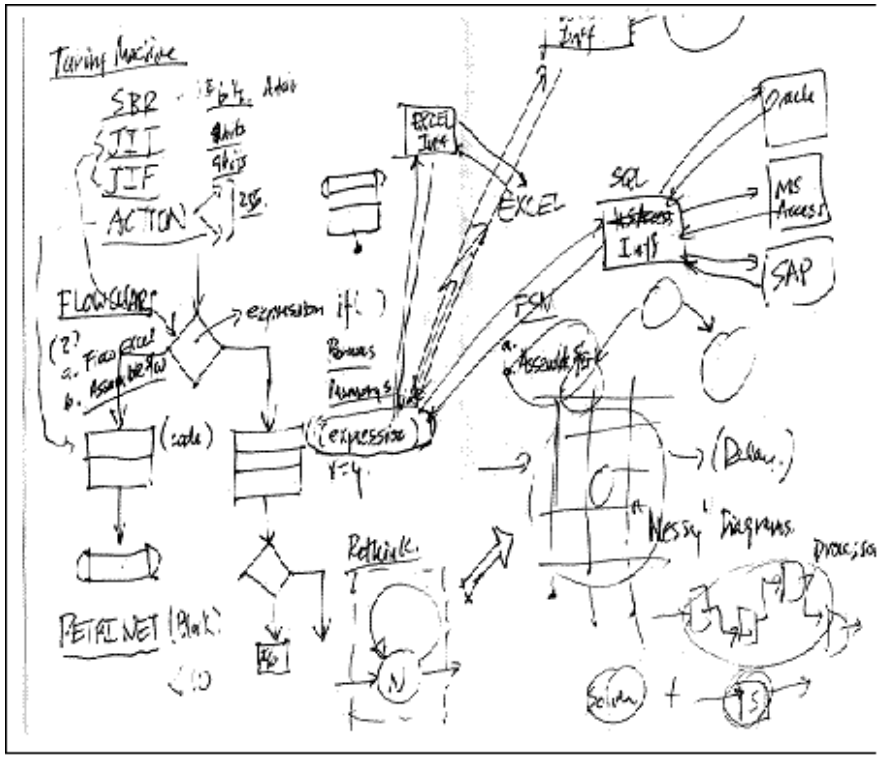
How to Interview (Object Oriented Design)

Daniel Kirschner Follow
Nov 11, 2016 · 12 min read



This is part three of my four part series on interviewing. In it, I talk about the four major types of questions recent college graduates with computer science degrees will see. Rather than focusing on the rote details of how merge sort works, or when its appropriate to use Djkstra's, we'll talk about the meta-process to follow when solving each type of question. Combining this with the knowledge you already have from your courses will give you the confidence to crush anything companies throw at you.

The first two types of questions we looked at fit right into the college mold. Algorithms was probably an explicit class you took, while most of your projects hopefully required Coding. Paying attention in class and reading my previous posts should be enough to get you through anything.

Object Oriented Design (OOD), however, is a little more interesting. Colleges generally don't offer explicit classes in design, or at least my school didn't, which means that students are stuffed full of textbook knowledge on an interfaces and inheritance, but still don't understand how to apply it in practice. In the past couple of weeks alone I've asked more than a handful of these questions and gotten blank stares back from the candidate. Its a shame, because OOD questions are a fun way to explore different choices and see how they impact your system, not just now, but in the future as well!

Background

OOD questions generally all have the same beginning. There is an intentionally vague set of initial constraints, and then a moment of awkward silence while the candidate mentally screams. In contrast to most typical interview questions, there isn't really a *right* answer that the interviewer is looking for. Really these problems are about one thing: can the candidate articulate and handle the trade offs between "needs to work now" and "needs to handle changes later".

The approach

Here are the steps you need to follow when answering a design problem.

- Find the right level
- Talk through use cases
- Find the objects
- Wire the relationships

Find the right level

A typical way I'll start an OOD question is to ask a pretty generic, open ended question like this:

> *Let's design a program to play a game of cards like BlackJack.*

The very first thing you need to do as a candidate is figuring out what output the interviewer is looking for. If it wasn't made clear in the question itself, don't be afraid to ask! "Are we going to be implementing

the design here, or are you just looking for an overall class structure?" Anything to do with implementation and its probably worth spending brain cycles on details like specific data structures. If they seem to be looking for the high level relationships, don't get sucked into the details!

Talk through use cases

First things first, do you know how to play blackjack? If not, now is the time to ask for details. Describing the problem is an important first step in design, because this is where the majority of your use cases will be fleshed out. For example, here is how I would describe the game of blackjack.

What you need:
A game of blackjack is played with a deck of cards, a dealer, and any number of players.

How to play:
Each round the dealer deals 2 cards to each player, and two cards to himself.
The player can draw more cards, or stop drawing at any point.
If the players score goes over 21, or is less than the dealers score, they lose.
At the end of every round the cards are put back into the deck and shuffled.

You'll notice that these are all relatively high level statements. That is intentional. I've had candidates get to this stage and immediately nose dive into specific scoring rules like splitting, doubling down, how to handle ties, etc. Its like meeting with a house builder and continually focusing on what color the tiling will be in the upstairs bathroom. Its awesome you are thinking ahead, but, seriously, lets get the blueprints drawn first.

Here is the key part: don't just think about your use cases, physically write them on the white board. This does two things. First, it sets a contract between you and the interviewer on what the important parts of the design are. As new use cases are introduced we will be updating this list. Later we can use it to double check that our design meets the requirements. Secondly, its an incredibly easy way to identify the right objects.

Find the objects

The cool thing about OOD is that its designed to mimic the way humans naturally express relationships. Look back at the sentences we wrote above. Each noun we wrote can be thought of as an object or property that we need to represent. Each verb is a behavior that interacts with these objects.

That in mind, what are the nouns we need to model? In order of first appearance, the ones I see are:
Deck
Cards
Dealer
Player
Score

The behaviors we are going to need to model are:
Dealing cards
Drawing cards
Shuffling cards
Calculating score

This is enough for us to get started on the design. As we add use cases later this list will expand to accommodate.

Wire the relationships together.

I like to solve problems like this bottom up and describe the smallest set of self contained behaviors I can before working up to the larger pieces. Working top down is much more difficult, because you end up trying to do things with behaviors you haven't defined yet. If that makes more sense to you, though, just… uhhh… try to read this section backwards.

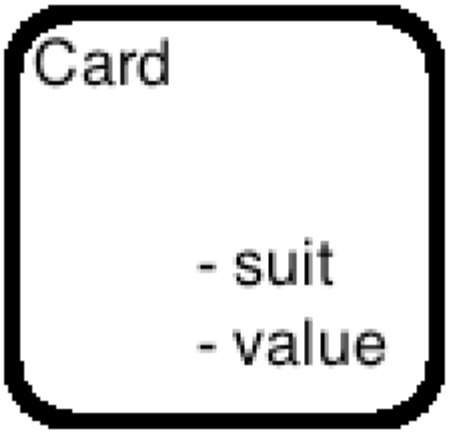
Cards

Just like in real life, individual `Cards` should probably have a suit (Spades, Clubs, Diamonds, Hearts) and value (2–10, Jack, Queen, King, Ace) so we can add those as properties. Hmm, one of the behaviors we saw earlier talked about scoring. Should `Card` s have a score? Lets think about it. An easy way to calculate scores would be to give each individual `Card` a score, and then add the scores for the cards in the players hand. It certainly works right now!

Before we really get invested in that though, lets take a step back. The common pitfall that junior developers fall into here is confusing “makes sense for this use case” with “makes sense for the design”. Think about a generic pack of cards you might buy in a store. If you opened it up and held a 9 of Diamonds in one hand and a Jack of Spades in the other, which one would score higher? Well… neither. The score for each card is determined by the game you are playing and how the cards interact with each other. Its not something that is intrinsic to the cards themselves.

I don’t see a score attached here anywhere

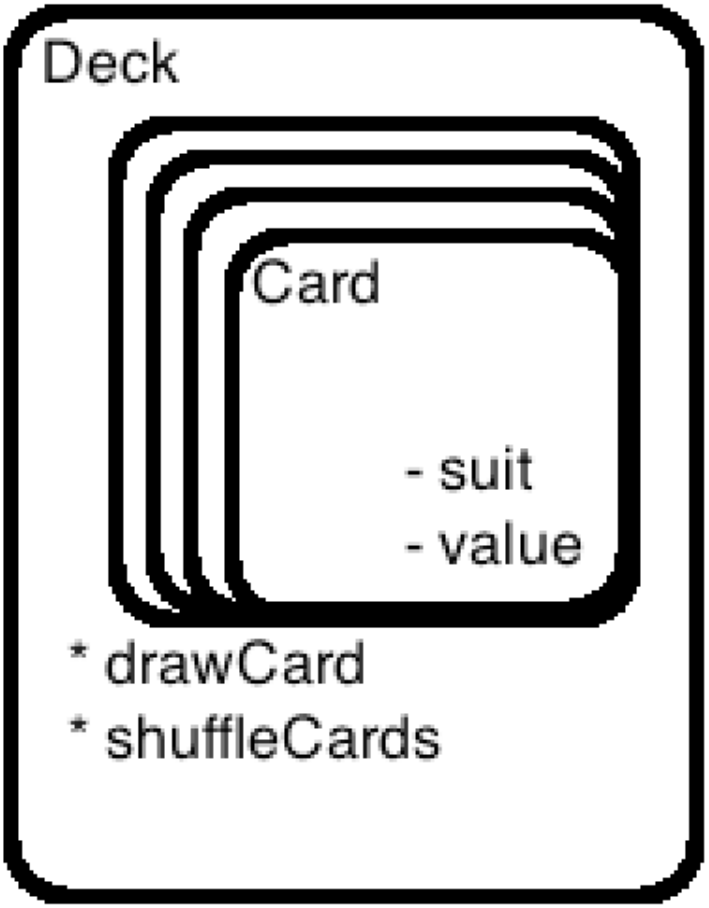
That in mind, individual `Card`s don’t seem like the right place to work with the score. Every other action we identified uses `Card`s, rather than is something a `Card` does, so we are finished here!



Deck

The next level we can look at is a `Deck` of `Card`s. Like the name suggests, this object represents a collection of individual `Card`s. The `Deck` itself doesn't have any properties, but what it does have is a set of behaviors. We identified 3 actions that involve a `Deck`, `Dealing Cards`, `Drawing Cards`, and `Shuffling Cards`. An interesting note to make here is that the distinction between `Dealing Cards` and `Drawing Cards` mostly comes down to *who* is performing the action, not the action itself.

We can handle both of those with a single `getCard()` behavior. Combining that with a `shuffleCards()` behavior gives us a pretty good looking `Deck`.



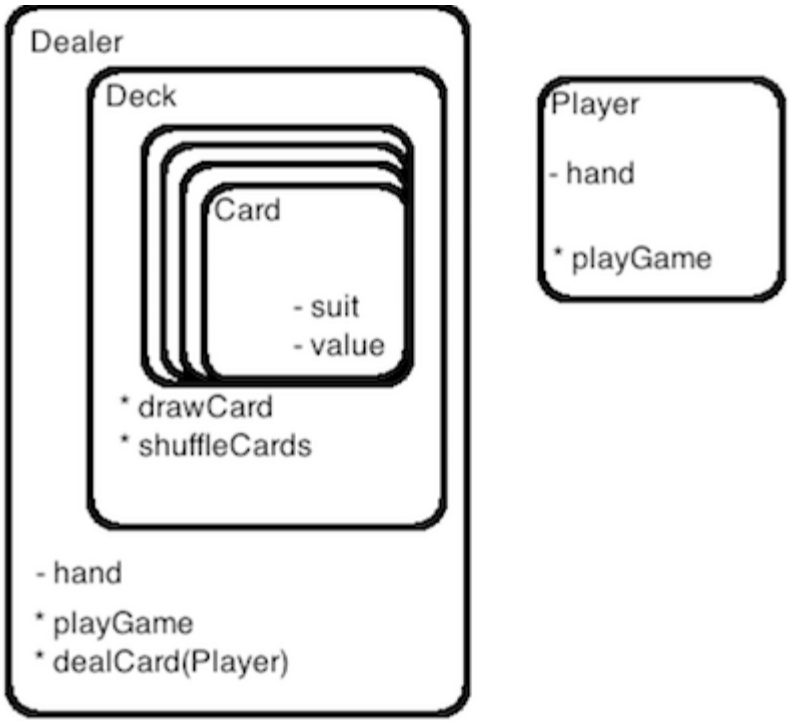
Player

The `Player` has the unique property of a `hand`, which holds the `Card`s that `Player` currently has. We also need a method to hold whatever logic determines whether a `Player` wants to draw a `Card`, or stay with what they have. What this method actually does sounds too low level for us to worry about right now, so lets just give it a name like `playGame` and forget about it. At the end we can come back and flesh it out if needed.

Dealer

The `Dealer` is pretty similar to the `Player` in the properties it needs. Again we need a `hand` to hold the cards the `Dealer` currently has. When its the `Dealers` turn it also needs some logic which determines whether to draw another `Card` or not.

Looking back through our use cases, one other behavior the `Dealer` needs to have is the ability to give `Cards` to other `Players`. There are two objects the `Dealer` needs access to in order to do this: the `Deck` to get a `Card` from, and the `Player` to give it to. Logically it doesn't make sense for the `Dealer` to contain any of the other `Players`, so we can take the `Player` as an argument. On the other hand, it does make sense at this point for the `Dealer` to own the `deck`. The `Dealer` is the only one so far who needs access to it, and, just thinking about it in terms of real life, the `Dealer` in a card game will often physically hold the `Deck` of `Cards`. If we need to change that later we can.



There is something interesting to note at this point. If you look at the list of behaviors between `Dealers` and `Players`, there is a lot of overlap. In fact, they really perform the same basic roles; they each have a hand of `Cards` and also some logic to play the game. Of course, in addition the `Dealer` has some extra functionality like the `Deck` and dealing cards. In real code I would expect the `Dealer` to inherit from `Player` to signify their relationship. To keep things simple I'm not going to annotate the diagram with this information, but in your interviews keep an eye out for situations like this.

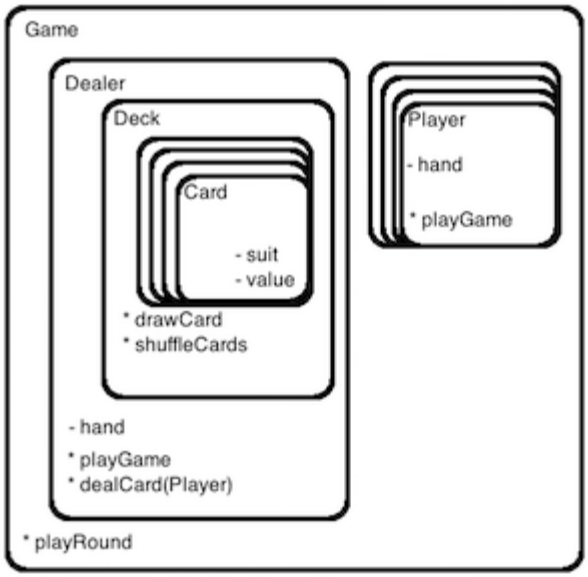
Game

So far we've talked about the different objects in the game, but not really the meta part of the design, which is how the `Game` itself works. Lets walk through the flow for one round of our game given the design we have setup above.

- Start Game
- Tell dealer to deal cards
- Ask Player 1 for what they want to do
- If they draw a card, ask the dealer to give it to them

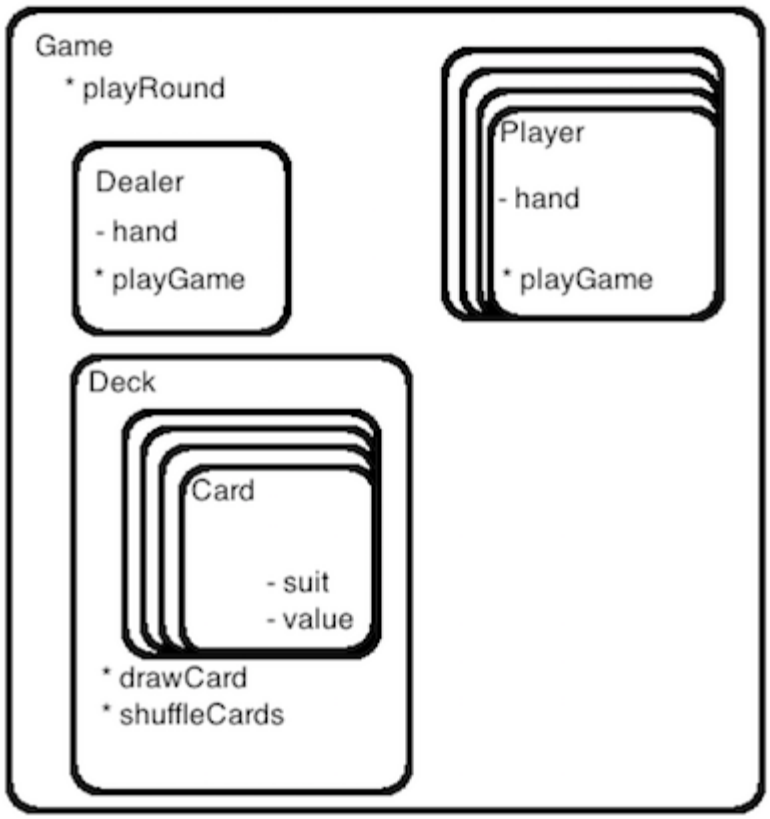
- … (repeat until all players have gone)
- Ask Dealer for what they want to do
- If they draw a card, ask the dealer to give it to themselves
- … (repeat until dealer is out of moves)
- Calculate score
- Shuffle Deck
- Start next round, or quit.

A couple things should probably jump out at you. First, we can add the `Dealer` and all of the `Players` as properties to the Game. This makes sense, even in real life, because those concepts exist only in terms of the `Game`. Being a `Dealer` with out a `Game` means, well, jail time I would assume. At this point our design looks like this.



Second, the `Game` feels like it should be responsible for how everything fits together, but its delegating a lot of the `Card` manipulation to the `Dealer`. This feels weird, especially when it comes to giving `Cards` to the `Dealer` itself. The easiest way to see this is to write out what that method call would look like: `dealer.dealCard(dealer)`.

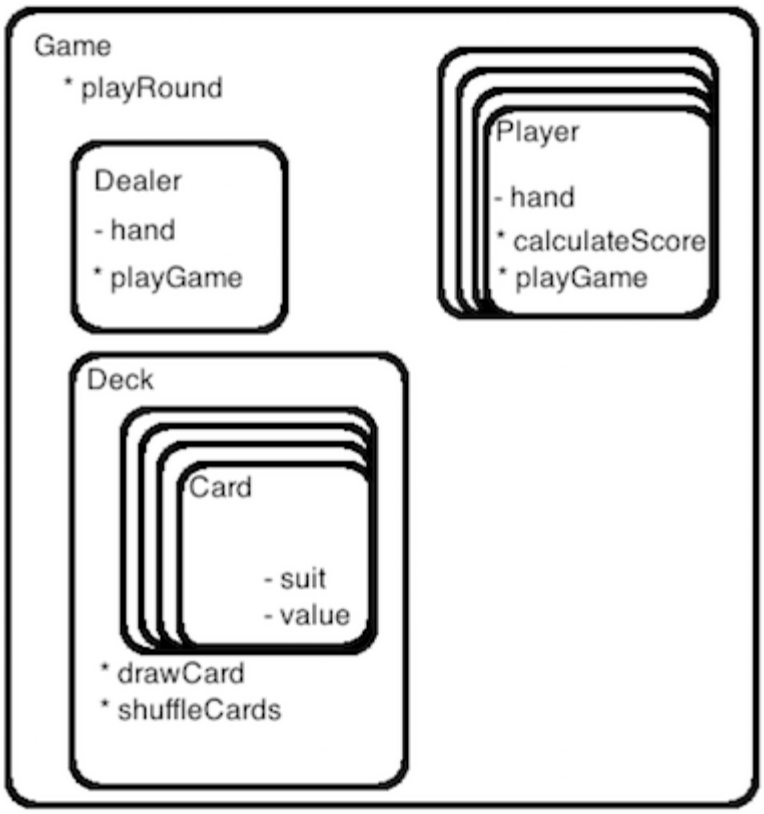
Fundamentally the problem is that the two references we need to make this work (the `Deck` and the `Player`) are at different levels. We can fix this by shoving the `Players` down into the `Dealer` object, or pulling the `Deck` up out of the `Dealer`. The first doesn't make any sense at all and would make our design even more convoluted than it is. The second option, though, is pretty easy to do and leaves us with this.



A cool outcome of this is that our `Dealer` object now doesn't have any unique behaviors or properties. Its special in the fact that we compare other `Players` scores to it when figuring out who won, but really its just another `Player`.

Scoring

The last thing to talk about is everyone's favorite: scoring! There are two ways we can handle this. First, we could add a behavior to each `Player` object which returns that individual `Players` score. To figure out who won or lost, the `Game` can compare what each `Player` says is their score against what the `Dealer` says is their score.

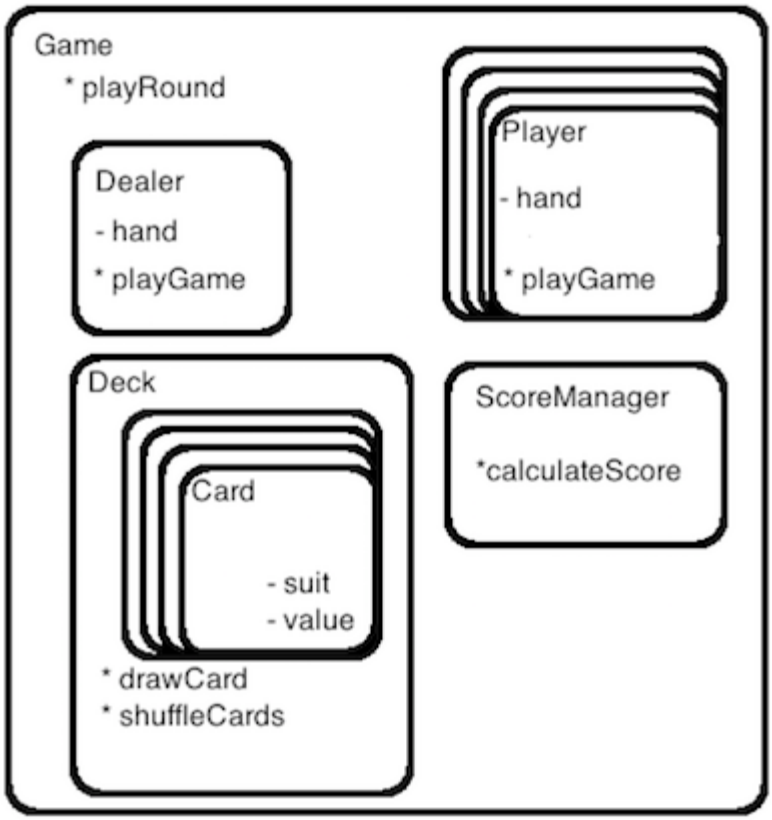


Players are now in control of two things: how a `Player` behaves and how the `Game` is scored. This feels weird. To see why lets look at one particular scoring rule. In BlackJack, any time the Dealer ties in score with a Player its called a “push”, and no money changes hand. When I play BlackJack, though, I always play with the house rule that ties are counted as a win for the `Player` . To make our design handle both of these scoring types we would need two different types of `Players` , one called `RealBlackJackPlayer` and one called `DansHouseRulesBlackJackPlayer` . All of the behavior would be the same between the two, with the exception of the scoring algorithm.

To be technical, our Player object now violates the Single Responsibility Principle, but without even knowing what that is, it just sounds too complicated. We can do better.

We could move the scoring up to the level of the `Game` , but that ends up with the same basic problem. We just want to change the scoring rules, not need a new `Game` for each house rule we add! The best solution here is to create a new type of object called `ScoreManager` , whose sole job is to understand these scoring rules. To get a `Players` score, the `Game` gives the `Players` hand to this object and gets back a

score. Now if we want to change the rules on how specific `Cards` are scored, or enforce some crazy rule that only a single `Player` with the highest score can win any given round, the only logic that needs to change is in this single object. All together that looks like this.



Looking forward

We started out by designing a program to play the game of Blackjack. Lets get nuts and see what it would take to support an entirely different type of card game like Poker.

Not much, as it turns out. Poker is just another card game and, since our design is so flexible, we can wire in all the changes with no problem. If we were to talk about this in terms of OOP concepts, `Player`, `Dealer`, and `Game` all can be thought of as interfaces with different implementations on a per game basis. We can imagine a `BlackJackGame` class which depends on a `BlackJackPlayer` class to work, but we could also use a `PokerGame` class which depends on a `PokerPlayer` class to function. Don't forget we need to update that `ScoreManager` class we just created as well! The key of a good design is

that the design itself doesn't need major changes to support new use cases, its just the *implementation* details.

What if we wanted to extend this to something different from a classic card game, maybe to something like Magic? We could do that too! `Deck` can be another interface, with specific implementations depending on what `Game` you are playing. The `MagicGame` class would need to change drastically, including adding a second `Deck` and some concept of a `Board`, but again, these are implementation details. The core design stays the same.

Conclusion

Whew. That was a long post, but lets recap. We started with a super vague question, massaged it into some use cases, and came up with a design that would make those possible. We started from the simplest object, a `Card`, and moved up to the most complex object, the `Game` itself. Along the way we talked about keeping our design flexible and always looking for inspiration in real life.

Have a pattern, or design you think is more flexible? Let me know in the comments below!

Update

Catch up on the introduction to the series, learn how to solve algorithmic questions, coding problems, Object Oriented Designs, or tell engaging stories for the behavioral interviewing questions!