Mini Projects

Project 1: Domain Models made Functional

```
struct Contact {
    first_name: String,
    middle_initial: String,
    last_name: String,

    email: String,
    is_email_verified: bool,
}
```

How many things are wrong with this design?

true if ownership of email address is confirmed

Shared Languages

```
enum Suit { Spades, Hearts, Diamonds, Clubs }

enum Rank {
  Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King, Ace
}

struct Card { suit: Suit, rank: Rank, }

struct Deck { cards: Vec<Card> }

struct Hand { cards: Vec<Card> }

struct Player { name: String, hand: Hand }

struct Game { deck: Deck, players: Vec<Player> }

type Deal = fn(Deck) -> (Deck, Hand);

type PickupCard = fn(Hand, Card) -> Hand;
```

Could non-programmer understand this?

Shared Languages

```
enum Suit { Spades, Hearts, Diamonds, Clubs }

enum Rank {
  Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King, Ace
}

struct Card { suit: Suit, rank: Rank, }

struct Deck { cards: Vec<Card> }

struct Hand { cards: Vec<Card> }

struct Player { name: String, hand: Hand }

struct Game { deck: Deck, players: Vec<Player> }
```

~~type Deal = fn(Deck) -> (Deck, Hand);~~

```
type PickupCard = fn(Hand, Card) -> Hand;
```

Could non-programmer understand this?

Shared Languages

```
enum Suit { Spades, Hearts, Diamonds, Clubs }
enum Rank {
  Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King, Ace
}
struct Card { suit: Suit, rank: Rank, }
struct Deck { cards: Vec<Card> }
struct Hand { cards: Vec<Card> }
struct Player { name: String, hand: Hand }
struct Game { deck: Deck, players: Vec<Player> }
~~type Deal = fn(Deck) -> (Deck, Hand);~~
type PickupCard = fn(Hand, Card) -> Hand;
```

Could non-programmer understand this?

Shared Languages

```
enum Suit { Spades, Hearts, Diamonds, Clubs }
enum Rank {
  Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King, Ace
}
struct Card { suit: Suit, rank: Rank, }
struct Deck { cards: Vec<Card> }
struct Hand { cards: Vec<Card> }
struct Player { name: String, hand: Hand }
struct Game { deck: Deck, pl type Deal = fn(ShuffledDeck) -> (ShuffledDeck, Hand);
type Deal = fn(ShuffledDeck) -> (ShuffledDeck, Hand);
type PickupCard = fn(Hand, Card) -> Hand;
struct ShuffledDeck { cards: Vec<Card> }
type Shuffle = fn(Deck) -> ShuffledDeck;
```

Code should sync with Real World Vocabulary

In the Real World

- Suit
- Rank
- Card
- Deck
- Hand
- Player
- Game
- Deal
- ShuffledDeck
- Shuffle

The design is the code,
The code is the design.

In the Code

- Suit
- Rank
- Card
- Deck
- Hand
- Player
- Game
- Deal
- ShuffledDeck
- Shuffle

Should not use
programmer's jargon

PlayerManager

DeckBase

AbstractCardProxyFactoryBean

Key DDD Principle

Communicate the design in the code

Project 1: What’s wrong again?

```
struct Contact {
    first_name: String,
    middle_initial: String,
    last_name: String,

    email: String,
    is_email_verified: bool,
}
```

How many things are wrong with this design?

true if ownership of email address is confirmed

Project 1: Domain Models made Functional

```
struct Contact {
   first_name: String,
   middle_initial: String,
   last_name: String,

   email: String,
   is_email_verified: bool,
}
```

Which values are optional?

Can names be arbitrarily long?

Can any string be a valid email?

Which fields are linked?
What are the consistency boundaries?

What is the domain logic?
- Must be reset if email is changed

Version 1

Which values are optional?

Can names be arbitrarily long?

Can any string be a valid email?

- Person’s middle name can be omitted.
- Person’s last and first names cannot exceed 50 characters.
- Only strings conforming to valid email address format are allowed.

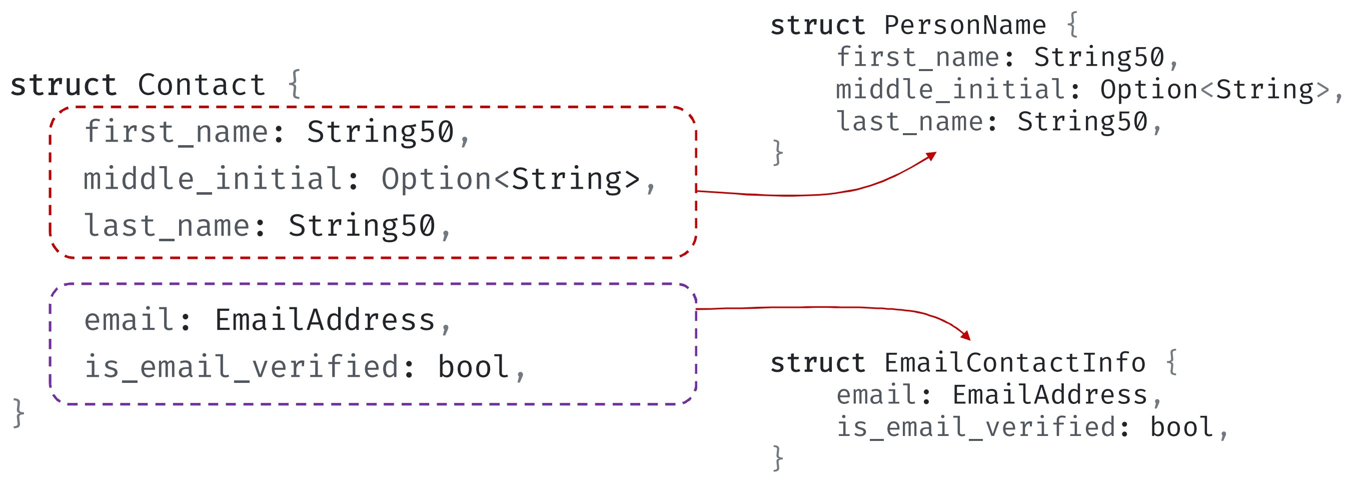
```
struct Contact {
    first_name: ???,
    middle_initial: ???,
    last_name: ???,

    email: ???,
    is_email_verified: bool,
}
```

```
regex = { version = "1.7.1", features = ["std"] }
```

Version 2

Which fields are linked?
What are the consistency boundaries?

- Separate linked fields as separate groups so that each group can be a consistency boundary.



```
struct Contact {
    first_name: String50,
    middle_initial: Option<String>,
    last_name: String50,

    email: EmailAddress,
    is_email_verified: bool,
}
```

```
struct PersonName {
    first_name: String50,
    middle_initial: Option<String>,
    last_name: String50,
}
```

```
struct EmailContactInfo {
    email: EmailAddress,
    is_email_verified: bool,
}
```

Version 3

- **Rule 1**: If email is changed, the verified flag must be reset to false.
- **Rule 2**: The verified email flag can only be set by a special verification service.

```
struct EmailContactInfo {
    email: EmailAddress,
    is_email_verified: bool,
}
```

➡

```
enum EmailContactInfo {
    Unverified(EmailAddress),
    Verified(VerifiedEmail),
}
```

```
type VerificationService = dyn Fn(EmailAddress) -> Option<VerifiedEmail>;
```

Version 4: Making illegal state unrepresentable

- **New rule**: A contact must have an email or postal address.

Rule implies:

1. email address only, or
2. postal address only, or
3. both email and postal address

```
struct ContactV1 {
    name: PersonName,
    email: EmailContactInfo,
    address: PostalContactInfo,
}
```

Any of theses satisfy the constraints?

```
struct ContactV2 {
    name: PersonName,
    email: Option<EmailContactInfo>,
    address: Option<PostalContactInfo>,
}
```

Version 5: Making illegal state unrepresentable

- **New rule**: A contact must have at least one way of contacted.

```
struct Contact {
    name: PersonName,
    primary_contact_info: ContactInfo,
    secondary_contact_info: Option<ContactInfo>,
}

// Way of being contacted
enum ContactInfo {
    Email(EmailContactInfo),
    Postal(PostalContactInfo),
}
```

Project 2: Implement Stack and List

1. Safe Stack
2. Unsafe Stack
3. Safe List
4. Unsafe List
5. Safe Doubly-linked List (See the standard library implementation)