Taking a Gamble with F# Implementing Blackjack

Cameron Presley

@PCameronPresley

http://Blog.TheSoftwareMentor.com

Pilot FLYING J

f(knox)

KNOX HACKS

Goals

Introduction to F# types

Model Blackjack using these types

Handling failures in a functional manner

Types Overview

- How to hold data
- Collections
- Combining different types as one

Know When to Hold Them Holding Data

Source

Holding Data
Tuples

Provides a way to model all possible combinations of types

```
1: // int*string
2: (42, "kumquats")
3:
4: // int*int*int
5: (255, 255, 255)
6:
7: // bool*int
8: (true, 25)
9: (false, 0)
```

Working with Tuples

```
 1: let fiveLemons = (5, "lemons")
 2:
 3: let printNumberOfFruit (number,fruit) =
 4:   printfn "I have %i %s" number fruit
 5:
 6: printNumberOfFruit fiveLemons // I have 5 lemons
 7:
 8: type NumberAndFruit = int*string
 9:
10: let sixApples = (6, "apples")
11: printNumberOfFruit sixApples // I have 6 apples
```

Holding Data
Tuples

Advantages

- Create on the fly
- Easy to deconstruct

Disadvantages

- Parameters have to be in order
  - (**int*string** is different from **string*int**)
- Hard to keep values in the right order (int*int*int)
- Doesn't provide a lot of type safety

```
1: let printNumberOfFruit (number, fruit) =
2:     printfn "I have %i %s" number fruit
3:
4: let fiveLemons = (5, "lemons")
5: printNumberOfFruit fiveLemons // I have 5 lemons
6:
7: let twoBears = (2, "Bears")
8: printNumberOfFruit twoBears // I have 2 bears
```

You Should Use Tuples

- When working within a function
- When there are few parameters

You Should **NOT** Use Tuples

- To represent a domain model
- When more type safety is needed
- Representing the same data types (i.e. int*int*int)
  - Can be hard to remember which element represents which

Records

Provides a way to model all possible combinations of a value

```
1: let person = ("Cameron", 27, "Software Engineer")
2: let diffPerson = (27, "Software Engineer", "Cameron")
3:
4: type Person = {name:string; age:int; jobTitle:string}
5: let person = {name="Cameron"; age=27; jobTitle="Software Engineer"}
6: let samePerson = {age=27; jobTitle="Software Engineer"; name="Cameron"}
```

Working with Records

```
 1: type Person = {name:string; age:int; jobTitle:string}
 2: let cameron = {name="Cameron"; age:27; jobTitle="Software Engineer"}
 3:
 4: let printPerson p =
 5:     printfn "%s is %i years old and is a(n) %s" p.name p.age p.jobTitle
 6:
 7: printPerson cameron // Cameron is 27 years old and is a Software Engineer
 8:
 9: let promotedCameron = {cameron with jobTitle = "Architect"}
10: printPerson promotedCameron // Cameron is 27 years old and is a Architect
```

Advantages

- Order doesn’t matter
- Named parameters
- Easy to retrieve a value

Disadvantages

- Can’t create on the fly
- Verbose definition

When Should You Use Records?

- When representing domain models
- When working with a lot of parameters
- Better type safety is required

When Should You **NOT** Use Records?

- For intermediate results for a function

Review

Tuples

- Create on the fly
- Easy to break down into separate values

Records

- Representing domain models
- Working with a lot of parameters
- Providing better type safety

Collections

Sequences

Define a collection using a generator function

Lazy by default

Similar to IEnumerable from C#

Sequences

```
1: let firstFivePositiveEvens = Seq.init 5 (fun index -> 2*index)
2:
3: let allPositiveEvens = Seq.initInfinite (fun index -> 2*index)
```

```
1: type Date = {Month:int; Day:int; Year:int}
2:
3: let calendar2017 = seq {
4: for month in 1 .. 12 do
5:     for day in 1 .. DateTime.DaysInMonth(2017, month) do
6:         yield {Month=month; Day=day; Year=2017}
7: }
```

Sequences

Advantages

- Can model an infinite collection
- Provides a way to generate all values
- Only computes values when needed

Disadvantages

- Expensive to add a value -> O(n)
- Retrieving an item by index is slow -> O(n)
- Can’t modify elements in place

Lists

Two parts: the first element and rest of the list

- First element referred to as the head
- Rest of the list is referred to as tail

Similar to linked lists

Typically used with recursion, pattern matching, and the :: (cons) operator

Lists

```
1: let rec findEvenNumbers numbers =
2:   match numbers with
3:   | [] -> []
4:   | head::tail ->
5:     if head%2 = 0 then head :: findEvenNumbers tail
6:     else findEvenNumbers tail
```

Lists

Advantages

- Can define recursive functions to process element by element
- Adding additional elements is fast -> O(1)
- Can be defined with a generator function

Disadvantages

- Can’t model infinite series
- Indexing to an element is slow -> O(n)
- Can’t modify elements in place

Arrays

Represents a collection as a dictionary such that the index finds the value

Similar to arrays in other languages

Use when lookup speed matters

Arrays

```
1: let numbers = [|16; 24; 34; 42; 58|]
2:
3: printfn "%i" numbers.[0] // 16
4: printfn "%i" numbers.[1] // 24
5:
6: numbers.[1] <- 99
7:
8: printfn "%i" numbers.[1] // 99
```

Arrays

Advantages

- Quick lookup -> O(1)
- Can change one element in the array with a different element -> O(1)
- Can model multidimensional data (2D arrays)

Disadvantages

- Can’t model infinite series
- Adding additional elements is slow -> O(n)

Review

Sequences

- IEnumerable from C#
- Values can be generated by a function or when modeling an infinite series

Lists

- Linked lists
- Use when the number of elements can change or when processing element by element

Arrays

- Similar to traditional arrays
- Best used when certain elements need to change or for quick access

Combining Different Types Discriminated Unions

Discriminated Unions

A way to combine different types into a single type

```
1: type NumberAndString = {number:int; string:string}
2: let record = {number=42; string="kumquats"}
3:
4: type NumberOrString = Number of int | String of string
5: let number = Number 42
6: let string = String "kumquats"
```

Multicase

```
1: type Shape = Circle of float | Square of float
2:              | Rectangle of int*int
3:
4: let calculateArea shape =
5:   match shape with
6:   | Circle radius -> Math.PI*radius*radius
7:   | Square length -> length*length
8:   | Rectangle (height,width) -> height*width
```

Enum Style

```
1: type Months = Jan | Feb | Mar | Apr | May | Jun
2:              | Jul | Aug | Sep | Oct | Nov | Dec
3:
4: let findMonthNumber month =
5:   match month with
6:   | Jan -> 1 | Feb -> 2 | Mar -> 3
7:   | Apr -> 4 | May -> 5 | Jun -> 6
8:   | Jul -> 7 | Aug -> 8 | Sep -> 9
9:   | Oct -> 10 | Nov -> 11 | Dec -> 12
```

Discriminated Unions

```
1: type Coordinate={Lat:float; Long:float}
2: let coordinate={Lat=43.5703; Long=89.7707}
```

Combining Different Types Discriminated Unions

Create a new coordinate and I do this:

```
1: let newCoordinate = {
2:     Lat=coordinate.Long;
3:     Long=coordinate.Lat
4: }
```

What's to stop me from making this error?

Discriminated Unions

Problem is that Lat and Long are both floats, which means they're interchangeable

Instead of modeling them as floats, they should have their own type

But float is the correct data type, so how I create a type that wraps around a float?

Discriminated Unions

```
1: type Lat = Lat of float
2: type Long = Long of float
3: type Coordinate = {Lat:Lat; Long:Long}
4:
5: let coordinate={Lat=Lat 43.5703; Long=Long 89.7707}
6:
7: // Fails to compile -> A Lat is not a Long
8: let newCoordinate={Lat=coordinate.Long; Long=coordinate.Lat}
```

Discriminated Unions

When different type signatures actually model the same thing

- Multicase

When there are a finite number of possibilities

- Enum Style

When a primitive needs to be modeled as part of the domain

- Single Case

Data Types

Holding Data

- Tuples, Records

Collections

- Sequences, Lists, Arrays

Combining Different Types

- Discriminated Unions

Modeling The Domain

Goals

Has to be easy to read and understand

- Easier to maintain

Defines a ubiquitous language

- Provides a way for developers, QA, and users to communicate

Make illegal states unrepresentable

- Why allow bad data to be created?

Identifying Models

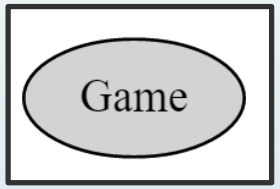
Build a dependency graph starting with a root object

Break down the root into smaller pieces

How do we represent the entire game?

Define a Game type

Identifying Models



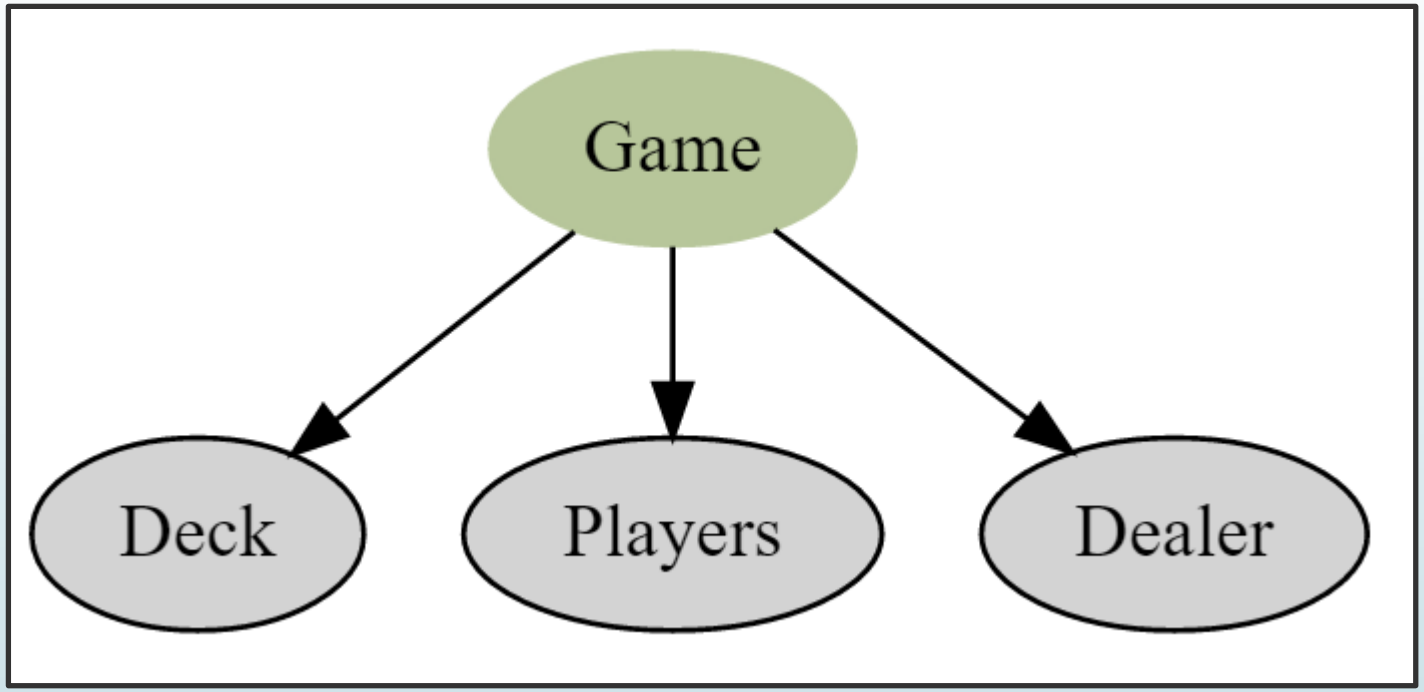
Identifying Models

What's included in a game?

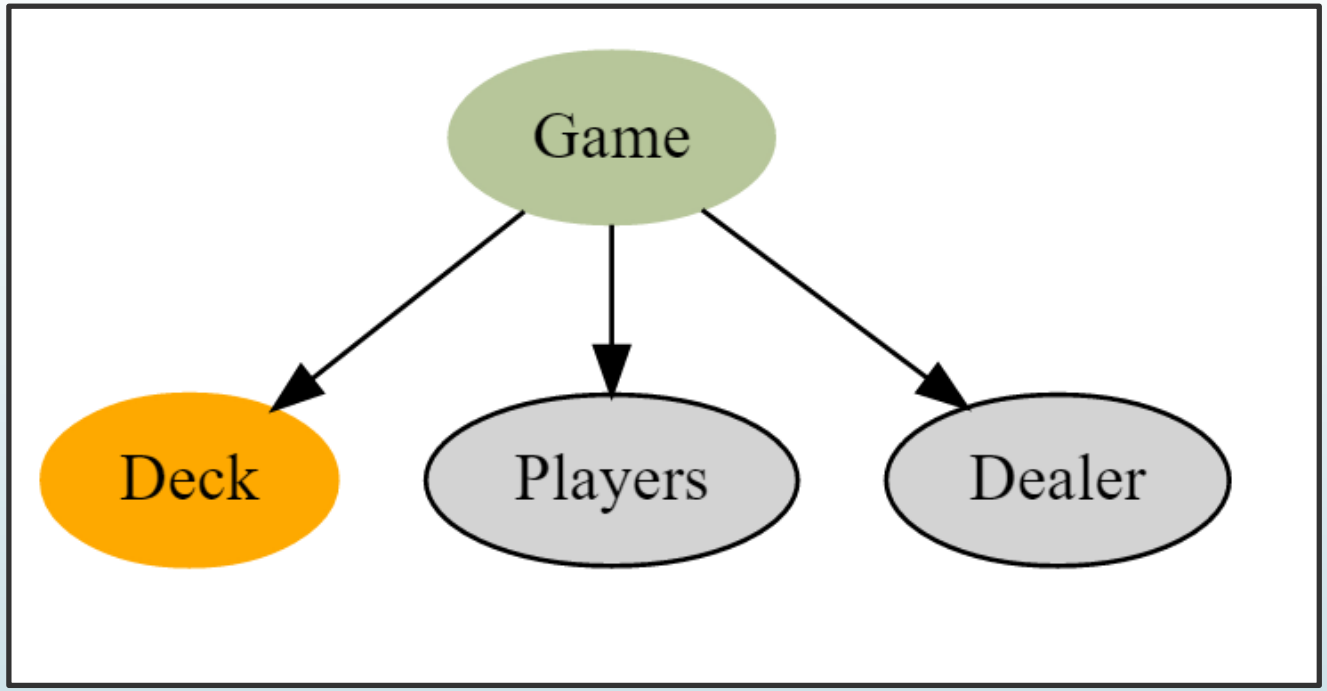
Game has

- Deck
- Players
- Dealer

Identifying Models



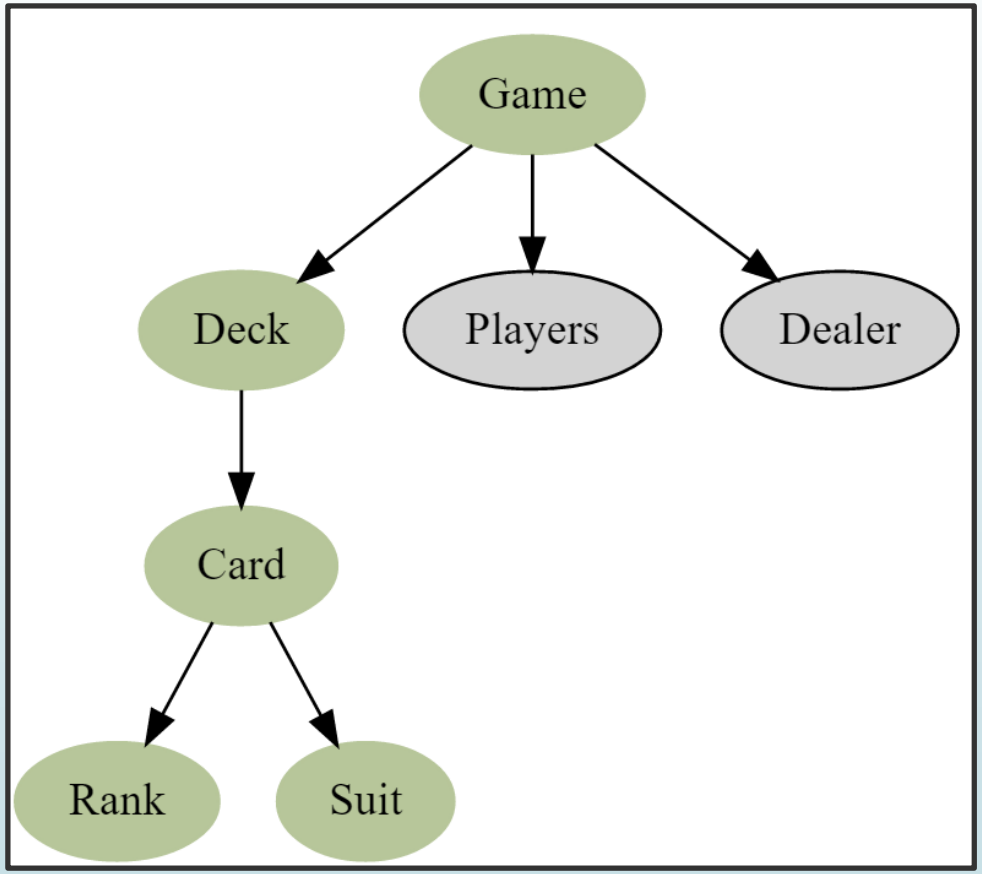
Identifying Models



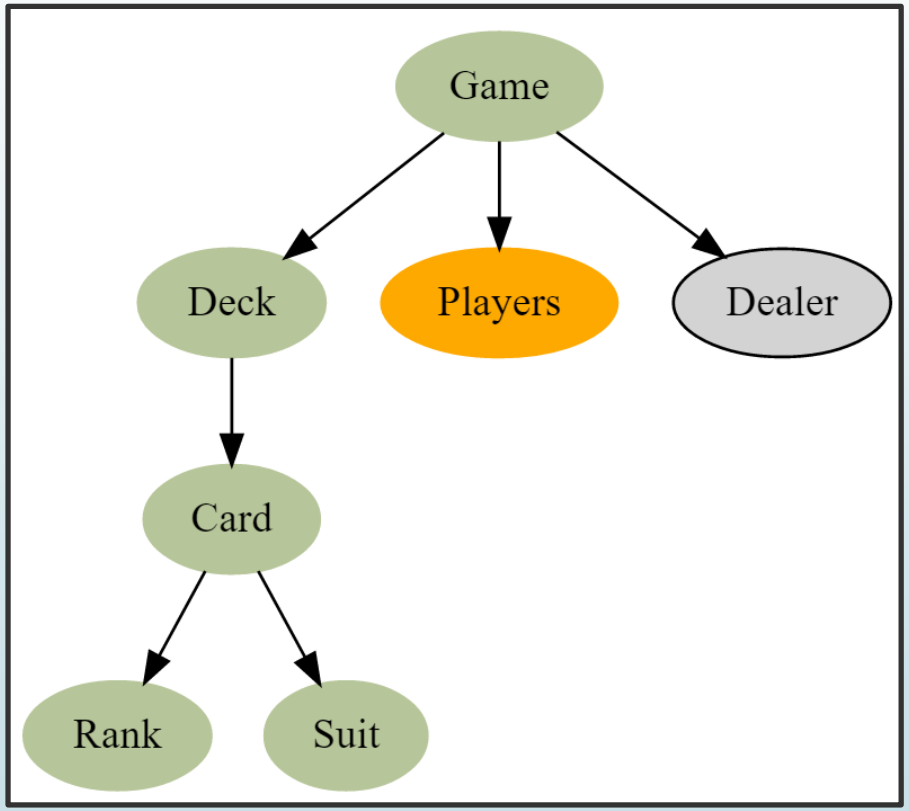
Identifying Models

- What's a Deck?
  - Represents all of the Cards that will be used in the Game
- What's a Card?
  - Combination of Rank and Suit
- What's a Rank
  - One of 13 possible values
  - A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K
- What's a Suit
  - One of 4 possible values
  - Hearts, Clubs, Spades, Diamonds

Identifying Models



Identifying Models



Identifying Models

What are Players?

- One or more Player

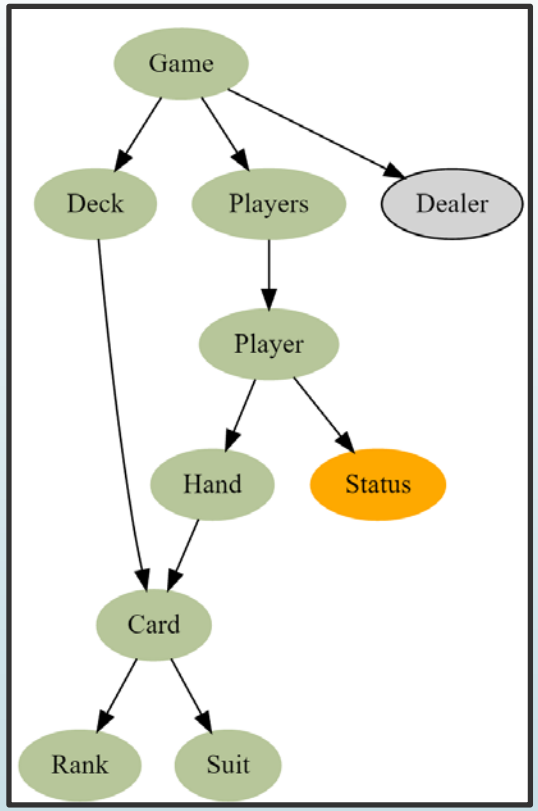
What's a Player

- Someone playing against the Dealer
- Has both a Hand and a Status

What's a Hand?

- Represent the Cards a Player has

Identifying Models

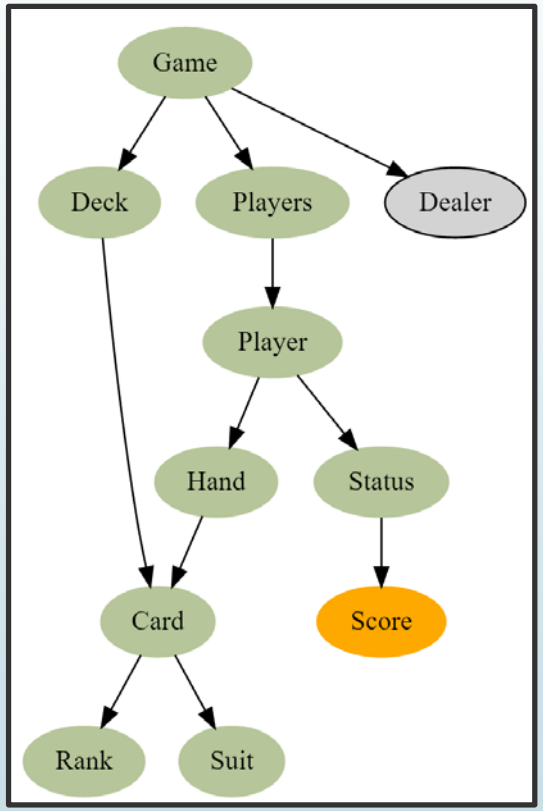


Identifying Models

Represents the state of the Player

- Blackjack -> 2 cards worth with 21
- Busted -> Has a Score over 21
- Stayed -> Not Blackjack and a Score of 21 or less
- Cards Dealt -> Hasn’t taken a turn yet

Identifying Models



Identifying Models

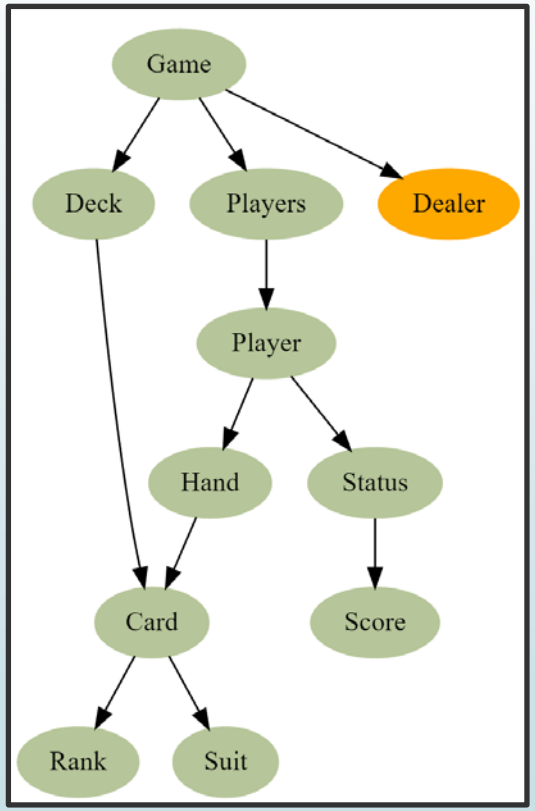
Represents the final result of a Player's Hand

Result is based on whether the Player Busted or Stayed

Can only be of one value (integer)

Important to know when comparing who won

Identifying Models



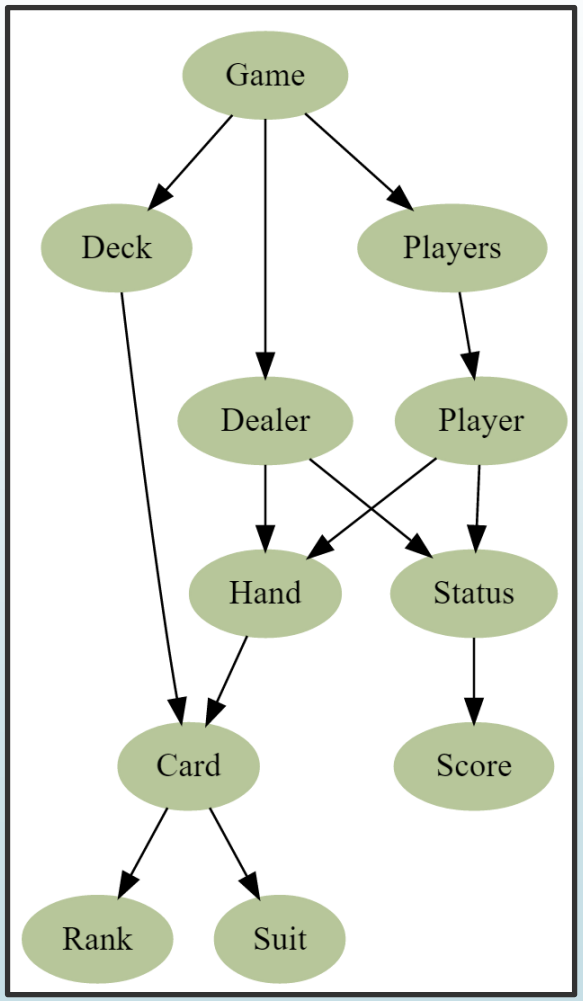
Identifying Models

Competes against every Player

Similar to Player as a Dealer has a Hand and a Status

Unlike Player, there’s only one

Identifying Models



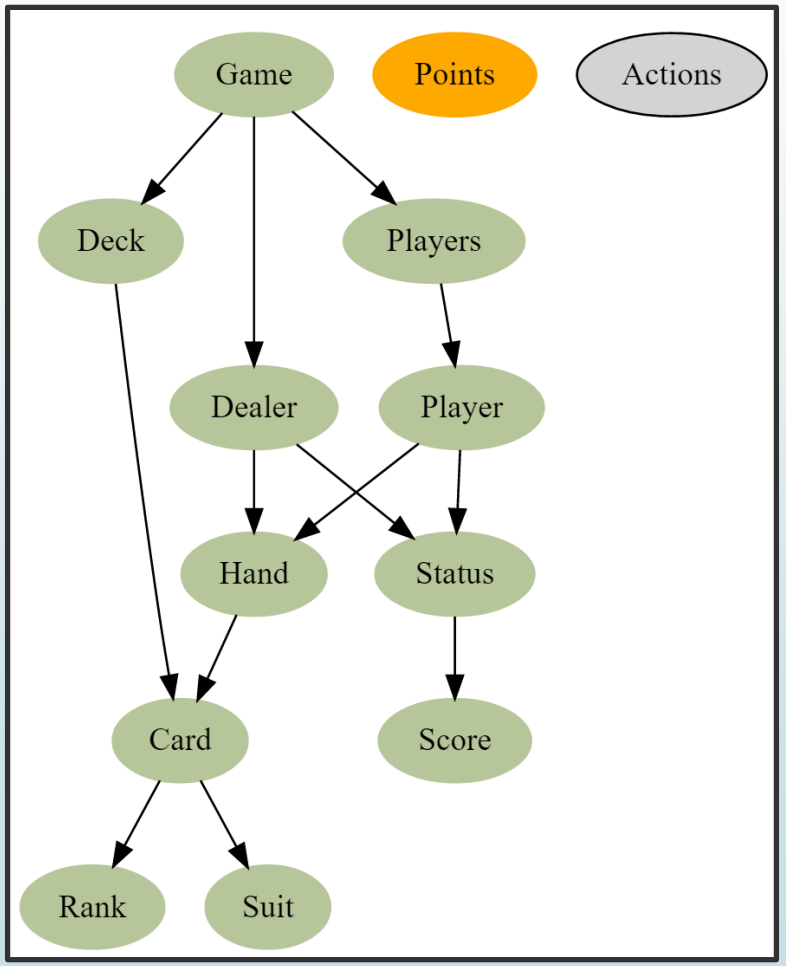
Identifying Models

Talked about the components of the game

What about points?

What about the actions that a player can take?

Identifying Models



Identifying Models

Cards are worth Points based on their Rank

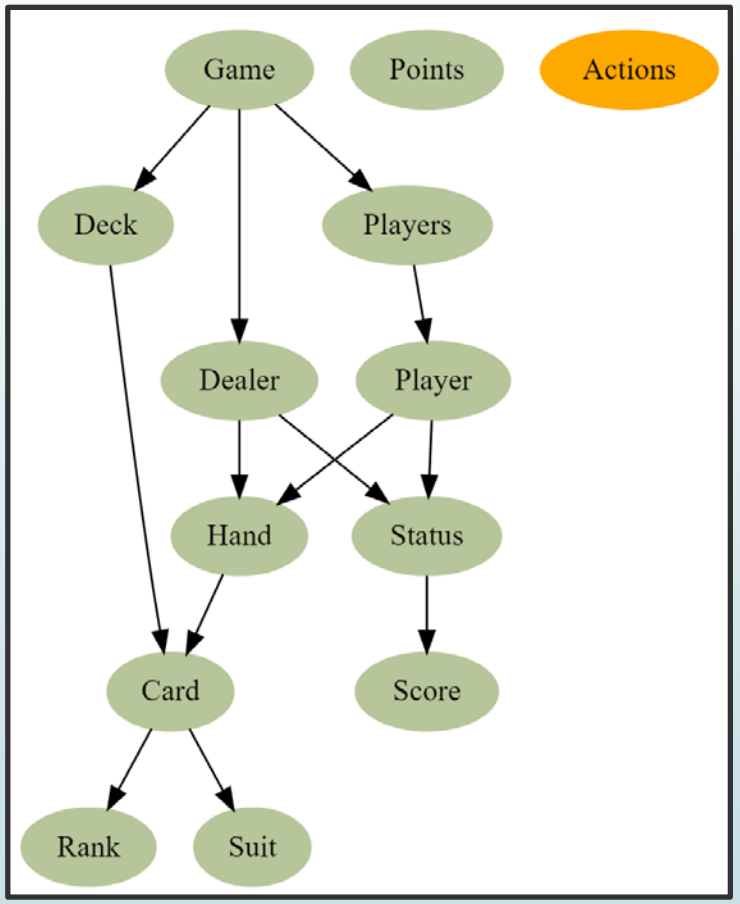
Face cards are worth 10 points

Point cards are worth that many

- (2's worth 2, 3's worth 3 …, 10's worth 10)

Aces can be counted as 1 or 11

Identifying Models



Identifying Models

A participant has two possible actions

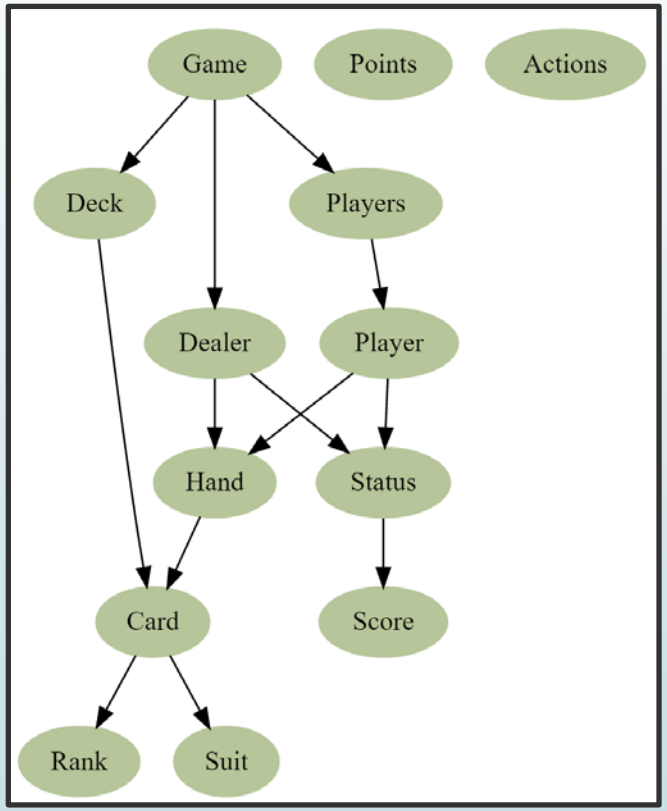
Draw Cards (Hit)

- Adds a single card to their Hand

Stay

- Stops drawing cards

Identifying Models



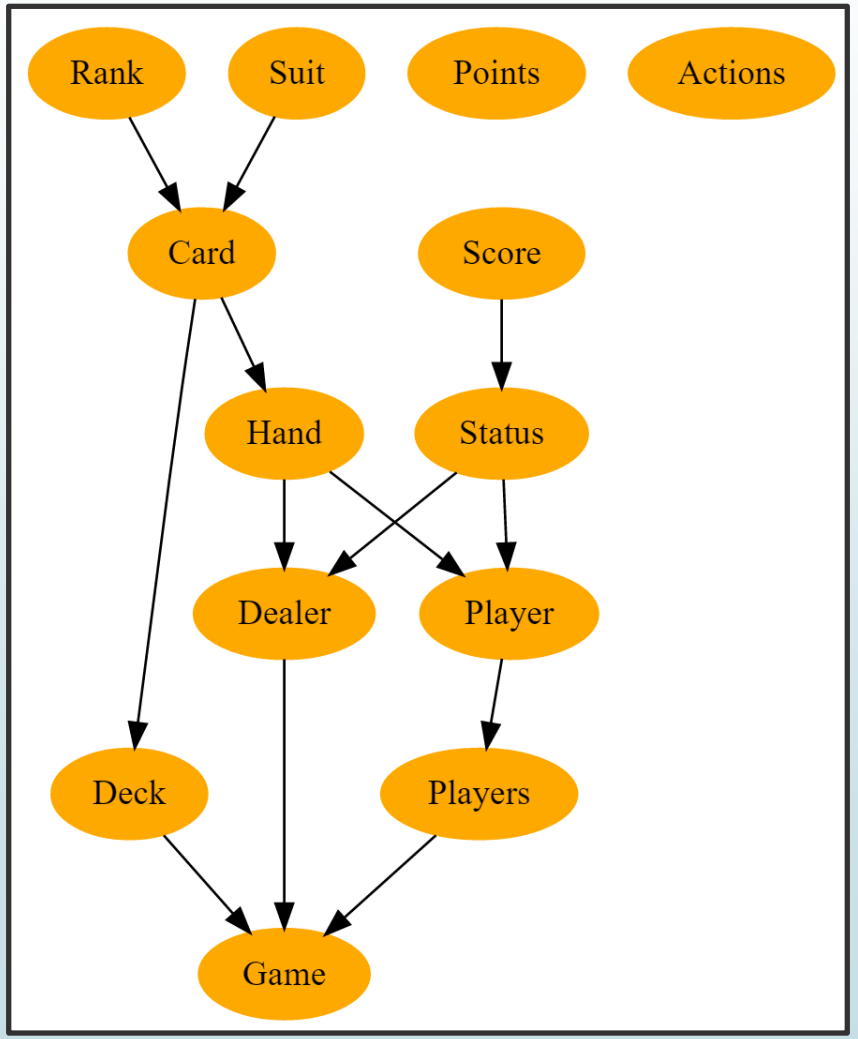
Implementing the Domain

Broke down the domain from the top (Game) to smaller pieces

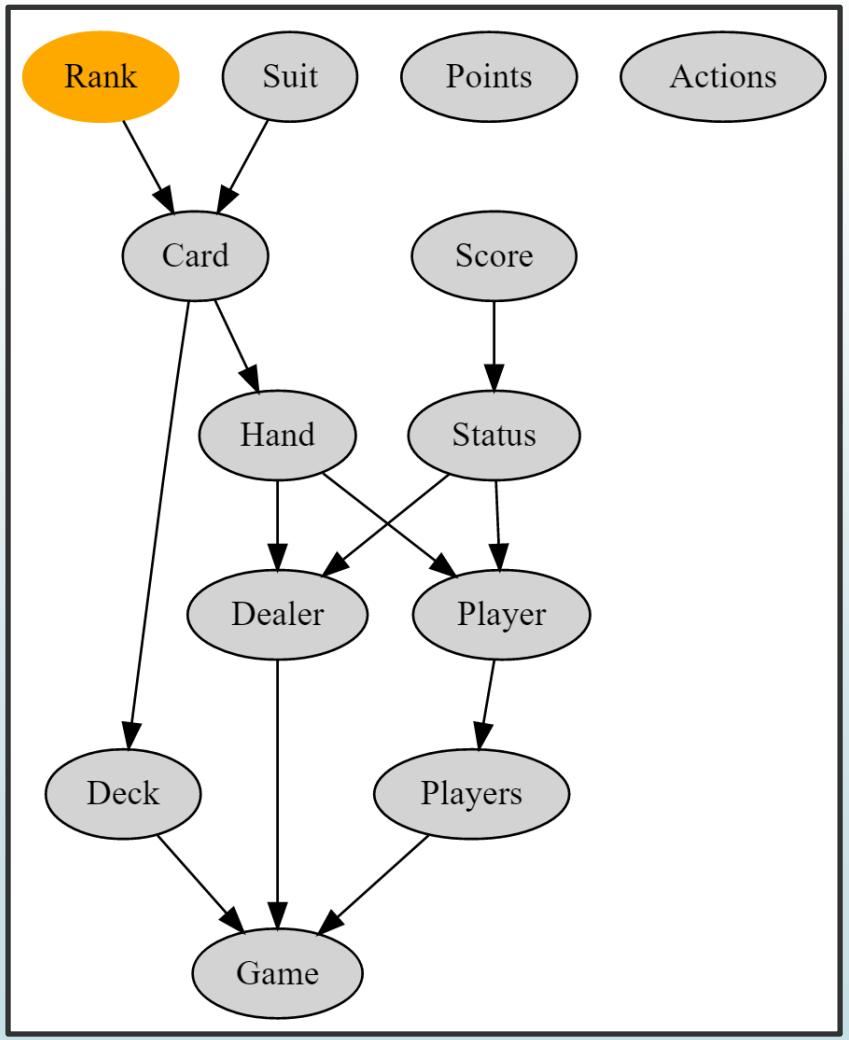
Created a dependency graph

Time to implement the pieces in a bottom-up order

Implementing the Domain



Implementing the Domain



Implementing the Models
Rank

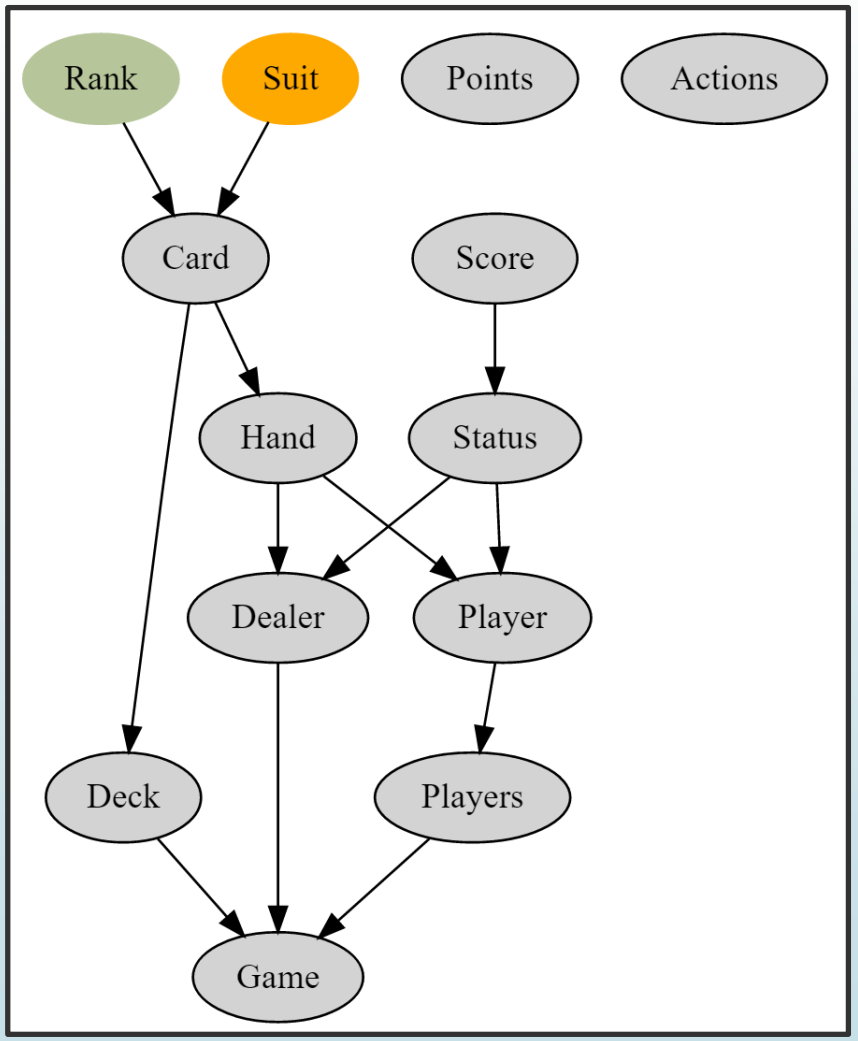
Can be one of 13 possible values

- A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K

What type is great for modeling a finite set of values?

```
1: type Rank = Ace | Two | Three | Four | Five | Six | Seven
2:              Eight | Nine | Ten | Jack | Queen | King
```

Implementing the Domain



Implementing the Models
Suit

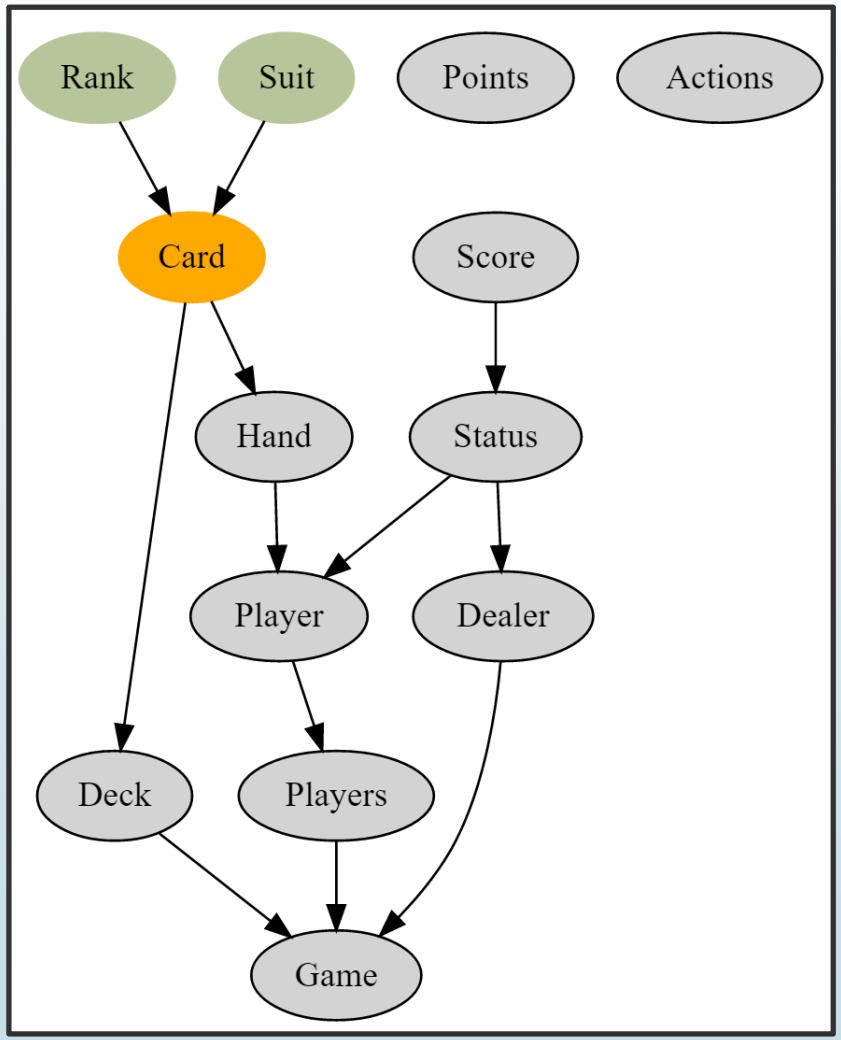
Can be one of 4 possible values

- Hearts, Clubs, Spades, Diamonds

How should we model this?

```
1: type Suit = Hearts | Clubs | Spades | Diamonds
```

Implementing the Domain



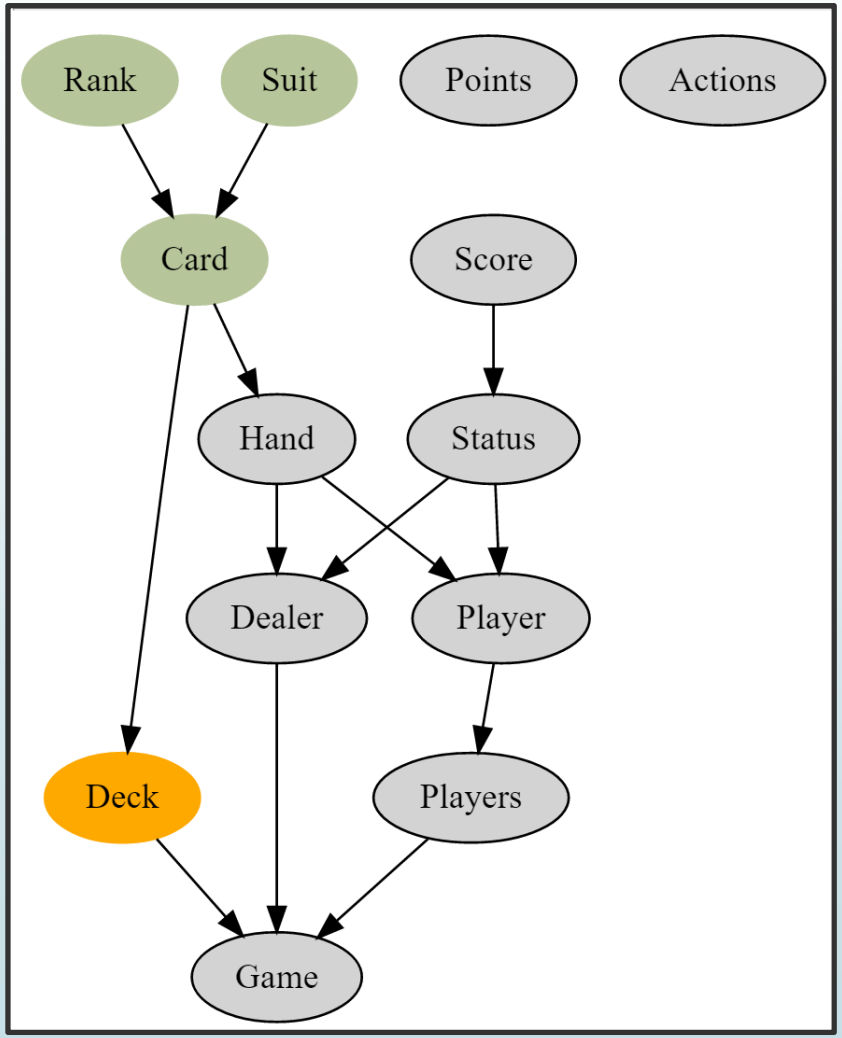
Implementing the Models
Card

Contains both a Rank and Suit

What type is great for holding data?

```
1: type Card = {Rank:Rank; Suit:Suit}
```

Implementing the Domain



Implementing the Models
Deck

Contains 1 or more Cards

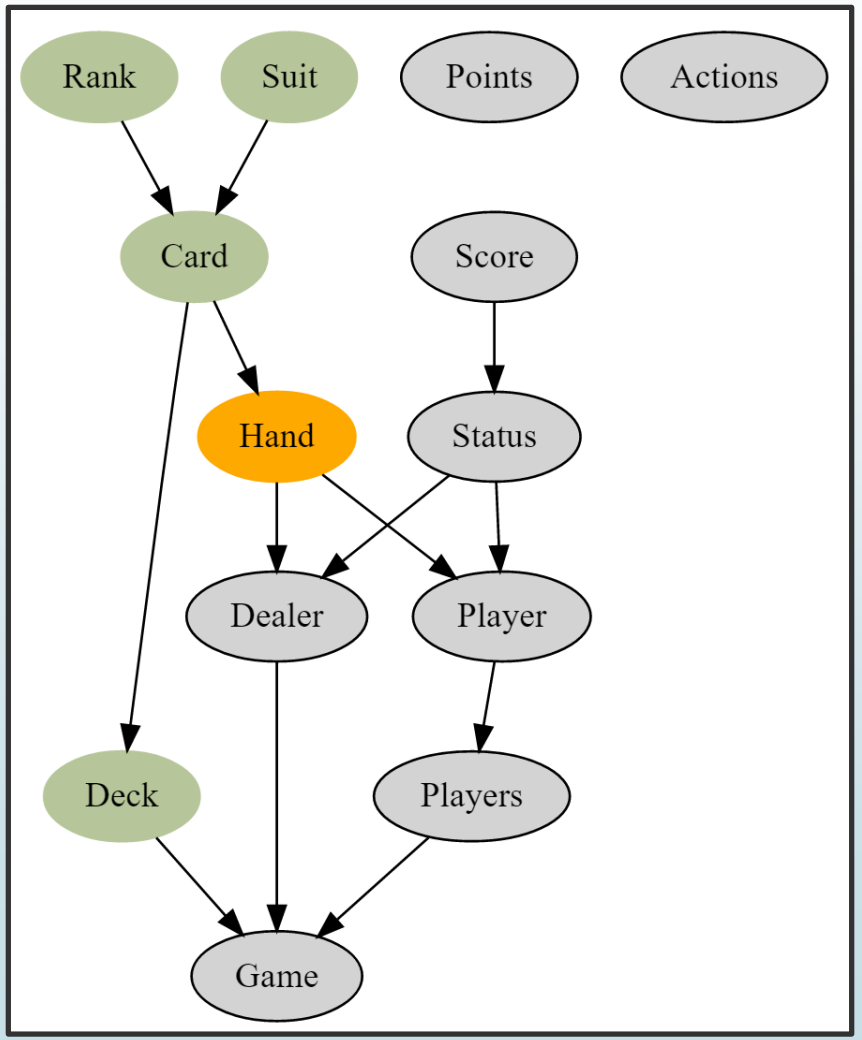
What are some properties of a Deck?

- Length will change throughout the game

Which collection type should we use to model this?

```
1: type Deck = Card list
```

Implementing the Domain



Implementing the Models
Hand

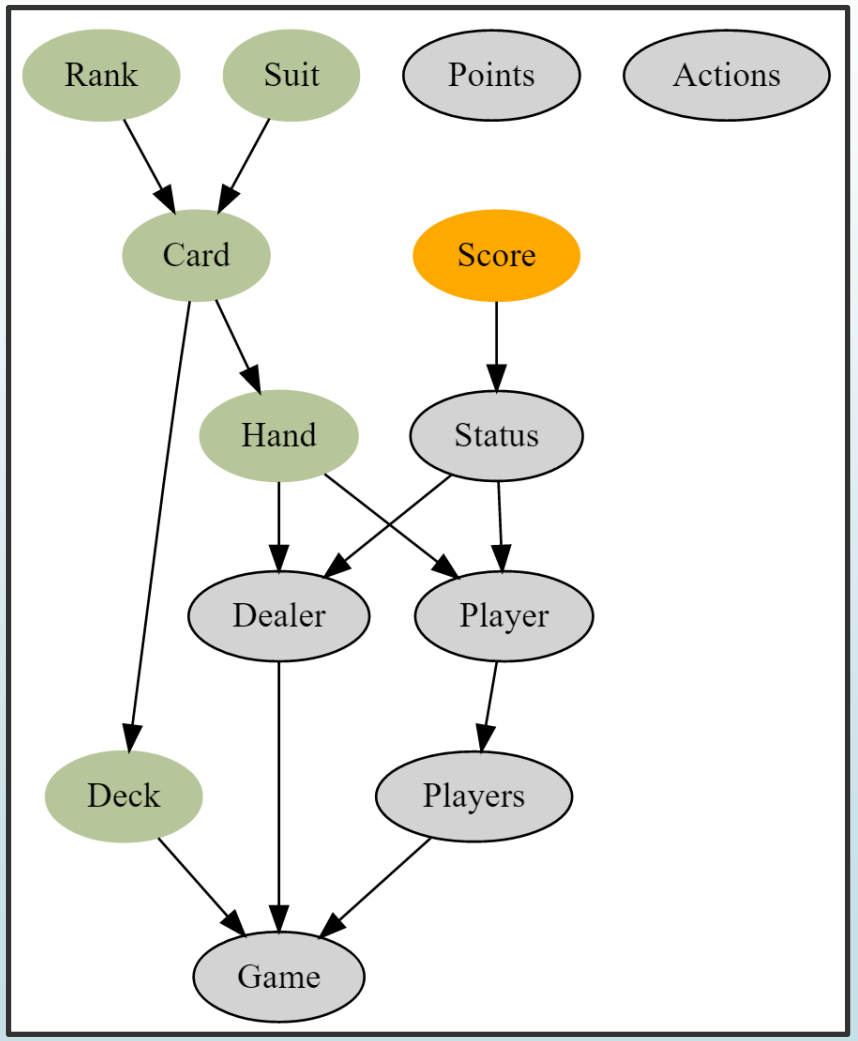
Contains the Cards for a Player

Initially starts with two cards, but can draw more during the game

How should we model this?

```
1: type Hand = Card list
```

Implementing the Domain



Implementing the Models
Score

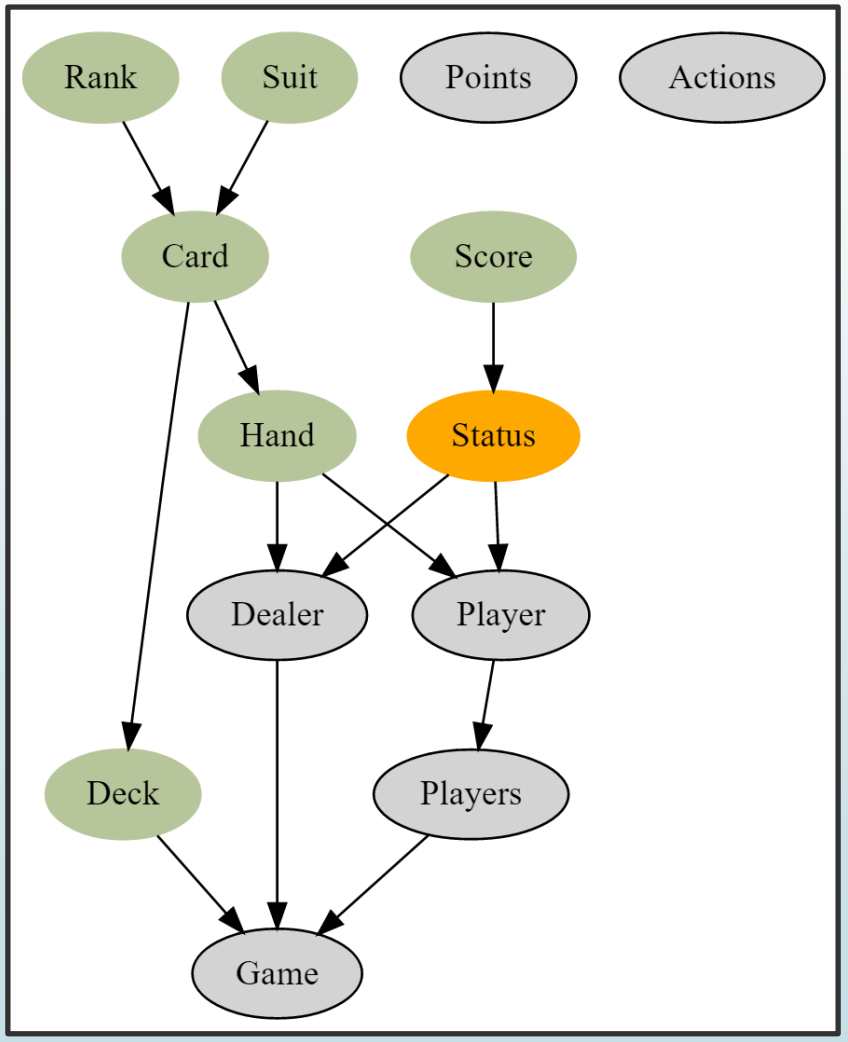
Represents the final value of a Hand

Can only be one value (integer)

What's a way to wrap primitives as their own type?

```
1: type Score = Score of int
```

Implementing the Domain



Implementing the Models Status

Blackjack

- Player has two cards and their score is 21

Busted

- Player went over 21 and that's their final score

Stayed

- Player decided to not take any more Cards and that's their final score

Cards Dealt

- Player hasn't taken their turn

Implementing the Models
Status

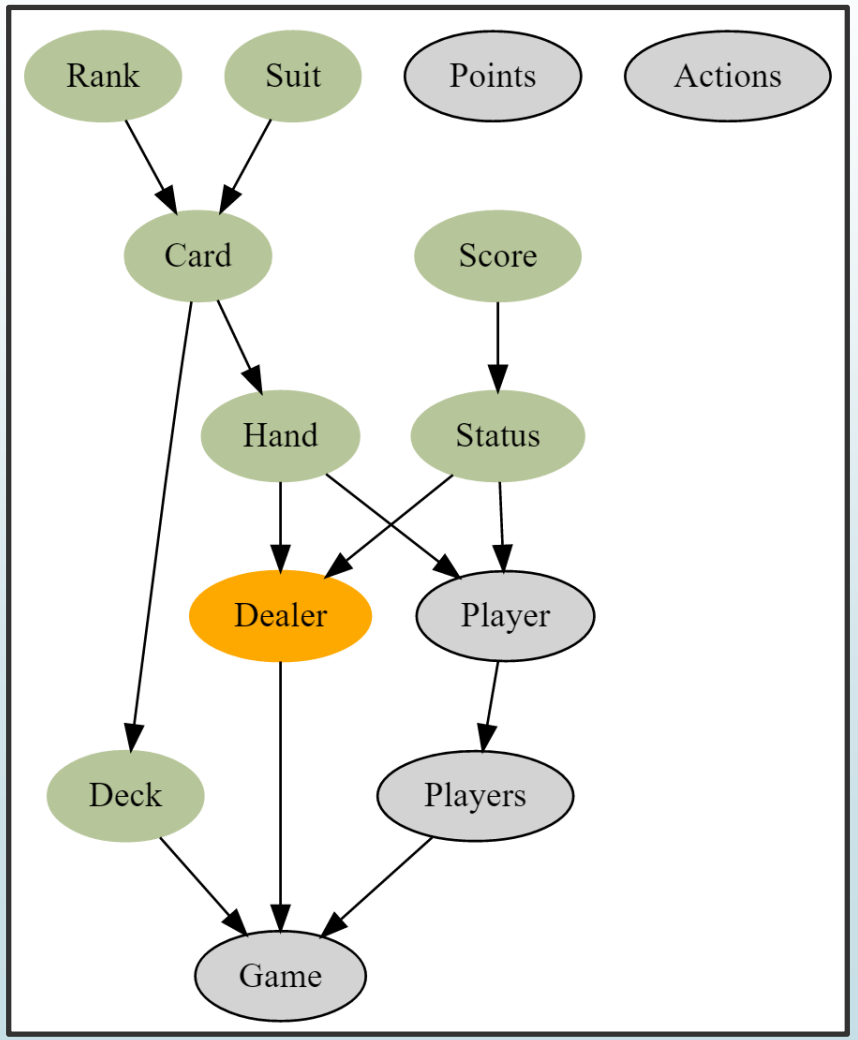
Can be one of a finite number of choices

Some of the type require Score and other ones don't

How should we model this?

```
1: type Status = Blackjack | Busted of Score
2:               | Stayed of Score | CardsDealt
```

Implementing the Domain



Implementing the Models
Dealer

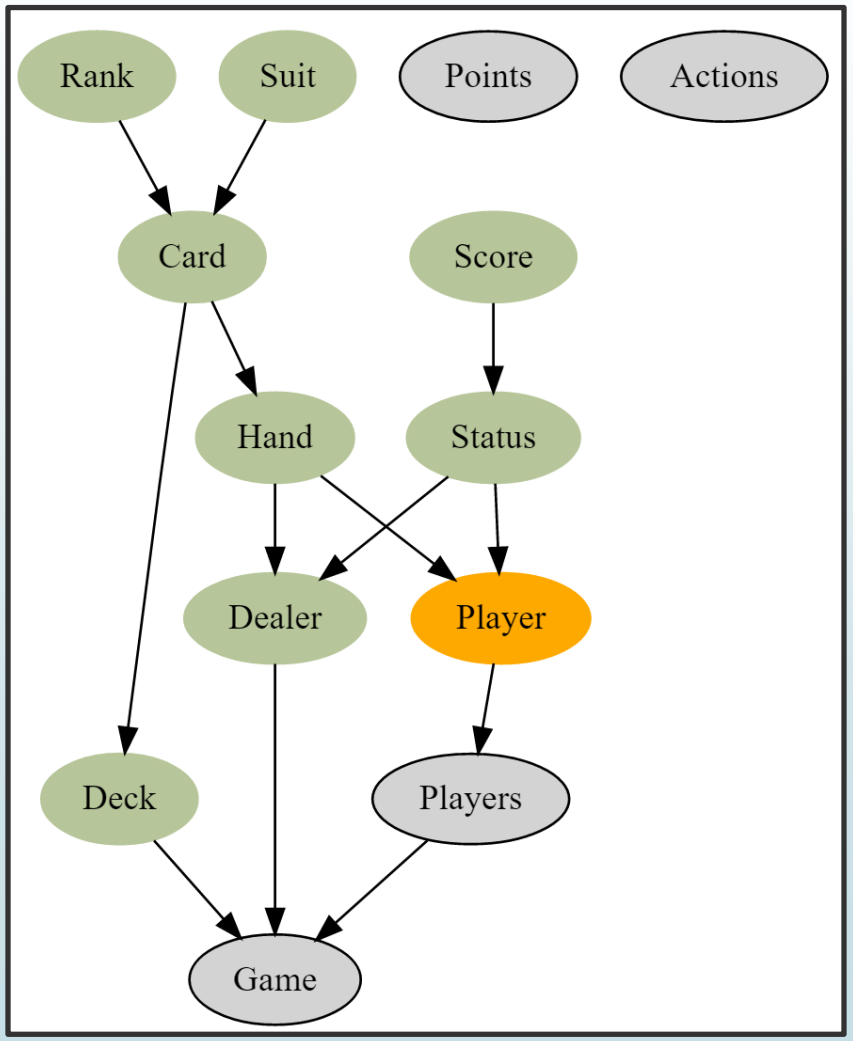
Dealer contains

- Hand
- Status

How should we model this?

```
1: type Dealer = {Hand:Hand; Status:Status}
```

Implementing the Domain



Implementing the Models
Player

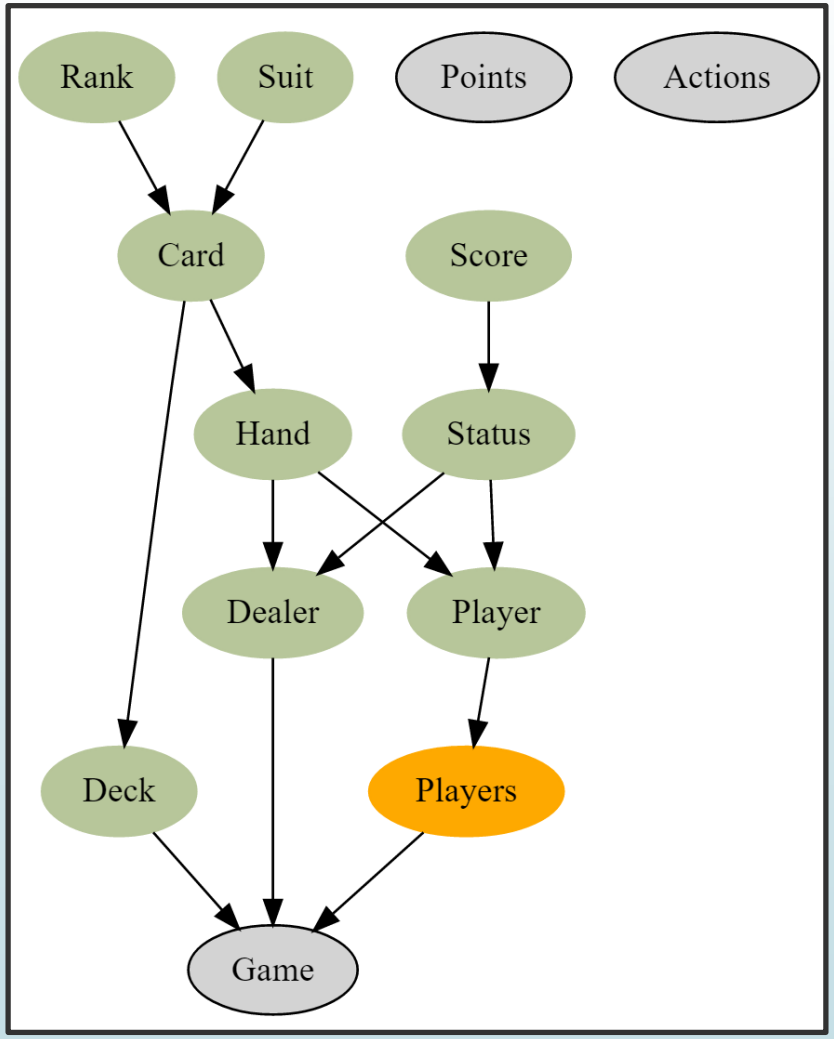
A Player contains

- Hand
- Status
- ID

How should we model this?

```
1: type Player = {Hand:Hand; Status:Status; Id:int}
```

Implementing the Domain



Implementing the Models
Players

Represents all the players in the Game

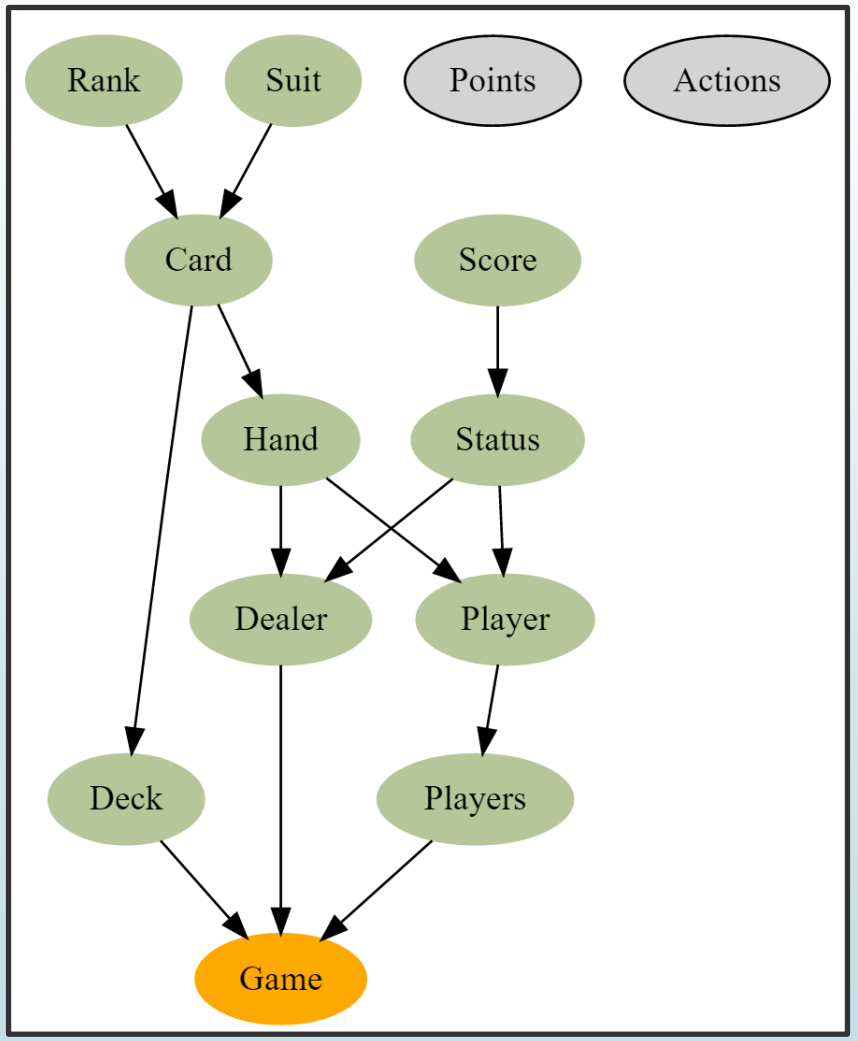
Number of Players won't change during the game

Each player must finish their turn before the next player can start

Which collection type should we use?

```
1: type Players = Player list
```

Implementing the Domain



Implementing the Models
Game

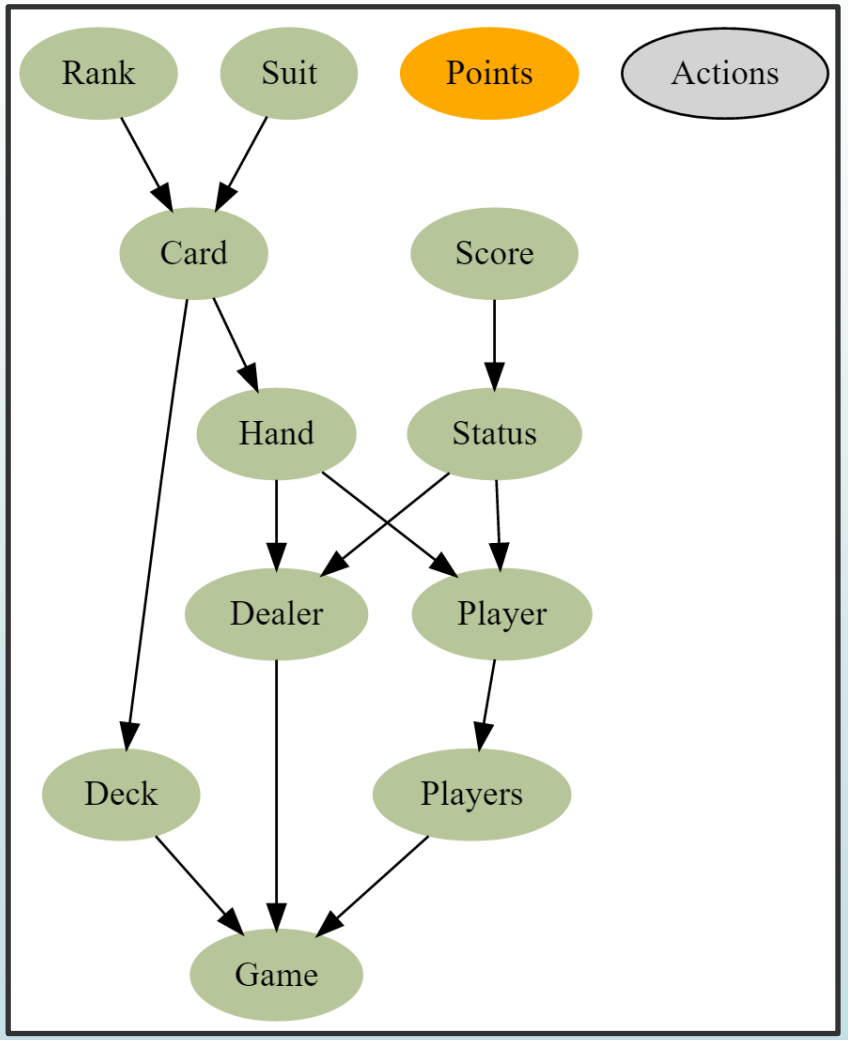
A Game has

- Deck
- Players
- Dealer

How should we model this?

```
1: type Game = {Deck:Deck; Dealer:Dealer; Players:Players}
```

Implementing the Domain



Implementing the Models
Points

Every Card is worth a value based on the Rank

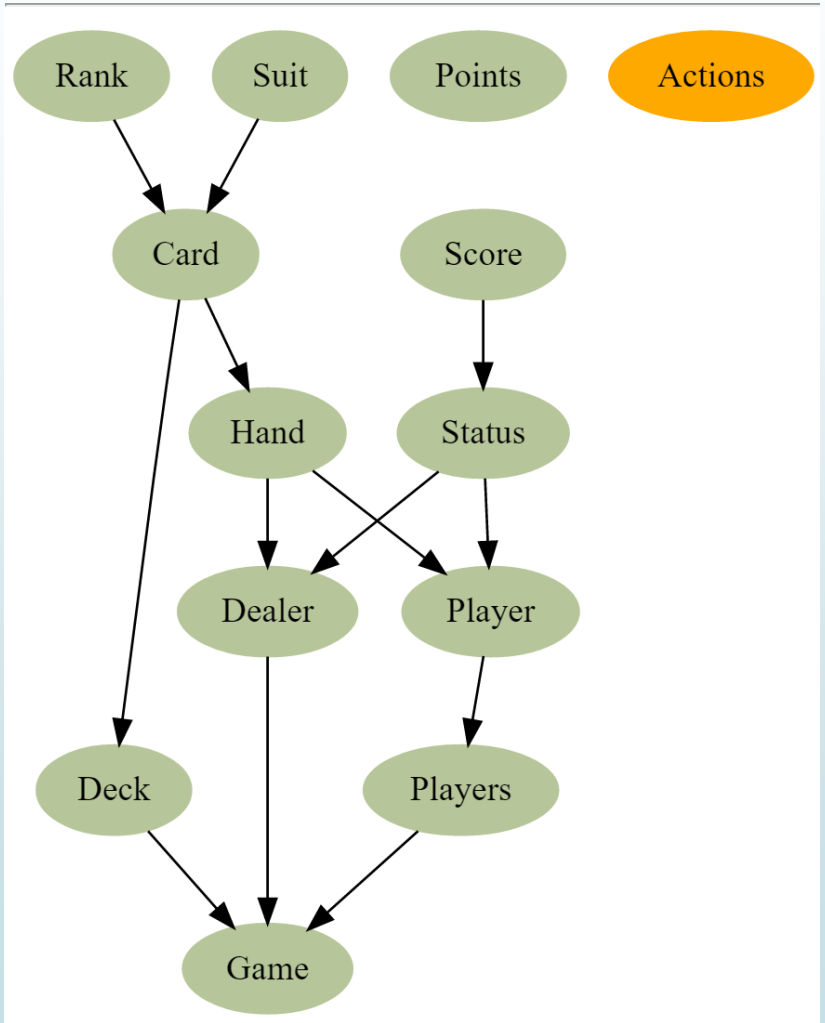
One of two possible values

- Aces can be worth 1 or 11
- Everything else can be worth one value

How to combine two different types as one?

```
1: type Points = Hard of int | Soft of int*int
```

Implementing the Domain



Implementing the Models
Actions

Represents what a participant can do during the Game

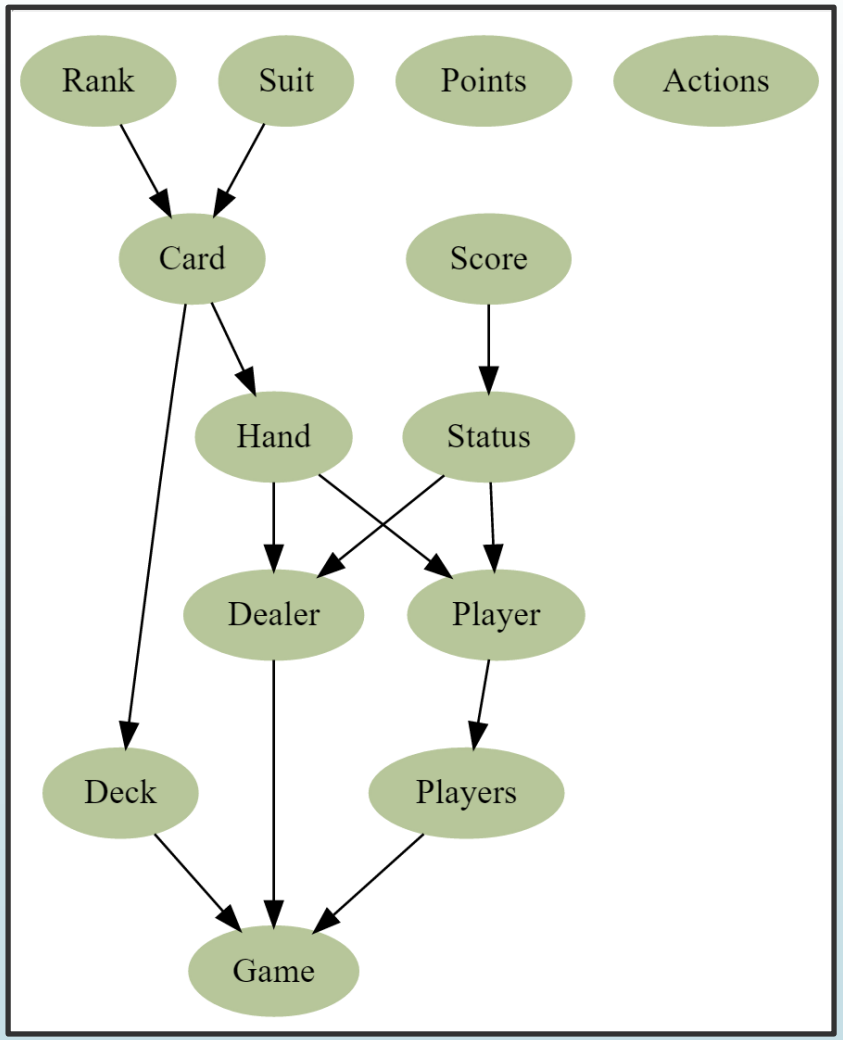
One of two possible values

- Hit
- Stay

Finite number of choices, how to model?

```
1: type Actions = Hit | Stay
```

Implementing the Domain



Full Implementation

```
 1: type Rank = Ace | Two | Three | Four | Five | Six | Seven
 2:              Eight | Nine | Ten | Jack | Queen | King
 3: type Suit = Hearts | Clubs | Spades | Diamonds
 4: type Card = {Rank:Rank; Suit:Suit;}
 5: type Points = Hard of int | Soft of int*int
 6:
 7: type Deck = Card list
 8: type Hand = Card list
 9: type Score = Score of int
10: type Status = Blackjack | Busted of Score
11:                | Stayed of Score | CardsDealt
12: type Dealer = {Hand:Hand; Status:Status}
13: type Player = {Hand:Hand; Status:Status; Id:int}
14: type Players = Player list
15:
16: type Actions = Hit | Stay
17: type Game = {Deck:Deck; Dealer:Dealer; Players:Players}
```

Summary

- Determined the models by using domain terms (ubiquitous language)
- Started with the biggest model (Game) and broke it down into smaller pieces and then repeating the process with the smaller pieces
- After finding the models, implemented the models in a bottom-up fashion
- Able to define the domain in a way such that bad states are unrepresentable

Playing the Game

Playing the Game

Sometimes, pure functions can fail

How can we handle this?

We'll look at

- Drawing a Card
- Setting up a Player

Playing the Game
Drawing a Card

Let's create a function, **drawCard** that takes a Deck as input and returns the top Card and the rest of the Deck as output

```
1: let drawCard deck =
2:     match deck with
3:     | topCard::restOfDeck -> (topCard,restOfDeck)
```

Drawing a Card

```
1: let drawCard deck =
2:    match deck with
3:    | topCard::restOfDeck -> (topCard,restOfDeck)
```

Are there any issues with this implementation?

What about an empty deck?

- ➨ **Match Failure Exception -> The match cases were incomplete**

How can we model a type that may have a value or not?

Playing the Game
Drawing a Card

Introducing the Option type!

```
1: type Option<'a> = Some of 'a | None
```

Let's rewrite the drawCard function to handle an empty deck

Playing the Game
Drawing a Card

```
1: let drawCard deck =
2:   match deck with
3:   | [] -> None
4:   | topCard::restOfDeck -> Some (topCard,restOfDeck)
```

Playing the Game
Setting up a Player

Now that we have a way to draw cards, we can now setup a Player

Remember that a Player starts off with a Hand of two cards and a Status of CardsDealt

```
1: let setupPlayer drawCard id deck =
2:     let firstCard,deck = drawCard deck
3:     let secondCard,deck = drawCard deck
4:     let hand=[firstCard;secondCard]
5:
6:     {Hand=hand; Id=id; Status=CardsDealt},deck
```

Playing the Game
Setting up a Player

Remember, **drawCard** returns an option, so we need to handle that.

```
1: let setupPlayer drawCard id deck =
2:   match drawCard deck with
3:   | None -> None
4:   | Some(firstCard, deck) ->
5:     match drawCard deck with
6:     | None -> None
7:     | Some(secondCard, deck) ->
8:       let hand = [firstCard; secondCard]
9:       Some ({Hand=hand; Id=id; Status=CardsDealt},deck)
```

Playing the Game
Setting up a Player

Try to draw two cards, if so, return Some Player, otherwise, None

Wouldn't it be nice if we could have the short-circuit logic of the second solution, but still have the readability of the happy path?

How can we work around all the pattern matching?

Playing the Game
Setting up a Player

By using the Maybe Builder pattern!

Known in other FP languages as Maybe.

Two parts

- Provides a way to short-circuit if the input is None (called Bind)
- A way to wrap a value as an option (called Return)

Playing the Game
Setting up a Player

```
1: type MaybeBuilder() =
2:     member this.Bind(input, func) =
3:         match input with
4:         | None -> None
5:         | Some value -> func value
6:
7:     member this.Return value =
8:         Some value
```

Playing the Game
Setting up a Player

```
 1: let setupPlayer drawCard id deck =
 2:    let maybe = new MaybeBuilder ()
 3:
 4:    maybe {
 5:       let! firstCard,deck = drawCard deck
 6:       let! secondCard,deck = drawCard deck
 7:       let hand=[firstCard; secondCard]
 8:
 9:       return {Hand=hand; Id=id; Status=CardsDealt},deck
10:    }
```

```
1: let setupPlayer drawCard id deck =
2:   let firstCard,deck = drawCard deck
3:   let secondCard,deck = drawCard deck
4:   let hand=[firstCard;secondCard]
5:
6:   {Hand=hand; Id=id; Status=CardsDealt},deck
```

```
 1: let setupPlayer drawCard id deck =
 2:   let maybe = new MaybeBuilder ()
 3:
 4:   maybe {
 5:     let! firstCard,deck = drawCard deck
 6:     let! secondCard,deck = drawCard deck
 7:     let hand=[firstCard; secondCard]
 8:
 9:     return {Hand=hand; Id=id; Status=CardsDealt},deck
10:   }
```

Playing the Game
Setting up a Player

Problem

- Creating a Player can fail because there's not enough cards
- Putting in the error handling code makes it cluttered and hard to read

Solution

- Using the Maybe pattern, we can short-circuit our function flow to return None before calling anything else

When to Use

- Lots of nested pattern matching on options

Wrapping Up

Overview of the different F# types and the advantages of each

Modeled and implemented the domain for Blackjack using the different types

Explored how to handle functions that can fail

Additional Resources

Learning F#:

- F# for Fun and Profit
- *The Book of F#* by Dave Fancher
- Mark Seeman's blog (https://blog.ploeh.dk)

Learning Functional Programming

- Reid Evans YouTube Channel

Thanks!

Email: Cameron@TheSoftwareMentor.com

Twitter: @PCameronPresley

Blog: http://blog.TheSoftwareMentor.com