, and returns the total amount After applying the discount. For example, when a customer spends 200000 KRW, then the total amount should be 180000 KRW (after 10% discount).

Conclusion

Further Readings

- https://tomasp.net/blog/type-first-development.aspx/.

Question?