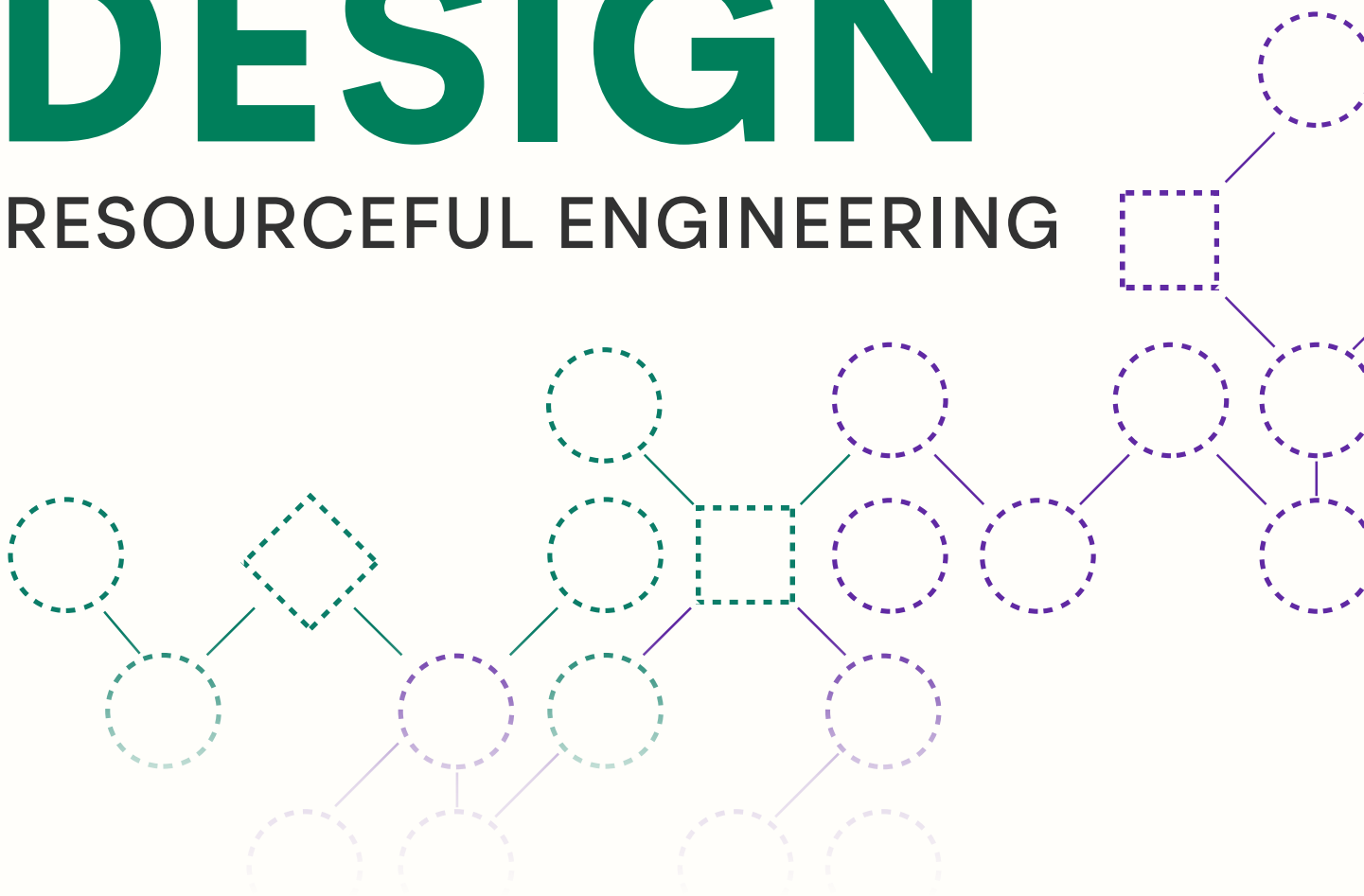


SAMPLE

MOBILE SYSTEM DESIGN

RESOURCEFUL ENGINEERING



Become a **better app developer** by using
mental models that **apply to the real world**

TJEERD IN 'T VEEN

Contents

Preface	1
0.1 Acknowledgements	3
0.2 This book is an early release edition; What to expect	4
1 About This Book	5
1.1 System Design versus Software Architecture	6
1.2 Why is System Design important?	6
1.3 Common challenges for mobile engineers	7
1.3.1 Rapidly changing environment	8
1.4 Why does this book exist?	8
1.5 Why this book's subtitle is called 'Resourceful Engineering'	9
1.6 What to expect during Mobile System Design interviews	10
1.7 What this book is not	11
1.7.1 This book is not a traditional programming book	12
1.8 This book is about timeless principles, not trends	13
1.9 This book is for iOS, Android, and multi-platform developers alike	13
1.10 Is this book for you?	14
1.11 How this book works	14
1.11.1 A strong focus on building the right things	14
1.12 The chapters	15
1.12.1 Chapter 1: About This Book	15
1.12.2 Chapter 2: Turning a Briefing Into a Strong Plan	15
1.12.3 Chapter 3: Holistic-Driven Development; Turning a Plan Into Code	15
1.12.4 Chapter 4: System-Wide Testing; Delivering Higher Quality Apps	16
1.12.5 Chapter 5: Cross-Domain Testing; Testing More With Less Effort	16
1.12.6 Chapter 6: Dependency Injection Foundations	16
1.12.7 Chapter 7: Sane Dependency Injection Without Fancy Frameworks	17
1.12.8 Chapter 8: Dependency Injection on a Larger Scale	17
1.12.9 Chapter 9: UI Frameworks, Architectures, and Supporting Multiple Products	17
1.12.10 Chapter 10: Delivering Reusable Views; The Art of Decomposing a Design	18
1.12.11 Chapter 11: Reasoning About Views, Components, Screens, and Bindings	18

1.12.12	Chapter 12: Pragmatically Implementing UI	18
1.12.13	Chapter 13: Delivering Self-Sufficient Features, Part I: The Art of Staying Nimble	19
1.12.14	Chapter 14: Delivering Self-Sufficient Features, Part II; Self-Loading Features . .	19
1.12.15	Chapter 15: Delivering Self-Sufficient Features, Part III; Making Features Portable	20
1.12.16	Chapter 16: Reusing Views Across Flows	20
1.12.17	Chapter 17: Taming Complex UI	21
1.12.18	Chapter 18: Crafting Robust and Reusable Navigation Flows	21
1.12.19	Chapter 19: Design System or Not; How a UI Library Lays the Groundwork . . .	21
1.12.20	Chapter 20: 20. UI Library Fundamentals, Part I: Typography and Colors	22
1.12.21	Upcoming chapters	22
1.13	About the author	23
2	Turning a Briefing Into a Strong Plan	25
2.1	The briefing	26
2.1.1	An initial impression	28
2.2	Evaluating common approaches	28
2.2.1	Start with UI?	29
2.2.2	A data-focused approach?	29
2.2.3	Creating an app-skeleton or flow-skeleton?	30
2.2.4	Starting by making components or features?	30
2.2.5	Drawing a diagram?	30
2.2.6	Decide on an architecture?	31
2.2.7	A recommended approach	32
2.3	Sketching out a landscape	32
2.3.1	Everything is connected to a course	34
2.3.2	How far do we decompose?	35
2.4	Uncovering secondary requirements	36
2.5	Working with Designers; Getting secondary features	37
2.5.1	Whether or not a design is the "law"	38
2.5.2	What is "pixel perfect", really?	38
2.5.3	Designs often encompass best-case scenarios	39
2.5.4	Not everything has equal priority	39
2.5.5	Verify the existence of pre-existing components	40
2.5.6	Ask general UI questions	41
2.5.7	Ask functionality-related questions	42
2.5.8	Talk about error handling	43
2.5.9	Talk about time-investments and start thinking in a less binary fashion	43
2.5.10	Giving feedback to the designer	44

- 2.5.11 Updating the landscape 44
- 2.5.12 A fast app is key 45
- 2.5.13 Scheduler 45
- 2.5.14 Deep Linking 46
- 2.6 Aligning with backend engineers 46
 - 2.6.1 Align on about User sessions, environments, tokens, and timeouts 47
 - 2.6.2 Align on consolidating network calls 47
 - 2.6.3 Be on the same page with errors 48
 - 2.6.4 It’s okay to deviate from backend custom error codes 48
 - 2.6.5 You might be the backend guinea-pig 49
 - 2.6.6 Read code from other client implementations 49
 - 2.6.7 Consider push notifications 50
 - 2.6.8 Feature-specific questions 50
 - 2.6.9 Updating the landscape with backend requirements 51
- 2.7 You are the link between backend and design 51
- 2.8 Closing thoughts 52
- 2.9 The takeaways 52

- 3 Want to read more? 55**

Preface

There's no doubt that the term "System Design" has been popping up more and more in the mobile engineering industry.

When mobile engineers apply for jobs, System Design is becoming more of a typical and standard part of the interview process. Many engineers may be familiar with the process of stumbling upon System Design books, only to discover that they cater to the needs of backend developers.

Unfortunately, the resources available specific to supporting mobile developers still remain limited when it comes to Mobile System Design.

In addition, although we may have a general working definition of system design for backend development, there lacks a universally conferred understanding of what we actually mean when we say, "Mobile System Design."

When we lack aggregate knowledge on the subject matter and pair this with fuzzy definitions, we decrease our likelihood of success as mobile developers.

Furthermore, in the mobile industry, there is a strong focus on the latest trends, delivering the shiniest UI, and engaging in the favorite pastime of squabbling amongst each other about which UI architecture is the "best."

But, in taking a step back from this all, our emphasis and attention could be more effectively directed towards improving our fundamental skills. Skills that are more timeless. Skills that go deeper, beyond beginner tutorials or fleeting trends.

This is what I hope to change with this book.

When we produce small apps as indie developers, we don't always require a technical design upfront. But, for regular jobs where you work in teams, mobile apps are typically larger and absolutely more complicated.

As soon as apps grow more features and become more complex, strong fundamental skills become ever more crucial. This is where knowledge about Mobile System Design is imperative and effective especially given the ever-changing landscape and fast-paced environment of mobile engineering.

In my experience as an iOS Tech Lead at ING international Bank and staff engineer at Twitter (before it became X), I noticed certain patterns where developers – myself included – find it increasingly more

difficult to deliver features as soon as apps grow in size and complexity.

To handle complexity, developers look for crutches, common patterns, or architectures to wriggle themselves out of problematic situations.

In reaching for heuristics, or rules of thumb, many developers often rely on SOLID principles. However, in going against the grain, I find it too outdated to keep up with modern demands. Unless you're really into subclassing and using interfaces to over-abstract your code, SOLID principles should be set aside.

Another related issue with crutches is the fact that some may call their own code "clean code", without actually checking if their coworkers find it readable or easy to comprehend.

The programming world is rapidly evolving. Today, software design favors composition over subclassing to handle complexity and we are entering the realm of using declarative frameworks to create UI, with mostly positive results. This affects how we design our apps.

One main reason this book exists is to assist with job interview preparedness. At the time of writing, the market took a big downturn and many people, unfortunately and unexpectedly, lost their jobs. When later churning through repeated rounds of interviews, many have realized that System Design has become an integral step in the interview process.

The aim of this book is to provide people with the necessary skills and knowledge to help increase their chances of getting the job that they really want.

However, this book isn't aimed solely towards mobile engineers taking part in future interviews. This book is also geared towards supporting and improving day-to-day work activities, given that strong technical design skills have the capacity to make positive, longstanding impact on one's work experiences.

Luckily, I was in a unique position to make this book a reality based on my past professional endeavours. I wrote a technical book before, *Swift in Depth*, published by Manning and have worked on multiple large-scale mobile applications for large-scale international businesses situated in Europe and North America. I have been a mobile developer for nearly 14 years, built features for over 24 years on different platforms, and I have been on both ends of the job interview process for mobile developers.

I wanted to take my years of experience and consolidate this plethora of knowledge into insights on mobile app development that can be accessibly shared with others. In light of these efforts, I'm now happy to say that there finally exists a Mobile System Design book available for mobile engineers.

That means that this book isn't a quick bite-sized snack. It does not contain quick-and-dirty tips that you may often find in blog posts or the many beginner-focused tutorials. There is a time and place for that, but that's not this book.

In this book, you will do what you probably are already doing: Making features and apps. But, we are going to go deeper at every step to refine your skills.

The core message of this book is that you become a better developer by working on your *fundamental* skills. Forget the fancy tools and frameworks for a moment, and let's get back to the basics. The takeaways from this book will last your entire career and make you a better developer in many critical aspects.

One way that this book stands out is that it keeps team dynamics in mind. It does not solely focus on the technical challenges. For instance, deciding what to prioritize often isn't a coding question. It's about how to make a *team* move faster, not just yourself. As you'll discover, sometimes focusing on the boring parts can make your team deliver faster as a whole.

This book is opinionated with a strong focus on keeping it simple. The book will often use one approach instead of showing the pros and cons for every approach. The benefit of doing so is that it allows us to go in-depth about certain topics. Expect to disagree sometimes. This is normal and I encourage you to be critical. But please, try to be open and see if you can learn something in every step of the process.

A goal of mine was to avoid making this book too high-level, focusing only on diagrams. It's a good idea to design your software with diagrams, but it's not necessarily enough.

When you write the actual code, you think of problems in more detail. That's when we realize that high-level diagrams don't always reflect reality. Designing your app is important, but being able to go in-depth and identify issues in your design is part of that. Hence why this book will often go from high-level design, to the nuts and bolts on a code-level, and back again.

I hope this book can help you improve your career prospects and your confidence as a mobile developer and I hope you have a lot of fun and insights reading it.

0.1 Acknowledgements

I want to give special thanks to Dimitar Gyurov, Marie Denis, Elvirion Antersijn and Nicole Yarroch for reviewing the chapters. Thanks to your help, I made chapters more accessible and clear.

I want to give special thanks to Donny Wals for being my sounding board. You gave me more perspective on the same problems that we run into during day-to-day mobile work.

Thank you, Leo G Dion for making people aware of my book and being so supportive of it.

Thank you, Hugo Visser for giving me deeper Android insights.

I want to give thanks to Martin Lechner and Cristian Caroli for their support while writing this book.

And special thanks to Jenika, my wife, for being so supportive and patient while this book kept me mentally occupied.

0.2 This book is an early release edition; What to expect

Thank you so much for your early interest in this book. You'll be one of the first people to read and absorb the concepts shared in here.

Before you continue, I'd like to share a few things to keep in mind since this book is in beta.

First, I will add more chapters as soon as they come out. Expect book updates with new sections, better phrasing, updated code listings, updated diagrams, and so on.

Expect typos, odd phrasing, and code-listings to be cut off awkwardly. Because the copy-editing happens after all writing is finished.

The chapters aren't set in stone, either. Sections, paragraph, and sometimes even entire chapters might get updates.

For example, maybe a section will be rewritten because it's not clear enough. Or maybe an unwritten chapter is replaced by a different one.

The chapters that are already written won't just disappear. But the unwritten chapters are more susceptible to change to help the flow of the book. This is a normal process of writing a book.

And last, the illustrations are not all final.

If you find any other issues, errors, or any wrong statements, then feel free to reach out to tjeerd@swiftindepth.com, or DM me on Twitter/X/Mastodon @tjeerdintveen.

When chapters or sections aren't clear enough, please let me know! I will gladly use your feedback to improve this book.

Enjoy!

– Tjeerd

1 About This Book

If you ask mobile engineers what Mobile System Design *is*, you'll get a wild variety of different answers.

Some might say it's about UI Flows, others about job interviews, or they will mention it's about UI-architectures. Yet, others may tell you it's about domain modeling or dependencies, or they'll say it's about a modular codebase.

The term "System Design" is usually reserved for backend engineers and often associated with job interviews. During these interviews, a backend engineer would design a technical solution to solve a problem that the interviewer gives.

For example, an interviewer might ask a backend-engineer: "*How would you implement a livestream chat-service?*". The candidate draws a graph of the services needed and explains how data flows between them. They answer questions about scaling up, storage, redundancy, and more..

Sometimes, the candidate would write (pseudo)code to explain the required types to make it all work together.

In the mobile world, we see an increasing amount of System Design interviews pop up, too. Yet the problems differ from the more traditional System Design challenges given to backend engineers.

During a System Design interview for mobile engineers, the question might be similar: "*How would you build a feature that does X*".

But you'd be focusing on designing the architecture for a client app with mobile-specific constraints, such as, but not limited to:

- Designing complex flows while considering limited screen space.
- Making a feature work offline, yet keep data in sync with the back-end.
- How to avoid hammering the servers while downloading large amounts of data.
- Handling networking and local cache invalidation.
- Making a feature reusable across multiple devices and platforms.

Despite the association, system design is *more* than just a step during the job interview process. It's about the ability to create a solution to cater to the requirements of a business. Both at work and in job interviews.

A system can be small. Perhaps your work today is making a tiny app with three screens. But it can also be gigantic; Some apps are a multi-modular codebase where hundreds of developers are working on the *same* codebase and have to share a lot of libraries together.

1.1 System Design versus Software Architecture

The term "*System Design*" is nebulous, because it touches upon many overlapping software design practices that fall under the umbrella of *Software Architecture*, such as: domain modeling, architectural patterns, API design, or component design.

With this book, we try to define System Design as *designing a technical solution to satisfy business requirements*.

To put it simply: You receive requirements, and you have to figure out what components to make and how they work together to solve the business' needs.

Specifically in this book, we narrow it down to *business* requirements, since it mostly reflects requirements found at work. As opposed to *any* software requirements, such as hobby projects and toy apps where a lot more rules can be broken.

Another way to think about System Design is *coming up with the components and their APIs to solve a problem*.

1.2 Why is System Design important?

Software is expensive.

It's important to reduce the time or sprints of a project, and having the ability to adapt swiftly to new demands, features, and requirements. Even being able to even delete entire chunks of your application is crucial.

Modern software moves fast, and mobile apps especially so. On top of a rapidly-changing platform, we have to be able to adjust and deliver a product to help solve our customer's needs.

We have to be ready for any changes and we should aim to avoid wasting time building features nobody uses.

But being nimble is not always possible when we're working in a rigid system, or a codebase that easily breaks.

Mobile System Design helps ensure our apps are of high quality, developed quickly, and can be adjusted where necessary while managing a growing codebase.

1.3 Common challenges for mobile engineers

Some might quip that mobile apps are JSON viewers—of which I am also guilty when feeling snarky. Once you go beyond toy-apps however, system design and a strong architecture do become critical for a mobile app.

Mobile development poses interesting challenges. As a team, you'll be shipping one binary. Your focus is more on local architectures and delivering code that's shared by the entire team.

The larger the app and teams, the more teams have to think about—and struggle with—topics such as:

- Delivering shared components that need to meet company-wide requirements.
- Taming abstractions and keeping complexity low.
- Over-engineering.
- Too much, or too little, code duplication.
- The danger of a convoluted codebase.

These problems are *not* unique to mobile development. It is, however, a situation mobile engineers quickly run into because they are shipping one binary together with *all* features combined; As opposed to, say, a web-team that can ship features independently.

You might choose to decouple features into their own namespaces, packages, libraries, SDK's, frameworks, and modules; However, all these pieces will still have to get along, since you'll be glueing all these "independent" parts together as a whole in the shape of an app.

Often, these independent parts are relying on components shared by all other features, such as UI components or shared business logic; If you're not careful, changing one element in a shared component can affect *all* features in the entire app for all mobile teams.

Keep in mind that there is no shame in relying on shared code either. When implemented well, it can give you engineering superpowers; Such as implementing a shared module that contains a UI Design System. This system would contain all the components that power up the features we make, saving a team tremendous time.

To get more autonomy, developers might choose to split up features into modules. They might wonder how many modules an app needs and how much they should rely on interfaces or even interface-modules. Splitting up your codebase into modules increases the need for strong API design and thinking a bit differently about components. When poorly implemented, a modular design will even backfire.

Whichever way you go about it; As mobile engineers we have to find a balance between shared components, modules, abstractions, architectures, handling dependencies, and everything in-between. All the way from a monolithic app to a giant modular codebase. This is where this book can assist you.

1.3.1 Rapidly changing environment

Another challenge that mobile engineers face is that the mobile industry moves *fast*.

New devices get released every year, and major OS updates bring new UI and exciting functionalities. Sometimes, we even get to deliver products for entirely new platforms, such as watches, TV's, or VR glasses.

On top of dealing with foundational changes, businesses constantly require changes to serve customers.

As a result, a mobile codebase is ever-moving, and frequently shifting.

On average, the UI aspect of mobile apps changes more often than foundational code for app developers. Because mobile apps are customer-facing and we have to keep up with the latest advances that come with OS updates.

Conversely, code that's more foundational, such as making a network calls or storing locally protected data, can often keep working regardless of the UI solution.

Since the UI layer tends to be more volatile for mobile developers, the book focuses on building a firm foundation. This way, you're ready to handle many changes to UI. Such as new OS updates, changing UI architectures, adapting to UI frameworks, and you'll even be ready for entirely different products; Such as tablets, watches, TV platforms, or VR.

1.4 Why does this book exist?

This book is an answer to mobile engineers who struggle with repeated and common issues during the mobile development process.

Over time, being asked to implement a feature transforms from “Sure! I’ll have it done by this afternoon!” to “I’d first have to check with Steve but he’s on vacation and we first need to refactor this other piece of code, but that code is in the middle of a migration so I can only make a temporary fix.”

Or: “I can’t move forward because we rely on version 2 of this third-party SDK, and we need version 3. But, version 3 has a singleton that makes it almost impossible to test this flow, so we need to wait for a change”.

These issues are never a specific problem with a clear silver bullet answer. “Oh, if only we had used reactive programming, our app would be so much easier to work with!”.

Unfortunately, challenges with mobile development are more nuanced.

Many engineers’ struggles are a buildup of smaller issues that are introduced during the development process.

Individually, these issues aren’t a big deal and easily slip under the radar during code reviews. A little abstraction here, a small “temporary” fix there, some duplicate code here. But now, these tiny problems turn into a tedious codebase that’s as much fun to work in as trimming a cat’s toenails.

It’s death by a thousand cuts.

Development-speed slows down, tech-debt increases and people might start saying the R-word (refactoring), worrying managers who’d rather have engineers focus their time on shipping features.

In this book, we’ll make smarter, more robust decisions that will lower the chances of our codebase turning into a “big ball of mud”.

NOTE: “Big ball of mud” is a term for complex technology that lacks structure and design. The term was popularized in Brian Foote and Joseph Yoder’s 1997 paper of the same name.

1.5 Why this book’s subtitle is called ‘Resourceful Engineering’

Looking up the definition of *resourceful*, we get:

as in skilled. “able to deal well with new or difficult situations and to find solutions to problems”

- merriam-webster.com

“Someone who is resourceful is good at finding ways of dealing with problems.”

- collinsdictionary.com

“able to deal skillfully and promptly with new situations, difficulties, etc.”

- dictionary.com

As a developer at a company, you receive requirements; you don't have the full information, and yet you must deliver. Even when you get an extensive list of requirements and many designs, there always are unknowns. You'll have to be ready for that.

Resourcefulness is something you are *yourself*, but it affects yourself and your team. You must be resourceful in developing software and features.

Once you're ready to develop, you need to be adaptable.

Customers, managers, and businesses can change their minds and our solutions may not work anymore. You must remain flexible and keep momentum.

On a technical level, you need to make sure a codebase is flexible enough. But, you also don't want to make a codebase *too* flexible. Once you develop solutions that "might be useful later", you risk over-engineering. It's a harmful practice that can make a codebase overly complicated, but it's very nuanced and hard to balance.

The book will break down "resourcefulness" into tools and strategies. It will help you become a resourceful mobile engineer.

This book puts a strong emphasis on being able to move forward without knowing all the information.

A guiding topic is to keep your code flexible and simple *without* over-engineering. This way, you can confidently deal with unforeseen changes and keep a simple, yet adaptable, codebase.

1.6 What to expect during Mobile System Design interviews

The book's focus is not limited to passing interviews. But all the techniques inside will be extremely helpful for the interviewing process.

During Mobile System Design interviews, you're tasked to design a feature or functionality. You do this by drawing diagrams or writing out (pseudo) code.

During this time, interviewers are looking for so-called "signals". These signals are indicators that will tell them you are at the level that you're applying for.

If you're applying for a junior, SW I or SW II level, job, you can expect to receive UI and feature specifications to come up with a technical design. For example, the interviewer might ask: "How would you build a screen that downloads and displays a list of workouts in a gym app?"

While modeling this feature, you'll cover topics such as networking, testing, and how to decouple UI from business logic.

When interviewing for a senior level, expect the questions to become more broad or “larger scale”. The interviewers might look for signals that you deliver a feature that serves larger flows and multiple use-cases.

Alternatively, they might ask how you would make a feature reusable across the *entire* application or multiple targets. This will touch on topics such as architectures, interfaces, generic code, domain modeling, and dependency injection. You can also expect to be asked about more complicated topics, such as downloading large amounts of local data, caching, or security.

If you’re applying for an even higher position, such as a staff engineer, expect to go more grand-scale. At this level, expect having to come up with solutions ranging from building features all the way to release management.

Interviewers may ask what is required to scale up the feature so that it can serve all teams, multiple targets, platforms, and domains.

The questions at this stage can go deep, such as writing generic, reusable components.

Alternatively, questions can go wide. Such as explaining how to introduce a large feature that requires a tremendous effort from many teams. They may ask you what a highly modular architecture would look like for a decoupled codebase suited for an entire organization.

If you’re preparing for an interview, this book will help you with a deeper knowledge of common interview questions. It will help you ask better questions while being briefed — signaling to the interviewer you are capable of thinking about a problem on a deeper level.

Once an interviewer asks you to create a program from scratch, then this book will have your back. In the Holistic-Driven Development chapter, you will see how to get something up and running *quickly*.

A common theme for Mobile System Design interviews is architectures, interfaces, domain modeling, testing, and dependency injection. The book covers all these topics in depth.

A big bonus that this book offers is that you don’t need to rely on third-party frameworks as crutches to achieve excellent results. You’ll be prepared in an interview setting where code will be vanilla and first-party, such as coding in a web browser.

1.7 What this book is not

This book focuses on feature-development, mimicking daily work for the mobile engineers. The book starts by building a local feature until you reach a larger scale where the book covers topics such as designing domains suited for an entire app and delivering a modular app.

However, in this book you will not make an app that’s feature-complete and ready to ship. This book is about processes, ideas, concepts, mental models, and avoiding pitfalls.

One aim of this book is to help you pass those System Design interviews. However, this book is *not* a template for System Design interviews, where one might expect prepared answers to common questions. The book isn't here to provide you with a generic script to regurgitate back to interviewers.

For example, the interviewer might ask, “How would you store remote data locally?” and you might share trade-offs between a local database versus a key-value store versus using a secure storage. That is definitely a useful topic, but it can only cover rehearsed problems.

This book is about learning to reason so you can design *any* feature that's thrown at you at work or during interviews.

For example, in the briefing chapter you are given a feature, but instead of giving you common answers – e.g. “Use a database for large data but a key-value store for a simple dictionary” – the chapter is about *asking* the right questions back to the person briefing you.

By asking questions, it will signal the interviewer that you're getting to understand a problem on a deeper level. During daily work, it will help you find a better solution to your problem.

Sure, maybe *today* the problem is offline storage and you can give some rehearsed answers to that, but *tomorrow* the problem could be something completely different, maybe you're tasked to make an AR video solution where you may not even know whether there's an SDK for that, yet. By the end of this book you'll have more confidence to tackle unknown problems.

Since the book can't cover all problems that ever existed or will exist, the book focuses on giving you tools to handle most problems that are given to you.

1.7.1 This book is not a traditional programming book

This book is also not a “code book”. It contains code, and in some chapters more than others, but only to some extent. A lot of pages are more filled with reasoning about what we're making, and less so about code tips-and-tricks to build things.

Luckily, the book won't keep its advice generic and shallow. It demonstrates *how* to do everything in code where needed, just not 100% of the time.

This book will not cover UI styling and animations. Despite that, it has a strong focus on UI from a system perspective; You will learn how to best decompose UI into components. You'll even go as far as designing a UI library powering a *Design System*. We will cover architectures and their roles, and you will implement a feature in an architectural sense combined with domain modeling.

Last but not least; This book focuses on *native development*. Releasing a hybrid app that's shipping with web-technologies is *outside* this book's scope. Keeping the project native is, to be honest, something most mobile engineers can appreciate.

1.8 This book is about timeless principles, not trends

One of the key messages in this book is that you can get a lot done with plain, vanilla, first-party code.

Forget trendy third-party frameworks and architectures for a moment. With some sane programming and safeguarding the code quality, you can deliver a giant mobile application.

Not too long ago in our industry, Photoshop and Illustrator got replaced by Figma and Sketch, and so will the next best thing replace them. Objective-C and Java are still in use, but they have to make way for Swift and Kotlin.

Nowadays, some might say UIKit and XML layouts are outdated, and may consider SwiftUI, Jetpack Compose, and Flutter the future. Who knows, maybe one day instead of writing native apps for iOS and Android we'll write a new thing that runs on anywhere, let's call it mobile-assembly.

Change is inevitable. But, despite tools and trends changing around us, the process will be similar.

You'll deal with companies or people having an idea to produce something. You're alone or in a team, you receive designs and specs, and you need to deliver and focus on the right things.

After releasing an app or update, you will update and maintain that code. It doesn't matter whether you use Swift or Kotlin or Kabomaflug (a new language I just made up), the concepts will be similar.

Some engineers may follow trends but struggle with foundational problems; They may use the latest and greatest frameworks, but their app might be a spaghetti code-mess underwater. Or they'd religiously program in a reactive way, but can't help but over-engineer their "pure functional programming" codebase. This book will help you avoid these pitfalls.

1.9 This book is for iOS, Android, and multi-platform developers alike

This book uses Swift as a vehicle to explain concepts and best practices, and it's not relying a lot on iOS specifically. This book focuses mostly on concepts, mental tools, reasoning, and approaches.

With some basic programming knowledge you'll be able to understand the Swift code examples with ease.

We won't go too deep about platform or language-specific requirements. Where needed, the book explains specific Swift keywords.

So whether you're an iOS engineer, Android engineer or use Flutter, React Native or other mobile platforms, you can apply the knowledge from this book.

1.10 Is this book for you?

This book assumes you're serious about developing apps, whether that's by yourself or within a team. The book does assume you will work with others, but it doesn't matter whether you're in a tiny startup or in a giant mobile department in big tech.

It's written primarily with junior and senior developers in mind. I'd argue that staff-level engineers can get plenty of value out of this book as well, depending on the background.

This book does go over the process that you're already familiar with, but aims to give you new perspectives, tools, and considerations at each step, so that you come out a better developer at the end.

If you're never made an app before, you can use this book to get some ideas on what starting a project from scratch would look like and how you can work in a mobile team. But it will not show you how to set up a project.

1.11 How this book works

This book starts by being briefed and receiving designs, and we build from there. So there is a natural flow to build something from scratch, and during this book's progression, this feature (and app) will grow.

If you want to jump to a specific topic, such as testing, you can. Just note that the chapters build on a feature that grows throughout the chapters.

1.11.1 A strong focus on building the right things

Building the wrong things *fast* is worse than building the right things *slowly*. It's easy to get distracted by details and losing focus.

Delivering fast means spending your time and energy in the right places. Because of that, this book will put a strong emphasis on keeping momentum and making sure we keep our focus.

1.12 The chapters

1.12.1 Chapter 1: About This Book

This is the chapter you're reading now. Where we cover the challenges of mobile development and how System Design can help.

1.12.2 Chapter 2: Turning a Briefing Into a Strong Plan

In this chapter, you'll get briefed for a feature that will be the key feature used throughout this book.

We'll reason about finding requirements, both obvious and hidden, and come up with a plan to implement it.

You'll learn how to sketch out a technical design while missing half the information.

The chapter covers talking to designers, backend developers, and other ideas to get a complete picture of what you'll be making.

If you're preparing for job interviews, then this chapter will be useful to you.

1.12.3 Chapter 3: Holistic-Driven Development; Turning a Plan Into Code

In this chapter, you'll learn on how to take a design and use that to deliver a feature. You'll do so by defining the components and designing the interfaces, or APIs required.

We'll take a holistic approach. Meaning that you'll jump between the components that you create, to end up with a working prototype. This approach helps you to deliver quickly and ensures you will keep the highest priority in mind.

This chapter is more code-centric; We'll go into API design, data modeling, and domain modeling.

It has a strong emphasis on designing without interface types and explains why.

You'll learn how to implement a robust feature quickly, without getting distracted by the details.

At the end of the chapter, we'll go over trade-offs and downsides of this approach to get a deeper understanding of when to apply this technique.

This chapter is crucial if you're a feature engineer or applying for jobs.

1.12.4 Chapter 4: System-Wide Testing; Delivering Higher Quality Apps

This chapter challenges the status quo in testing.

A common approach to testing is to use many interfaces and test the smallest units.

But in this chapter, you'll see how to test using fewer interfaces, allowing you to test the system on a larger scope.

The reason we do this is to get *more* quality guarantees *early* on in the development process. This chapter explains why this is important for mobile development.

As a bonus, you will learn how to test *more* by writing *less* code.

Like all testing approaches, neither is this approach a silver-bullet solution. But, we'll cover the trade-offs and you'll discover some techniques to handle its downsides.

1.12.5 Chapter 5: Cross-Domain Testing; Testing More With Less Effort

A major benefit of system-wide testing is that we can test across multiple domains more easily. We'll exploit that benefit to spend even less time to test more.

In this chapter, we'll reason about which domains are more volatile and which are more stable. To help you figure out where to invest your time and energy for writing tests.

It covers where to fix issues if any arise in our entire program, and the responsibilities of a domain when testing across them.

1.12.6 Chapter 6: Dependency Injection Foundations

This book has *three* chapters dedicated to dependency injection. This is because it's a contentious topic that many developers struggle with.

We start with the basics of *why* we need dependency injection. It covers some common scenarios that can be harmful.

One big part of this chapter is dealing with singletons, since they are common in mobile development.

It might be common knowledge that they are harmful. But this chapter offers some fresh perspectives.

Many developers like to modularize their code. This chapter shows the negative effect singletons have when you wish to modularize your code.

Another issue is that developers sometimes think that making singletons thread-safe is enough. But this chapter will show easy it is to break them in a real-world scenario, even when they are perfectly thread-safe.

Lastly, we will cover scenarios when singletons *do* make sense and are a good idea to implement.

1.12.7 Chapter 7: Sane Dependency Injection Without Fancy Frameworks

In this chapter, we'll start making an implementation for the feature in this book.

The goal of this chapter is to show you how to pass dependencies around *without* any fancy techniques or frameworks. The goal of this chapter is to show that we can keep it relatively straight-forward and simple.

1.12.8 Chapter 8: Dependency Injection on a Larger Scale

In this chapter, we handle dependencies in larger code-hierarchies.

You see how to reduce a giant dependency tree to make the problem smaller and more simple. By doing so, you'll learn how to weave dependencies across many types.

Then we go one step further; Handling dependencies in a modular app.

Because when an app grows, people tend to modularize them. But, when we are passing dependencies across modules, we have unique problems to consider. This changes the way we reason about our dependency solutions.

1.12.9 Chapter 9: UI Frameworks, Architectures, and Supporting Multiple Products

In this chapter, we will enter the phase where we'll implement UI.

But, before we do so, this chapter will cover how to *reason* about a feature so that it supports any UI architecture or framework.

We'll cover why UI architectures aren't as important as you might think.

The chapters will go over the benefits when you delay an UI implementation. You will learn about the benefits you get when you treat your features like a self-sustained command line tool.

Lastly, it will cover how to reason about a codebase when you want to support a wild variety of UI architectures, frameworks, and multiple products.

1.12.10 Chapter 10: Delivering Reusable Views; The Art of Decomposing a Design

Turning a design into views is a common task for mobile engineers. But, this chapter will show you how to do it in such a way so that you not only deliver your feature, but you will develop a UI library *without extra work*.

A big topic of this chapter is naming, and you'll see why it's crucial for delivering components. It encourages us to think about important concepts like abstractions, reusable components, over-engineering, and long-term thinking.

It's an important chapter for any mobile developer, and should help you make *better* components for mobile *and* other platforms.

1.12.11 Chapter 11: Reasoning About Views, Components, Screens, and Bindings

After breaking down the design, we end up with various views where some of those live in a separate UI library.

In this chapter we'll determine the approach for connecting, or *binding*, these views to our business logic to create the course feature.

This chapter covers UI patterns, emphasizing viewmodels as a common example. It will assist you in determining whether a UI pattern is necessary at all. Then, we'll make different considerations for an imperative approach, too.

After that we look into more philosophical questions, contemplating the roles of views, where we should place business logic, and how views are related to "screens".

This will equip you for the development of more intricate screens, showing that UI patterns are merely one means of connecting views. It will aid you to create user interfaces that are both simple and complex.

Finally, the chapter discusses the components and their role. We will dissect the nuances to help you decide when it is appropriate, or not, to create a reusable view component.

1.12.12 Chapter 12: Pragmatically Implementing UI

This chapter takes a practical approach by guiding you through the step-by-step implementation of a screen based on the initial briefing in this book. Its primary objective is to provide a detailed examination of considerations at the code level.

Assuming you have experience with typical UI implementations, the chapter also shares insights on maintaining a productive and efficient development process.

Exploring both top-down and bottom-up approaches, it discusses their relationship with Holistic-Driven Development.

A key insight emphasized is that, despite significant differences between declarative and imperative programming, the approaches in these paradigms can be remarkably similar.

Demonstrating this point, the chapter showcases the implementation of the same component both imperatively and declaratively.

1.12.13 Chapter 13: Delivering Self-Sufficient Features, Part I: The Art of Staying Nimble

In this chapter, we'll look at why self-sufficient features are important.

Self-sufficient features are features that can load themselves, handle their errors gracefully, and don't interfere with their parent's navigation by mutating their stack.

After explaining the concept and benefits, the chapter will share some high-level designs we'll apply to the project in this book to make a feature more self-sufficient.

This chapter will be part I of a brief series. Meaning that we'll spend a few chapters to fully grasp the concept. For instance, in the subsequent chapter, we will take a code-level look at the decisions we make in this chapter.

1.12.14 Chapter 14: Delivering Self-Sufficient Features, Part II; Self-Loading Features

Continuing with the self-sufficient features series, we'll look at how to ensure a feature can be self-loading and the advantages it brings.

You'll learn how to make a feature that you can "cut and paste" across an app which makes it easier to move around, including across modules. As a bonus, this chapter will show you how much easier it is to support deep-linking to a screen.

To achieve this, we'll look at supporting ID lookup and how a view can load and mutate data. Then, we'll look at a different approach where we'll make a view more flexible and reason about its dependencies.

Last, we'll briefly cover how to apply this method to features that can partially load themselves, such as using a pull-to-refresh mechanism.

1.12.15 Chapter 15: Delivering Self-Sufficient Features, Part III; Making Features Portable

Continuing with the self-sufficient features series, we'll look at how to ensure a feature can be portable.

A portable feature is another way of saying that a feature works regardless of targets, platforms, architectures, and UI patterns.

Portable features bring us a lot of benefits; First, it allows us to deliver features that can survive UI migrations, such as moving from UI pattern to a new one, or moving from one paradigm to the next.

Besides that, portable features enable us to disconnect a feature from a screen, ensuring it can work in the background.

But perhaps most importantly, portable features allow us to unit-test a larger surface, so that we can rely less on UI Tests or manual verification for the same quality guarantees.

In this chapter, we'll look at how to make features more portable by pushing more logic out of the UI domain into the model domain. We do this by taking a detailed code-level look at how to achieve this.

1.12.16 Chapter 16: Reusing Views Across Flows

Delivering views are common, but making sure that we can reuse them in multiple flows comes with some caveats.

You might think that your views don't need to be reused in different flows. But, as this chapter will show, you can get the benefit of reusability with a similar time investment and complexity.

We'll start this chapter by exploring two common problems that prevent a view from being reused across flows.

Then, we'll look at three solutions.

The first approach involves closures, since it relates to declarative UI. After implementing this, we'll look at some trade-offs and downsides.

Moving on, we define a navigation interface, to be used by views. We learn that, even if we decouple a view from its navigation, the naming and expectations of an interface can still prohibit a view from being reused.

Finally, we reason about how we need to modify views when placing them in different flows. For fun, we design a declarative API, allowing us to build up the complexity of a view, while configuring it for various flows.

1.12.17 Chapter 17: Taming Complex UI

With Complex UI, we refer to views that take on too many responsibilities. They not only manage UI elements, but also various async operations, error handling, data, state, user interactivity, and so on.

Once UI becomes more complex, we risk at coming up with hard-to-maintain views.

In this chapter, we look at three common UI paradigms that are more complex than most views.

To come up with an excellent design, we take a high level look where we consider naïve approaches, and iterate on it to come up with better solutions.

This chapter is filled with plenty of diagrams and builds heavily on the previous chapters, namely chapter 11, and chapter 13 through 16.

Thanks to the knowledge of these chapters, we are well prepared to implement complex UI.

1.12.18 Chapter 18: Crafting Robust and Reusable Navigation Flows

Previously, in Chapter 16, we took a deep dive into making views reusable within flows.

In chapter 18, we'll build on that by focusing on how to design flows that make use of these reusable views.

There are many idioms and patterns for implementing navigation flows, which is why this chapter takes a more philosophical approach.

First, we'll implement a flow using a straightforward approach. Then, we'll examine the purpose of flows and redesign the same flow with a completely different approach.

Finally, we'll explore how this affects API design, ultimately creating a flow that's easier to implement and reuse.

1.12.19 Chapter 19: Design System or Not; How a UI Library Lays the Groundwork

In this chapter we cover what a design system is, why you might want one, and how to start with one.

It covers how a design system can be a giant project, and how we can start by focusing on expanding a UI library. This will then allow us to turn this UI library into a full-fledged design system.

This chapter is more theoretical, and helps you reason about approaches on how to introduce a design system into your company.

1.12.20 Chapter 20: 20. UI Library Fundamentals, Part I: Typography and Colors

In this chapter, we build the low-level elements of a healthy UI library.

We focus on typography and colors, breaking down the roles of semantic and primitive types.

For typography, we streamline the system to ensure developers use the right types, improving consistency and ease of use. We also explore key API design decisions.

For colors, we go deep—covering the layers of abstraction between color primitives, semantic colors, and various notations. This will help you think about colors in a more structured way. From there, we extend our system to support dynamic colors and theming.

To wrap up, we refine the color API, adding granular control that lets developers customize colors within clear constraints.

This is a hands-on, code-heavy chapter. It may not feel like system design at first, but typography and colors form the foundation of nearly every UI. Getting them right makes everything else easier.

1.12.21 Upcoming chapters

The upcoming chapters are being written as you're reading this. They aren't final, but expect to see chapters related to:

- Design systems: Turning a UI Library into a full-fledged design system.
- Designing large-scale app architectures. How to approach a highly modular app.
- Quality assurance: The role of UI Tests, integration tests, and manual testing. How to use these techniques for a higher quality app.
- A summary of all lessons in this book and how to apply them to your own work.

1.13 About the author



Tjeerd in 't Veen is a mobile developer since 2010, with a long history in iOS and working closely with Android developers. His career includes being a staff engineer at Twitter 1.0 and a iOS tech lead for the ING international bank. He has been delivering features across various platforms since 2000, and has been involved in hundreds of projects.

Tjeerd wrote the highly rated book called *Swift in Depth*, published by Manning.

You can find his blog on <https://www.mobilesystemdesign.com>. You can find him on Substack @tjeerdintveen, on Twitter, now X, at @tjeerdintveen or on Mastodon at @tjeerdintveen@mastodon.social.

When he's not working, he spends most of the time being a family man, a dad of two daughters, and noodling on a guitar.

2 Turning a Briefing Into a Strong Plan

In this chapter

- Being briefed for a new project
- Creating an approach from a lot of unknowns
- Ending up with a starting-point
- How to get a fuller feature set from a partial briefing
- How to make sure we focus on the right things
- Avoiding pitfalls when planning a new feature

It's a common routine: We receive some designs, some (vague notion of) requirements, and then we are asked "*Can you make this?*" followed by the fan-favorite "*When do you think it will be finished?*"

Somewhere in an alternate universe during a Systems Design interview, the question is not "*Can you make this?*" but "*How would you make this?*" After which you'll have to describe your thought process of turning a design and specs into bits and pieces and sometimes even pseudo-code.

Meanwhile, two interviewing engineers will look at you pensively, sweat drips down your forehead because even though it's *again* the same argument about how **import UI** is (not) allowed in a view-model, giving a good answer could mean you get a cool job and maybe even relocate your entire family to a new country... or not. The pressure is on.

It doesn't matter whether we're talking about a full-sized app or a tiny feature. Every time we get a design, we have to figure out how to turn pretty images and cool ideas into a real – albeit intangible – thing.

To some, it might seem easy when all we have to make is a tiny component in a toy app or tweak a button's width; But as soon as we are making an entire tech stack with abstractions, domains, and components shared between numerous features and teammates, then the landscape changes and we have to think differently.

Not to mention that you're in for a treat when dealing with (smelly) legacy code that nobody dares to – nor wants to – touch.

In this chapter, the book briefs you in a similar fashion. You will receive a design and then we are going to cover approaches. After that, we will try to get more (hidden) information to get a complete picture.

The goal of this chapter is to give you mental models on how to receive specs and a design. Then you'll end up with a sound plan for delivering the components needed for this feature.

But, we will start small for a couple of reasons:

- First, by starting small, we can focus on the process in a smaller setting, which you can replicate in both small and bigger applications.
- Second, to match the real-world experience; Often inside a company or during a System Design interview, you will be asked to work on a smaller feature-set consisting of only one or a few screens at a time, not a giant app from scratch.

2.1 The briefing

Together we're going to work on an app where a student can connect with tutors to learn new skills, such as learning to play guitar or learning to speak Italian.

The app is unreleased, but we can assume the project already exists so it's not 100% from scratch. We don't have to worry about setting up a new project.

With this product, tutors or coaches will check in with students via 1on1 calls. Between these 1on1 sessions, students would follow a plan – offered by the tutor – consisting of assignments or exercises to make sure they keep on track and keep improving.


Tutors will be the ones making these tailor-made assignments and can give feedback when needed.

Now imagine you're joining my team to build this, and I'm giving you a screen for this app, asking you to implement it. Don't worry, we'll do it together.

Here we can see the main screen a student would see. It has assignments, tutor information, a way to reach out to the tutor, some scheduling information, and a worksheet or todo list of recurring activities.

Courses +

Anyone can play guitar 🇮🇹 Italian Mastery C




Caleb Davis
@CalebGuitar Message

You're doing great! Be sure to slow down, accuracy is more important than speed.

Remember, practice makes permanent.





Next 1on1

 **Friday, 8:30 pm** Join call

[Reschedule](#)

This week's schedule Reset all

- Transcribe verse of Stairway to Heaven ➤
- Every day**
- Find all C notes on the fretboard ➤
- Practice intro of Smoke on the Water ➤
- Practice the G major scale ➤

 Courses  Messages  Payments  Profile

In the tab bar at the bottom we see more features in this app, but we don't have to worry about the rest of the app for now. With just this screen, we can apply a lot of principles.

2.1.1 An initial impression

Looking at the design, we can see it's already quite high fidelity.

NOTE: A high fidelity design is a term used by designers to indicate a design that's close to the final product. As opposed to low fidelity, which is more conceptual and wireframe-like.

This screen alone may look simple at first sight, but has some not-so-obvious features to make everything work.

For example, there is quite some linking going on; There are TODO items, and they have arrows hinting at details. This screen can open other features such as messages and opening a call. At this moment, we don't know if those buttons add views to the current stack, or if it's a deep link to a distinct part of the application. Perhaps the buttons link to entirely different apps.

Next, notice how this screen is populated with properly filled content. But what if we don't have anything scheduled? Or what if the tutor forgot to add TODO items? Or what if the tutor didn't add a callout message? Will this screen just be mostly empty space?

Notice how some of the TODO items are daily. Who resets these daily items? Perhaps the user does it manually, or should these items auto-reset themselves locally in the app? Or perhaps the server should auto-reset them?

All these things are unclear for now.

Next, what about tablet-support, or landscape mode, or dark/night mode? What about the data? Where does it come from? What about local storage or caching? What about error handling?

These are just the initial things we know we don't know. What about the *unknown unknowns*? We don't know what else we haven't thought about.

Long story short: **There are quite some *known unknowns* and *unknown unknowns* at this stage.**

2.2 Evaluating common approaches

So where to begin?

There are a million ways to implement an app. Despite that, let's take a moment to consider various approaches and why they may or may not be a good idea.

Warning: Opinions incoming. It's okay to disagree! But please humor me and follow along. Seeing differing approaches can give perspective.

2.2.1 Start with UI?

Would you open your editor and start making the UI and screen right away?

It's a common way to start, and it's heavily pushed on us non-assuming developers everywhere. Tutorials, blogs, online videos, they all show how “easy” it is to make the most exciting (toy) apps.

Which is great if you want to get started on an idea, or to learn, or to get excited about making something.

For your own projects, I definitely encourage you to start with UI. It's a lot of fun.

In a larger real-world app, however, the rules are a bit different.

When starting with UI, the pitfall is that we avoid considering multiple angles. We'll “just start building” without thinking about most of the required parts.

Eventually, you'll be painting yourself in a corner. You'd discover hidden features and requirements too late; Like building the walls and then realizing afterwards that the new homeowner would like some more windows.

By not planning, you'd risk slapping code on much later in a quick and dirty way, adding tech debt in an early phase.

Another pitfall is that by starting with UI, you might be pulled into discussions with teammates why SwiftUI is better for this than UIKit, or whether to use Jetpack Compose vs XML.

Just because we receive UI doesn't mean we have to start there.

Let's take a step back. We just established that there are already a lot of unknowns.

If we start with UI, we'd be thinking too locally. What I want to emphasize is to see beyond the UI and think in the bigger picture. Try to uncover all systems hiding in plain sight that are working together to make this app work.

2.2.2 A data-focused approach?

Maybe you'd start programming but focus on the data? You might start thinking of all the information that is required, stored, and passed around.

It's a good idea to think about what data is needed. It will make you ask good questions. Such as which fields are optional, which determines whether some UI components are hidden or which screen variations we need.

Thinking about data will also make you think about the foundations required to build a feature. Thinking about data will also make you think about persistence, networking, and caching.

These are all important topics in a job interview and building features.

There is a pitfall: Don't get carried away into creating a perfect working application with the perfect names, types, structs and fancy testability.

Let the data guide you to think about what needs to be stored, passed, and presented and use that as a starting point.

2.2.3 Creating an app-skeleton or flow-skeleton?

Assuming there is no app yet, would you be making the entire app skeleton first? Such as settings up the entry point (e.g. the Main class or AppDelegate), navigation bars, tab bars, maybe add placeholder UI with empty screens?

Without focusing too much on UI or features, having some sort of skeleton app is a productive way to sketch out how everything will fit together. At this stage, it's cheap to add, delete, and update screens and flows.

Using placeholder screens, you get a good impression of the feel of the program. Just make sure you don't get lost in the details yet.

Placeholder screens are a great idea when receiving a flow. However, in our scenario, we can focus on a single screen.

2.2.4 Starting by making components or features?

Would you focus on making independent components? Such as a scheduler, todo list, and messaging service? Or maybe UI Components? Such as buttons and other views.

This can be a good idea, but we still have to think about how it all comes together as a whole. At this stage we don't know all the requirements yet.

I would avoid focusing too much on single components. Otherwise, we risk getting carried away by making the perfect views or nicest todo list (we wouldn't be the first developer to get carried away).

2.2.5 Drawing a diagram?

Would you draw a diagram of this feature and see how everything is connected? Maybe to help decide what to build first?

Sketching out a diagram encourages thinking which bits and pieces you're going to need.

It's not a mandatory step, but it definitely helps you think about the whole system.

In a real-world application, that's a great idea. During a System Design interview, coming up with a diagram is also how you'd be communicating your ideas to the interviewers.

In a team-setting, I **strongly** recommend drawing some sort of diagram to make sure everybody agrees. In this book, that is also what we'll do. We'll use the diagram as a vehicle to make decisions at every step.

2.2.6 Decide on an architecture?

Would you perhaps start with an architecture in mind, thinking of maybe using a reactive approach, or decide between MVVM (Model-View ViewModel), MVC (Model View Controller), or MVP (Model View Presenter)?

NOTE: Often when app engineers talk about architecture, they are referring to the part that glues the business code to the UI code. Not the entire, global, architecture of an app.

Here is my advice: **Stop thinking about architectures**, we haven't even written a single line of code yet. Architecture is for much later, if at all. Because maybe all we need are a few minor components.

So far, we've only seen one screen. So if someone blurts out "This is a great use-case for reactive programming!". Then it signals they haven't really thought about the problem deeply enough yet. This is a red flag that someone is too married to "the one" architecture that they're comfortable with.

Picking an architecture would be a next step, but not the first. We have to consider an architecture that works best for this app, not what we are comfortable with.

You might catch yourself leaning towards a single architecture regardless of what you have to make. In that case, try to check in with your biases and experiment with other architectures and programming languages to expand your horizons.

There is no silver-bullet single architecture, only trends and best practices.

In our case, deciding on an architecture is a bit too early stage. We'll get there if needed. For now, let's agree to let our architecture grow organically the more we understand the problem.

Maybe, for instance, we need to make local architectures that differ for each domain. For example, maybe we want to make a specific declarative architecture to deal with the Calendar part and scheduling events. Maybe a coworker wants to use a different mini-architecture for offline-storage functionality.

Then maybe these mini-architectures will grow together in one large overseeing architecture in the app itself. Maybe it will be complicated to tie it all together, and maybe it won't, let's park that decision for later.

2.2.7 A recommended approach

We've gone over the pros and cons of some common approaches. I'm sure there are more approaches you may prefer.

There is not one perfect way to attack this "problem". There are plenty of approaches to take. Many are valid, some are problematic.

The bigger takeaway here is to try not to get lost by details too much. We're in the sketching-phase, we need a general outline of what we need and we need to uncover hidden requirements and we are dealing with *known unknowns* and *unknown unknowns* right now.

In terms of UI, we wouldn't start with fancy drop shadows and animations before we even have a functional screen, right? No it's better to get some parts working first before we make it look pretty.

So let's forget architectures and discussions about SwiftUI vs UIKit or Jetpack compose vs XML, forget thinking about putting data in controllers vs viewmodels.

Let us not argue about why viewmodels do (not) belong in declarative UI. We can defer those discussions until later. This will give us a better understanding of what we're going to build.

No matter the approach you prefer, I hope we can at least agree on one thing:

The best decision at the first stage is to understand the problem better.

Understanding the problem better will give us a better starting-point, because it will help us uncover wrong assumptions and missing features.

So that's what we're going to do right now. Let's begin by drawing a diagram.

2.3 Sketching out a landscape

We will create a landscape, which is a collection of domains and components that will be required to make our feature work. We can think of this as our architecture, but instead of "choosing an architecture", we will let it organically grow and evolve.

First, let's get the obvious features out of the way. This will get them out of our heads, and then we can focus on the not-so-obvious secondary requirements.

We see that the screen has some functionality staring in our face since it's UI and thus customer-facing. There is an avatar, some sort of Todo list, a way to open a 1on1 call. But, we don't know the details yet.

Let's note down the UI-based features in plain text, so that we can turn these into a graph after for a better visual representation.

Feature set:

- Some sort of TODO List
 - TODO's are recurring (by week or day, maybe others?)
 - TODO's can open a detail (details unknown)

- A Tutor profile
 - Has avatar and name
 - A tutor has some sort of dismissible callout
 - Some sort of messaging feature to contact a tutor

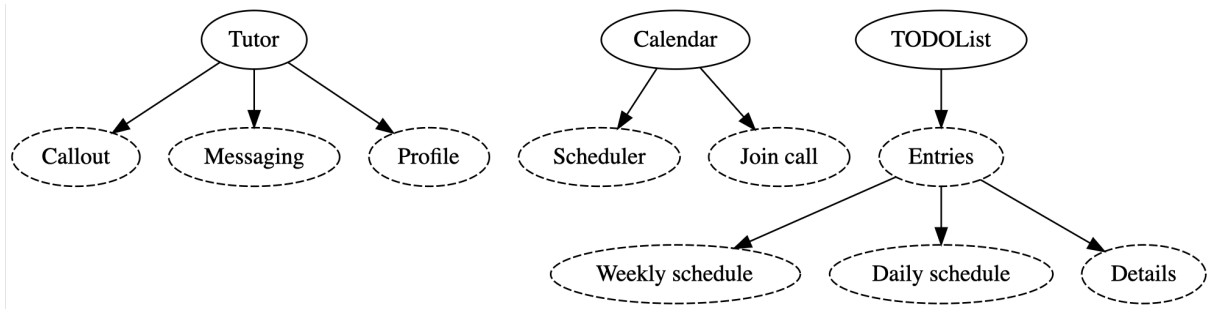
- Some sort of calendar functionality
 - There is a summary of planned 1on1 calendar-events
 - Ability to reschedule
 - Ability to join calls

- Swapping between courses: Up top, we see that the user can open a different course with a different tutor.

We are sketching here. Hence, we call the feature "some sort of..." since we don't know yet exactly how it will work.

For example, we see a calendar event for a meeting. Should we call it a **Meeting** or **1on1**? Or maybe **CalendarEvent** or **Scheduler**? Let's go with **Calendar** since it's an umbrella term for scheduling events, rescheduling, linking to moments in time, etc. We can always rename it if needed. Then **Calendar** can help us join an event or call, and maybe it can help reschedule with some sort of **Scheduler**.

Let's turn the aforementioned feature set into a diagram (we'll deal with Swapping tutors in a bit).



NOTE: It's okay and expected to start naively. There is no pressure to figure it out perfectly in a single attempt. We're brainstorming.

Notice how **Tutor**, **Calendar**, and **TODOList** have a regular outline. This means that they are decomposed, they are broken down into child nodes. If we wanted, we could start implementing them roughly, despite their child-nodes not being figured out.

Then, one layer below, we mark features with a dashed outline; Dashed means that components aren't either fully figured out yet, or they are not decomposed yet. It's a way to signal that they aren't "done" thinking about.

For instance, the **TODOList** -> **Entries** connection isn't figured out, since its requirements are vague as of now. Are they stored locally? Do they need an **API** call? We have too many unknowns to mark them as *resolved* with a regular outline.

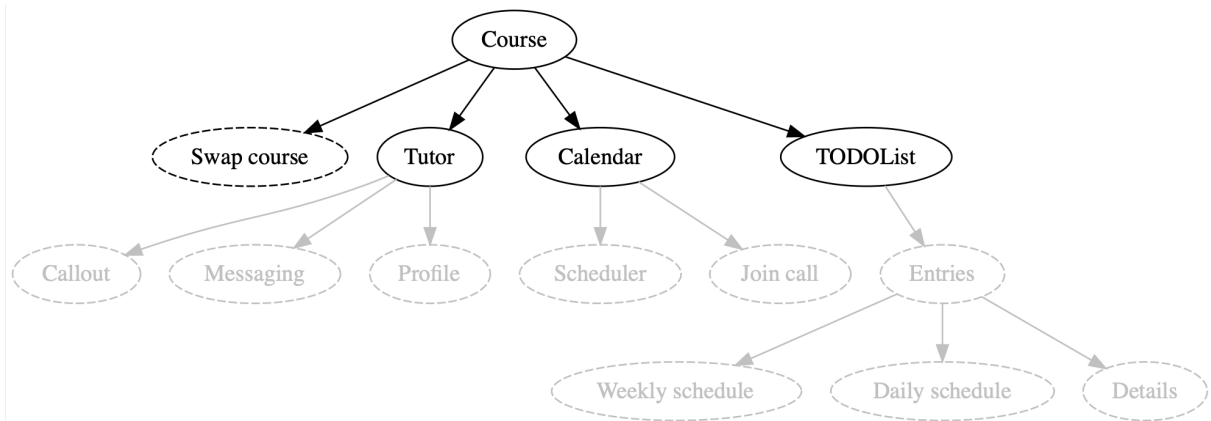
Going beyond that, **Entries** link to **Details**, what does **Details** show? We don't know yet, it's not important for this screen. All we need to know is that it does "something", so we mark it as dashed as well.

2.3.1 Everything is connected to a course

If we zoom out a little more, we can say that all these features are connected to a course in a way.

In other words, a course owns, **Tutor**, **Calendar**, and **TODOList**. On top of that, there is also an option to swap courses.

Let's update the graph to represent that these components all belong to a **Course**.



Looking at **Swap Course** we can see it's a child node of **Course**, we haven't really figured swapping courses out yet so we leave it dashed.

Consider swapping courses a low priority feature, because why worry about supporting multiple courses or tutors while we haven't even gotten a single course to work?

We do know we need it, so we add it to the diagram and mark it as dashed. After we talk to the designer we may learn more about this feature.

NOTE: Remember, we are sketching and iterating and learning about the requirements. It's okay if it's "good enough" for now. Try to turn off that perfectionism internal dialogue (if you know how, please share your secret).

2.3.2 How far do we decompose?

If we were to keep zooming out, we'd see the entire app connected as one giant graph of components depicted as nodes.

You may wonder how far to keep decomposing.

A rule of thumb is: *Decompose until you feel you understand the problem well enough to start the implementation.*

There is no need to keep decomposing until we dissected components into ones and zeros.

At this stage, the art of a technical design is jumping from feature to feature, from requirement to requirement, and figuring out which components to make. It's important to keep it high-level, so we don't lose time focusing on details.

Make it clear what to focus on and acknowledge that there are unknown components that you need to resolve later.

Before we start implementing this feature, let's continue figuring out requirements and components until we feel confident enough to start.

2.4 Uncovering secondary requirements

We now have an approach; We are creating a landscape graph where we decide what to make and what to figure out.

We focused on “obvious” UI features that were staring us in the face. However, UI is only part of the picture.

We *could* start implementing now, but our job right now is to uncover hidden requirements, find edge-cases, not just what we see in the UI.

If we don't make a plan, and “just start building” it will hurt us later. Such as having to redo features because we uncovered an important detail nobody thought of yet. Or worse: We'd focus on features that aren't needed at all in hindsight, throwing away weeks of work.

Let's focus on the not-so-obvious parts and move on to secondary requirements.

Don't open Xcode or Android Studio or whichever IDE you prefer. Don't start programming. I know, I know, it's where the fun happens. We'll get to that in the next chapter.

Unfortunately, taking a moment and talking to people has priority now.

NOTE: “Fun” fact: Did you know that the more you get promoted, the less time you spend programming and more time talking to people in meetings?

We should take some time and uncover unknown requirements. We do this in a few ways:

1. Think about and figure out all components (not just UI) that are needed to build the feature. It sounds obvious, but many can't resist programming right away.
2. Figuring out missing functionalities or requirements; Such as by asking the right questions to teammates with various roles and disciplines.
3. Trying to challenge the design, and think of ways that the design would break, uncovering new requirements.

Chances are, we'd uncover details or missing elements that others haven't even thought of yet.

There may be details lurking that can be important to know before we even write `class Course`. By getting a full picture of the problem-space, we might even make sure that our team iterates over features before we start writing code. It might surprise you how often you can uncover an important “*Oh we didn't think of that*” which affects both UI and backend.

NOTE: During System Design interviews, you can't go back and forth with a designer. But you can communicate the unknown details and missing information. This shows the interviewers that you're understanding the problem on a deeper level, and are considering edge-cases.

2.5 Working with Designers; Getting secondary features

Let's approach talking to fictional designers, so that we can uncover more requirements, and as a result, come up with a better design.

The goal of talking to designers is threefold:

- First, to make sure we understand the problem better.
- Second, we want to make sure they understand the problem better, too. By offering some technical perspective.
- Third, we want to challenge the design itself. Not to annoy the designer, but to make sure we find edge-cases, (de)prioritize UI components, and uncover situations that nobody has considered yet.

With some good input, you can trigger a small iteration of the design before you even begin.

A productive conversation should result in an improved design and a better understanding of the feature in the team.

NOTE: Keep in mind that the designer is your ally! You are improving the product together. You're not here to be their devil's advocate and critique their work.

While you're at it, inform designers about the way you work. Tell them that the UI — although functional — probably won't look nice in the beginning. Remove their worries that you will not finish the details later. Fancy UI details might not be the highest priority for us, but it often is for designers.

Working closely with the designer has a lot of perks, one of them is minimizing the communication gap. The worst thing is to not communicate together, wait for weeks, and have the designer give you the *final-final3-final-reallyfinal.sketch* design file for you to implement.

It's vital to work closely together to create a better design and plan.

Let's go over some talking points so we can learn more about the feature and update the graph where needed.

2.5.1 Whether or not a design is the "law"

You may wonder if a design is “the law” and should be 100% followed, or if it’s more of a communication tool, ready to be interpreted.

The designer could deliver everything with a low fidelity (low detailed) design, or even in plain English “*Make a screen with an avatar top-left, and a todo-list at the bottom, and ...*”

This approach is too hard to align on. Maybe that’s how we will work in the future with AI generating apps.

More commonly, we receive a design that brings us closer to the final product. But it isn’t the final product – we aren’t shipping images to customers after all.

We can’t assume a design captures all variations, even when a design program supports code.

When you multiply all supported device sizes *times* platforms (e.g. phone and tablet) *times* dark mode *times* light mode, *times* all font sizes, the design file would be huge. It doesn’t make sense to design everything upfront.

Even if a designer gives us those variations, the design wouldn’t define everything still.

A design doesn’t convey things such as animations, all language variations, or how the screen would look with a slow network connection, or what the experience is like when the customer has to retry a failing submit form.

We can get close, but the design is still an approximation of the final product.

2.5.2 What is “pixel perfect”, really?

When we say we’re delivering something “pixel perfect”, we usually refer to making the UI exactly like the design *where we can*. We will pick the exact colors and border widths, and we will position the elements for a specific screen. We might overlay the design over the app and pixel peep to make sure *everything* looks the same.

However, there is room for variables, where each screen will look different because of its content and environment. The app might have dark mode enabled, or use a Right-To-Left language. Or use larger font sizes for accessibility. With little effort, we already deviate from these undesigned variations.

That means that “pixel perfect” is often an approximation. Even the way apps render shadows differ per platform, so that’s not “pixel-perfect”, that’s “as close as we can get that makes the designer sign off on it”.

The design is a communication tool depicting the final implementation. It’s a plan of what to make.

At one point we have to decide “This design is enough for us to agree on things and get going”. It can’t be the absolute law for everything.

For a smoother process, use the design as a starting point. For all deviations, be sure to include the designer during development while iterating. Make sure they agree on decisions, such as supporting large fonts, or small fonts, animations, and anything else that might deviate.

2.5.3 Designs often encompass best-case scenarios

A designer will usually deliver neatly filled screens. They’ll pick a nice stock photo to fill the necessary images and they’ll make sure all text fields will contain a full ‘lorem ipsum’ text. It’s the perfect-looking content for the perfect screen.

But then real life comes into play. Real-life data hits different. It’s random, dirty, ugly, and all over the place and that may not always fit the intended designs.

Once you let people upload their own pictures and add their own information, the app won’t look like the design. We can’t guarantee that the — once aesthetically pleasing — screen won’t look as appealing in practice.

An avatar might be missing, or perhaps it’s too low-resolution. The descriptions might be *way* too long and half the fields might be empty. Not every user takes great care in filling in their details.

If texts are missing, is that okay? Here, we need to make sure to check with the designer that it’s a viable option. If not, there’s a strong chance that the UI might break with imbalanced content.

As a rule of thumb: **Ask for a worst-case scenario design with poor content and see if the design still holds. Even with the worst content imaginable, the design should not break.**

As a developer, thinking of a particular design isn’t necessarily your job. However, you are working on this project together and poor content is a truth we can’t avoid. So, it’s better to find any issues now preemptively versus facing potential complications down the line.

2.5.4 Not everything has equal priority

A surefire way to deliver less efficiently is to accept anything in a design as absolute, uncontested truth and implement it as if to blindly follow orders.

But not all ideas are weighted equally. A design is not set in stone; It’s a preliminary plan or proposal of what to make. Like all designs, it is subject to iteration. With that in mind, you’ll be the one making it come to life – albeit behind a pane of glass.

When receiving a design, don't assume all components are equal in terms of priority and magnitude.

It may sound counterintuitive, but finding ways to *not* build features can ultimately lead to better outcomes for everyone, including the designer. Getting a team to agree to prioritize features will allow you to ship more quickly. As a result, the team — including the designer — will receive insights from customers and other stakeholders within a shorter time frame.

You will also be able to focus on the core features that we absolutely need to ship, versus those that are more accessory.

Taking a proactive approach to learning at the early stages of the development cycle allows us to change course more quickly as needed. It also helps us avoid the potential risk of throwing away valuable time building features that customers didn't actually want.

For example, in our case, the assumption right now is that people can have multiple courses (tutors) at the same time. In reality, usually someone is learning one major skill at a time (e.g. learning Spanish). But it's rare that someone wants to learn Spanish, German, English, and Korean simultaneously.

In the design we see someone has two courses (Guitar, and Spanish), two courses are more than enough for most users.

Even if there is a strong disagreement with a designer about supporting only one course; Practically speaking, we need to support a single course before we can even build support for multiple courses, anyway.

So the question is less about “*will we support multiple courses*”, and more about “*will we ship with support for one course first?*”. It's about prioritization.

Don't underestimate the difference in time investment. A designer might add a multi-course feature within a short time, but implementation may take days or even weeks.

When receiving designs, it's the perfect moment to stop and think mindfully and critically about what's *really* important for the user. Re-evaluating priorities together with the designer mutually benefits everyone in the long run.

2.5.5 Verify the existence of pre-existing components

After receiving designs, be sure to run the components by other client engineers. Be sure to check what they already know exists in order to avoid reinventing the wheel.

It may seem completely obvious, but yet doesn't always happen in reality; A single question in a Slack channel, such as “*Does a component like this already exist?*” can save you days of unnecessary work.

Conversely, maybe there already is a component that you potentially could use, but it may be just slightly different from what the designer gives us. This is a telltale sign that there is some plausible misalignment between the client libraries and designs. If so, *this might be a good time persuading the designer to use what we have.*

If there is a sound reason to make an adjustment to the existing component, or a new component altogether, then so be it. However, a little pushback for shipping more quickly can be a tremendous timesaver, yielding other indirect benefits. These include, but are not limited to, avoiding having to maintain more duplicate components, which is a hidden time sink that compromises productivity.

2.5.6 Ask general UI questions

These are a repertoire of general questions you can ask which effectively apply to most UI-based projects.

Try to think of things that the designer hasn't thought of yet. This helps us find requirements that we could miss.

Some examples are:

- What if there is more information than fits the screen? Will you make the screen scrollable? Or will you resize elements?
- What does the screen look like when it's empty? (In our case, what if the tutor has not filled any information for us yet?)
- Have you thought of large font sizes on small devices? Will the screen break?
- What would the screen look like with long labels and/or verbose languages? E.g. German needs longer text. Will that fit?
- Have you thought of tablets? Will the screen look too empty?
- Do we support landscape mode?
- How will we treat errors? Just throw an alert or something nicer and more inline?
 - What about partial errors? E.g. the tutor data is loaded, but the TODO List can't be loaded. Will you show partial errors, or will you throw an error for the entire screen?
- Do we support dark/night mode?

2.5.7 Ask functionality-related questions

After you've exhausted your general UI questions, you can get even more information by trying to find edge-cases that might break the screen.

Try to come up with ways the feature might work different from intended. Like the general UI questions, asking these questions will help us find potential problems early on.

These questions would be feature-specific, such as

- When a user completes a TODO item, do we assume it sends that message to the server right away? If so, what if that network call fails? A giant alert might be too much. Can we use some sort of notification or toast? And will the TODO reset exactly?
 - If the network call fails while the user backgrounds the app, will we let it fail silently or will the app send a local notification?
- Do all TODO items have a detailed screen? Or is that optional?
- Are all TODO's always scheduled? Or can they be unscheduled TODO's without a deadline?
- What if the tutor hasn't made a plan (yet)? What will the student see?
- What if you haven't selected a tutor yet, what would the screen look like?
- What happens if a tutor has an extremely long callout message? Should it be cut off, or expand on press? And at how many lines?
- Are the daily TODO's automatically reset? And when would that happen, maybe at midnight at their local time zone? Or after X time, such as after 24 hours?
- If I press "reset all", would the user get a warning? Some sort of alert perhaps?
- If user dismisses the tutor's callout, will they lose it forever? Or can they get it back?
- What happens if a user joins the Calendar event before it's ready? Can they join an empty meeting? Or do they get an error?
- Will Calendar calls be a link or handled in app?
- Will messages be a link or handled in app?

By imagining you're using the app, you can come up with practical questions to get you and the designer thinking more in-depth about the functionality.

Perhaps you'll get a couple "*I didn't think of that actually*". Which is good, because that's exactly what you want to hear *now* as opposed to *later* after most of it is implemented.

It also makes you think of a problem more deeply, thus giving you a better understanding of what to build. As a result, you will get more “ownership” of the problem. In essence, bringing you closer to being the expert on this screen.

During a job interview, it’s important to show all the missing details that you’re thinking of. It’s a way to impress interviewers by showing you think of a lot of non-obvious parts and secondary requirements.

2.5.8 Talk about error handling

Error handling is often a low priority for designers and developers alike. Within good reason, it’s more important to get a feature working first.

But a real problem is that people don’t often think about the errors in the initial stage. That’s problematic since it can negatively impact design if it’s added as an after-thought.

If you don’t think about errors early on, chances are, you’ll deliver an app with a giant alert obstructing the view, stating "Something went wrong". Which is a bad user experience and sort of a "last resort" error.

Push towards using errors that don’t block the UI, such as toasts (little notifications) or a view having a special inline message. Avoiding blocking the UI where possible for a better user experience.

You may need to work with the designer to help them understand where and when errors can occur. Maybe your UI only has a single point of failure, or maybe errors can be displayed inline per individual component that loads.

2.5.9 Talk about time-investments and start thinking in a less binary fashion

In some cases, maybe you’re tempted to avoid implementing specific parts of a design. For instance, the designer might have a custom navigation bar that’s quite difficult to get just right, and it may be even harder to maintain.

It’s easy to fall into the trap of thinking in binary, black and white terms such as “worth implementing” vs “not worth implementing”.

For instance, you might be great at devising arguments which demonstrate why a custom navigation bar is a bad idea. Despite the fact that, on paper, the points you’ve made appear objectively correct – making a custom navigation bar is almost always a maintenance headache – To others, you may be unwittingly labeled as a “rigid” developer.

However, don’t jump the gun and reactively say “No.” Instead, shift the conversation towards focusing on priorities and timelines. Instead of saying, “I wouldn’t do that”, you could alternatively say: “This will be x weeks more work for us to do.”

From that point on, as a team, you can then decide if the custom navigation bar is worth all that extra time. Because, maybe it *is* worth it to the company to have a custom styling to maintain a central theme in the application.

The point is to quantify the consequences and make decisions and alternatives more tangible.

2.5.10 Giving feedback to the designer

Not everyone can take feedback like a champ. I'd argue it's difficult for most people. As developers, we are more likely to be battle-hardened by having our work critiqued day after day, multiple times a day, but it always stings.

However, for designers, the process is different. They design for days or weeks on end, maybe even wait a week to finally get approval from the "design systems council", then they iterate some more, and then finally, after hard work, they will showcase their work to the rest; After which *the entire department* will have opinions about their work. *"I wish we used the colors from before" or "I wish the border wasn't so strong" or "Why doesn't Google use drop shadows yet we do?"*

It stinks for designers to be critiqued by experts and non-experts alike. Everyone has an opinion on UI, yet not everyone has an opinion about code.

No matter how often people say they don't get attached to their work, *be sure to not only critique UI but balance feedback with positive remarks as well, and try to keep it objective.*

2.5.11 Updating the landscape

After talking to the designer, we learned a bunch of things for our project, such as

- The app is for phones only, not tablets.
- Dark mode is needed but we'll do that in a later stage. (We mark it as lower priority)
- The weekly and daily schedule auto-resets. (We don't know yet if that happens on app or backend and how often)
- Scheduler opens a picker (There is no design yet)
- After proposing a schedule, the other party must agree. If the other party doesn't and a schedule has passed, the proposed schedule is deleted.
- We agree that we won't support multiple courses for the first version.

Talking to a designer will uncover a lot of details that won't make it into our landscape graph. But for our purpose, we will focus on a few things that stand out which we'll cover next.

2.5.12 A fast app is key

By thinking about the problem more and talking to hypothetical designers, we uncovered more important pieces of the puzzle.

For instance, the designer shares that users want to hop in quickly, check off TODO's, and leave the app again.

We realize that if the app would be fully dependent on the server it would mean the user would open the app, log in, fetch data, hopefully not get an error, and then finally see the TODO List, check off a single item and leave the app, hoping the network call succeeds before the app backgrounds or is killed.

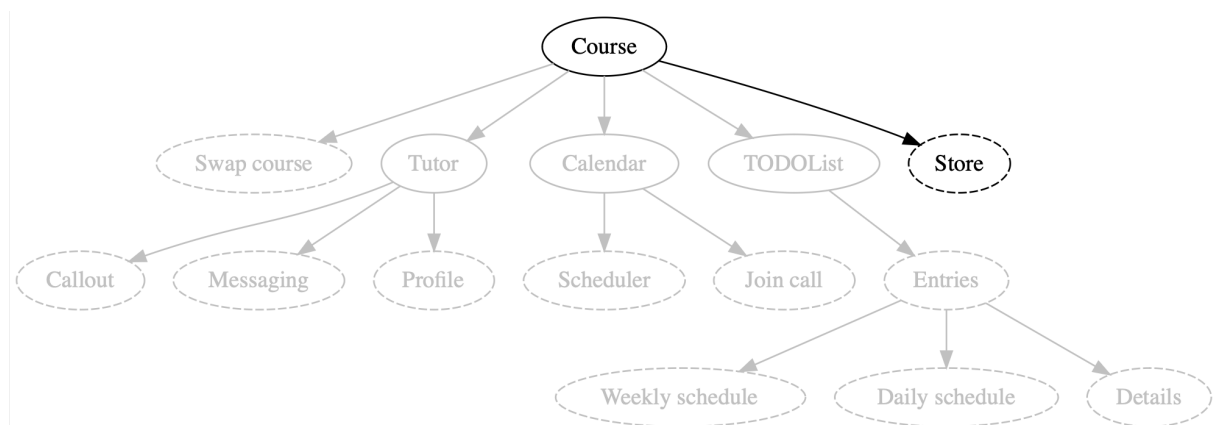
We can translate that requirement into a “thing” we need to make, or a Non-Functional Requirement. In our case, that’s offline-mode support in the shape of a persistent store. So that the course and its TODO items are always available, making it less dependent on a stable network connection.

We don’t know yet whether this store is going to use MySQL, NoSQL, or an insecure text file.

The designer or customer doesn’t care about the details long as it works. It’s our job to care about *how* it will work, but not at this stage. Because at this stage, we need a good idea of what we’re about to make, but we don’t need all details yet.

Let’s update our landscape and add some sort of persistent store component that we need to offer for *Course*. We’ll call it *Store*.

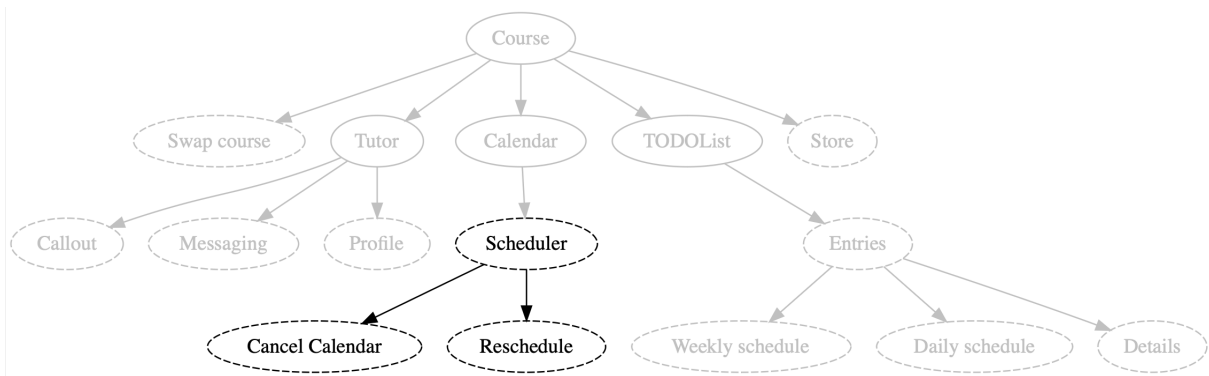
Because we don’t know the details yet, we mark it as dashed.



2.5.13 Scheduler

We also learned we need some way to not only reschedule, but also *cancel* calendar events.

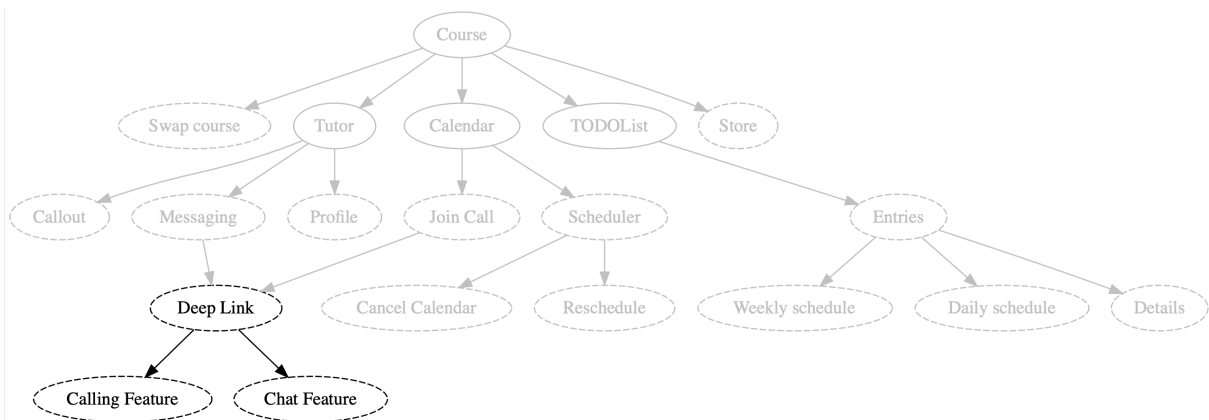
The designer confirms that cancelation is out of the scope for this screen, since rescheduling or canceling triggers a new flow. This means we can write these features down, but we don't have to focus on it too much. We mark them as dashed in our landscape graph.



2.5.14 Deep Linking

We also got confirmation that features such as messaging a tutor, or opening **Calendar** calls with the tutor, will happen in other parts of the application. For now, we can assume these will be deep links, so we add that to the graph.

Again, we mark these as dashed since we know they are needed *at some point*, we just don't need to figure them out at this stage.



2.6 Aligning with backend engineers

Thanks to talking to the designer, we have become more aware of secondary (hidden) features. We can follow a similar process by talking to the backend engineer.

This way, we hope to uncover more secondary requirements and important details related to data-flow between backend and client.

In real-life, the backend engineer can already link to documentation for you. In our case, there are still bits and pieces to fill specific to this screen.

In this section, we'll go over some tips to make the integration process smoother. At the end of this chapter, we'll update the landscape again with new feature-specific requirements.

2.6.1 Align on about User sessions, environments, tokens, and timeouts

Once you start talking to a backend engineer about which environment to talk to, it will be hard to avoid mentioning login tokens and user sessions.

Be sure to cover that, get the necessary information to make backend calls as soon as possible. Because you'll learn right here and now of any problems that you will run into later.

For example, maybe there isn't a staging environment, or maybe that user account they made for you doesn't have the proper access that you need.

Or maybe you need to request user-permissions and the team approving it is slow. It's better to know these limitations earlier than later.

Ideally, you can already experiment by making an API call from the command line (such as by using the `cURL` command line tool) and try to make it work. Then, if you have trouble integrating an API call from the app client, you can verify if the `cURL` call will work. Saving you time since you can rule out if the problem is on the backend or not.

Then there are more feature-specific features. Such as obtaining a login token, how can we get one without a login screen (it's not built yet).

Think about timeouts. Let's say someone tries to complete a TODO item after being logged in too long. Does that trigger a timeout? If so, what kind of errors will we get on timeouts?

If timeouts are a requirement, then most likely *all* API calls must work with this mechanic and the app needs to respond to it, which can get complicated. Something to keep in mind.

2.6.2 Align on consolidating network calls

Regarding our feature, check if you want a single API call to populate the screen, or if you need to make multiple calls to get a populated screen. Multiple calls is easier for the backend, but might complicate things for you since you have to combine them.

An argument to ask for a consolidated network call (assuming GraphQL isn't an option) is to think about multi-platform implications.

Let's assume that you require multiple API calls to fill a screen you're working on. Then the iOS app needs to make some sort of logic to accommodate for this. This may be the same time investment for backend, so it's not enough to convince a backend engineer to take on this extra work. If you ask, you might be told that it's not an option and endpoints need to be pure and semantic. (Sometimes people just don't want to or have time to do extra work, who knew?)

If bribing the backend developer with coffee doesn't work, consider shifting the conversation towards time investments.

For instance, let's say we are also making – or planning to make – an Android app. It's the same problem again, except Android engineers now *also* need to invest time for this. Now let's say we also make a web client. It's the same problem *again*.

If you're going to make the same investment multiple times, it's not that hard to convince a product owner to push for making this investment *once* in the backend, as opposed to *thrice* on clients.

2.6.3 Be on the same page with errors

Next, consider how errors are handled. Will there be a single endpoint but with granular errors shown for each data model? Or will you get one generic "something went wrong" error?

A common pitfall that sometimes gets overlooked, is ignoring the fact that backend-errors are localized on the client. It may sound obvious, but experience shows that it can easily be forgotten when it comes to handling errors.

For instance, sometimes a backend API will give plain English errors, such as "Could not load the user's TODOList". Which is fine for us to help debug issues. However, this should not be customer-facing text.

We need error codes. Because, as a client engineer we can localize them. As an example, we can turn "error code 11" into "The TODOList couldn't be loaded" for English, and "De TODO lijst kan niet worden opgehaald" in Dutch.

As a result, you need to align with backend engineers on the *meaning* behind error codes. You can propose a list where each code represents an error and their app translations.

2.6.4 It's okay to deviate from backend custom error codes

A backend may already have error codes, code 11 meaning "The TODOList couldn't be loaded" or code 12 meaning "The user session has timed out". But you don't have to align them fully with backend,

because *on top of* backend error codes, you will also have client error codes related to the backend call.

So keeping your error-codes in sync isn't always possible.

For instance, maybe the backend will *not* give you an error, but then you try to parse the data and *that* gives a client-specific error. Or, perhaps, the client will have a network timeout which isn't necessarily backend-specific.

So you will still need error-codes *on top of the backend ones* that are not shared by the backend.

2.6.5 You might be the backend guinea-pig

If you're making the first client talking to the backend then the process is likely *much* slower, and you have to take that into account.

In that scenario, you implicitly take on a second role, which is to be the backender's betatester or guinea-pig. You will be the first "customer" that will make use of a new untested, beta version of the backend.

For instance, when implementing API calls, you may receive an error with a vague message. Or maybe you're treated with cryptic 500 errors. Either way, you will depend a lot more on a backend developer to fix issues, before you can move forward. As opposed to integrating the second or third client where a lot of issues have already been ironed out.

Often when you hit an issue during integration it will be unclear where a bug is; Is it a client bug or backend bug? Now it probably takes at least two people to debug it. You'd have to look at both the backend and client to find the issue, which is quite a large scope of "The bug should be anywhere around these parts."

For this, cURL is a good tool to verify if the network call works well, even without a mobile client. So you can narrow the problem-scope and establish (or rule out) there is a client problem.

Alternatively, let's assume there already is a working Android app and now you're making the iOS app. If your iOS app has trouble communicating with backend, the scope of the bug is much narrower. You know it's most likely a bug on the iOS side since the Android app is already working.

When planning, take it into account whether you're the first integrator; If you are the first integrator, as a rule of thumb, you should double or triple the time it takes to integrate something.

2.6.6 Read code from other client implementations

If possible, you can considerably speed up your backend-integration by checking out the source from other clients that already talk to the backend.

Verify if there already is a Web client or Android app or something else like a headless client (command-line interface only). Inspect how they solved similar problems. A little web debugging goes a long way.

It's okay if the programming languages used are unfamiliar to you. As long as you're able to get the gist of it, you'll be fine. There is no need to be afraid to ask for help, most engineers are excited to share how they solved a similar problem.

2.6.7 Consider push notifications

Although not part of the UI, push notifications are part of the UX (user experience).

We might need to support certain messages, such as when a tutor left you a new message, or when you received a new schedule. Either way, there should be some way to inform the student.

To make push notifications work, we also need to think about registering a device for backend.

Push messages aren't always localized in each app. If a users' app is set to French, we should avoid sending English push messages. It's not a great experience, so you need some sort of mechanic to submit a device's language to the backend.

For push notifications to be localized, we need to send a user's language settings to the backend.

Although this is not specific to our feature, this is relevant in most apps, so worth to keep in mind.

2.6.8 Feature-specific questions

Next, let's think about feature-specific questions that we can ask to uncover secondary requirements, such as:

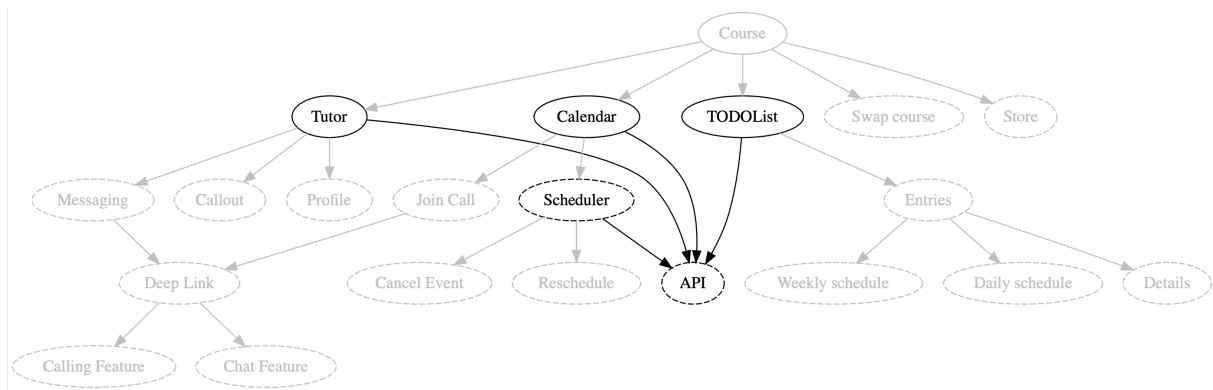
- Which fields are optional?
- Will I get all data at once or do I need to assemble it?
- How will the app submit the TODO items?
- Assuming some TODO items are reset every 24 hours, how do we communicate this? Does the backend know which timezone we're in, how?
 - Maybe the client needs to send their current timezone?
- What format will we use? JSON, GraphQL, Protocol buffers, something else?
- Which data is something that customers fetch every time? If so, maybe we can cache some locally?

Depending on the context you can get as detailed as you want. Talk about security, caching, and so forth. But note that at this stage, we haven't implemented anything yet, so it's good to get a big picture, but don't get lost in the details at this stage.

2.6.9 Updating the landscape with backend requirements

We learn that a few things from this screen will be fetched; The TODO elements, the tutor's profile (with its callout message), and the schedule information. All these need some form of network (API) calls.

Even though we know we need an API for the network calls, at this stage we don't have to worry about the implementation details yet. So we can add it to the landscape graph and mark it as dashed (to be figured out later).



2.7 You are the link between backend and design

Even if you are on the same team that regularly has standups. As a client engineer, misalignment details and misunderstandings will come out during *your work*. Because the designer(s) and backend engineer(s) might share only general information during team-meetings, or they may work in silos, or they might be working out the details in their own domains and only mention those to you. Now you'll be connecting the backend APIs to the UI, making you the missing link between them.

There will probably be some misalignments because of this, and it's best to assume it's your job to catch these misalignments early on a detailed level.

Try to think of all data that needs to be communicated and how that reflects in the UI. Try to find those edge-cases.

Optionality is a big one. It's a small thing to say "A name may or may not be filled in", but if the designer assumes data is always present then it might break the UI.

Or vice versa, a backender might make wrong assumptions about the data. For instance, they may think a tutor will *always* have a full name, but it might not match the customer's and designer's expectations. Maybe some tutors use aliases and the backend engineer wasn't aware. Now they have to update the codebase, APIs will return different values, documentation needs to be updated, and so on.

To emphasize: **you are the link between backend and design and will be the one uncovering wrong assumptions.** Find misalignment now at the early phase, as opposed to later during implementation.

2.8 Closing thoughts

We've only received one screen, yet there is already a ton to consider. By taking some extra time to think we have a more rounded idea of what to make, resulting in a landscape graph which we can use to express the components and domains of our architecture.

This approach might be overkill for a personal hobby project. But I hope you have gotten some useful points out of it that apply in a regular work setting.

Even though we didn't start programming right away, in the end it's about saving time and working on the right things. It's better to save time upfront than implement code and having to refactor most of it because we didn't plan accordingly.

In the next chapter we will actually start implementation, which is where the real fun begins.

2.9 The takeaways

In this chapter, we covered:

- Try to understand the problem better before starting to code.
- Drawing a graph with limited information helps design a component landscape.
- That it's okay to *not* have all the answers at this stage.
- Talk in terms of time investments, not in what you should, or shouldn't, implement.
- If the feature already exists on another platform, talk to developers of that platform and ask to inspect their code.
- Aligning about optional data impacts both the data model and the design.
- Check if timeouts impact your feature. Such as staying logged-in too long.
- Ask what errors you can expect.
- Check if partial errors are an option, such as partially loaded data. This impacts both network calls and design.

- You are the link between backend and design. Save lost time by finding misalignments during briefings.

Design

- A design is a communication tool, not a representation of the final product.
- Try to uncover hidden requirements and functionalities that are not clear from the design.
- Verify if there are pre-existing components that you can use.
- Communicate with a designer on what to prioritize and what can be skipped.
- A design usually encompasses a best-case scenario. Be sure to ask for a design with real-world data.
- Find hidden requirements and edge cases by trying to break the designs.
- Always be kind and gentle when critiquing designs.

Backend

- Align on user sessions and user tokens
- Check if non-UI features impact your feature. Such as notifications, that is part of the UX, but not UI, and is aligned with backend engineers.
- If you're the first implementer of a backend API, assume that you're *also* going to be its tester. Adjust your planning for this.
- Steer towards error-codes, not string-based errors supplied by the backend.
- Align on consolidating network calls. Maybe you can make your life easier by receiving a single network call.

3 Want to read more?

Enjoyed this sample? There's a lot more to discover in Mobile System Design! The book dives deep into turning real-world challenges into robust mobile solutions, covering everything from planning and testing to UI architecture, reusable components, and scaling up an app, including modular design and design systems.

Use code [SAMPLIST](#) for an exclusive 10% discount on the full book.

What you'll learn:

- How to transform briefs into actionable development plans
- Strategies for writing testable, maintainable code with minimal effort
- Practical dependency injection without unnecessary complexity
- Decomposing designs into reusable, scalable UI components
- The art of building self-sufficient features that stay nimble
- Managing complex navigation and crafting resilient UI flows
- When and how to implement a UI library or design system
- Scaling mobile architectures while maintaining quality

Buy the book at www.mobilesystemdesign.com and level up your mobile system design skills!

