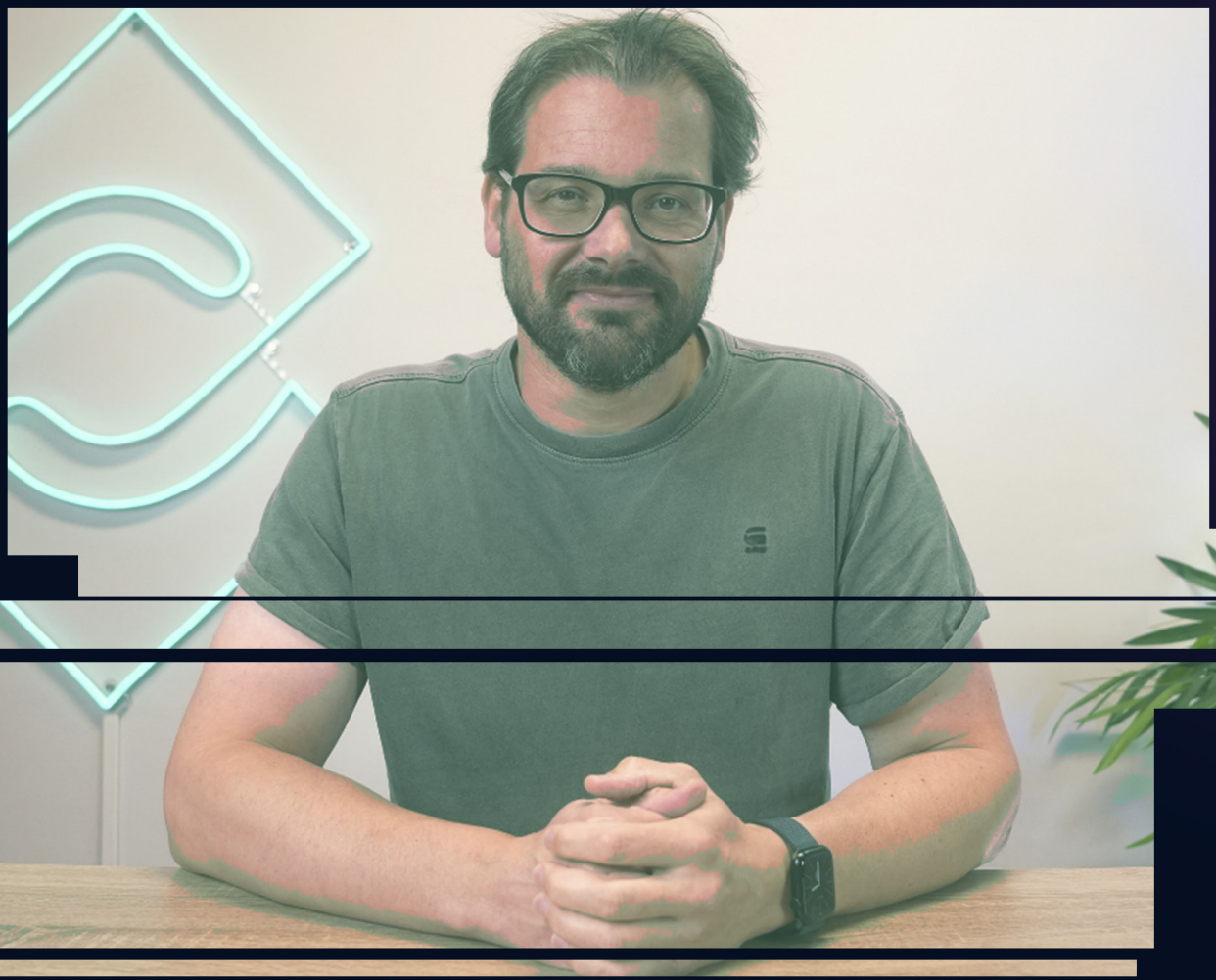
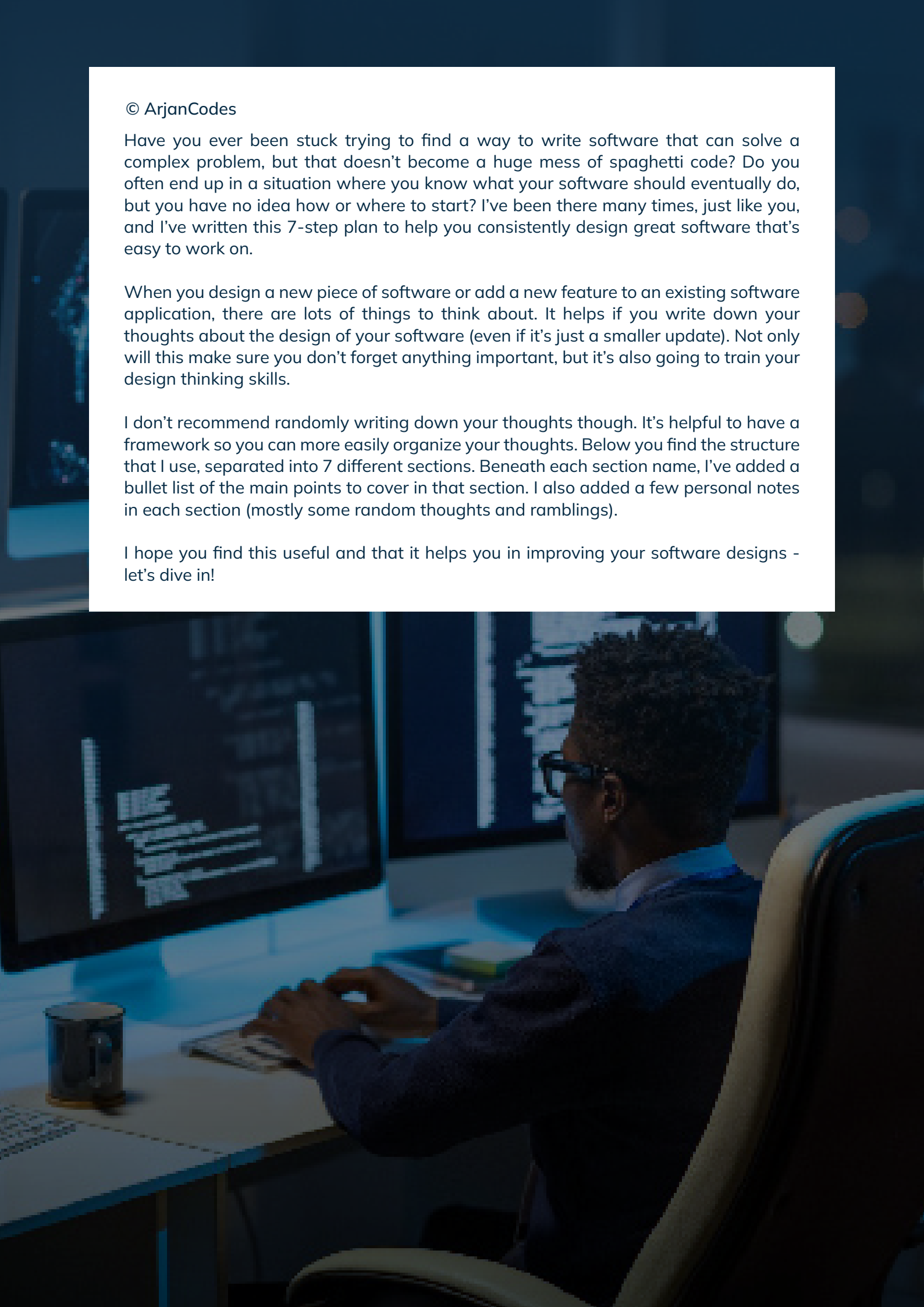
ArjanCodes

SOFTWARE DESIGN GUIDE





Have you ever been stuck trying to find a way to write software that can solve a complex problem, but that doesn't become a huge mess of spaghetti code? Do you often end up in a situation where you know what your software should eventually do, but you have no idea how or where to start? I've been there many times, just like you, and I've written this 7-step plan to help you consistently design great software that's easy to work on.

When you design a new piece of software or add a new feature to an existing software application, there are lots of things to think about. It helps if you write down your thoughts about the design of your software (even if it's just a smaller update). Not only will this make sure you don't forget anything important, but it's also going to train your design thinking skills.

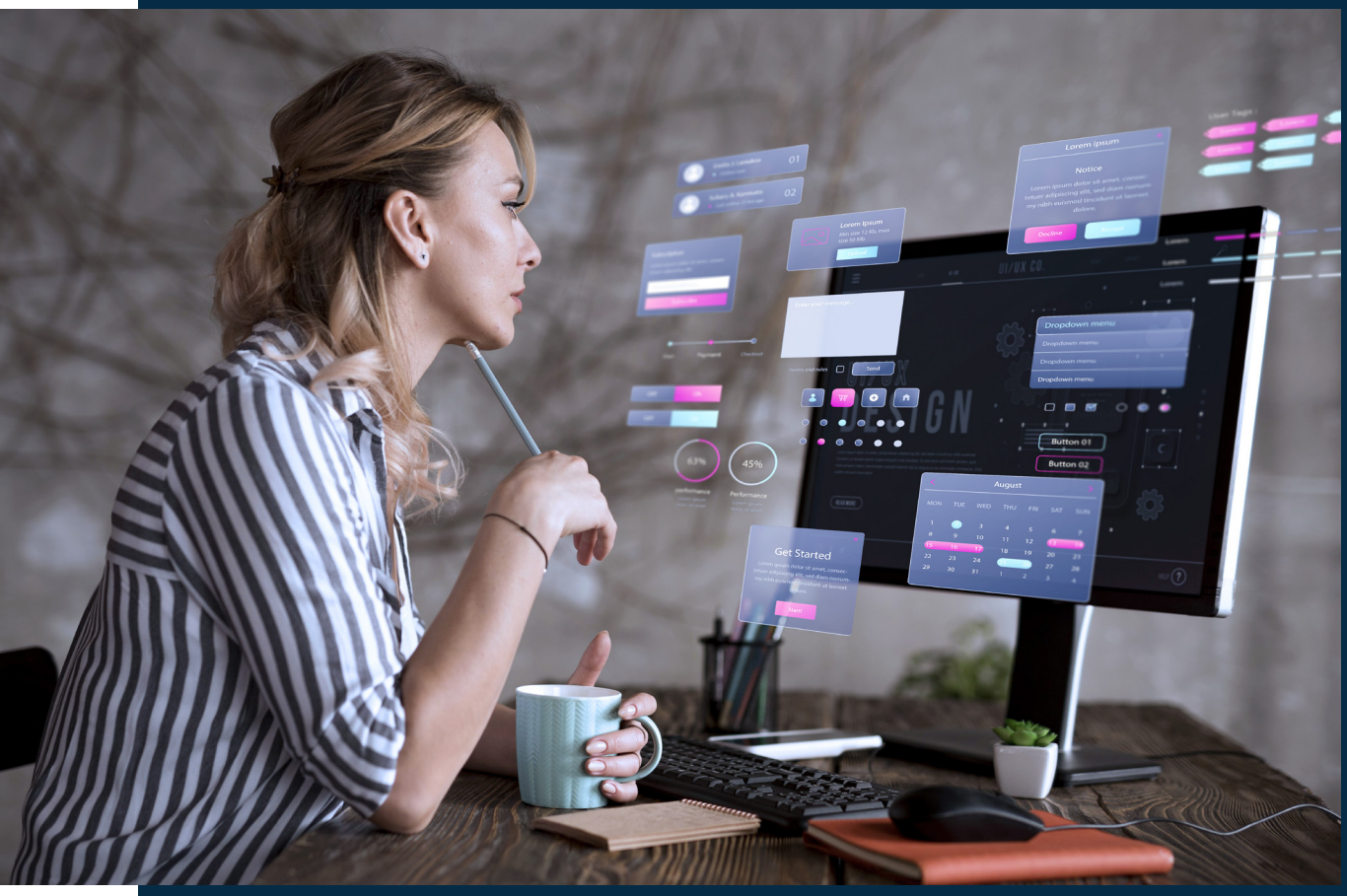
I don't recommend randomly writing down your thoughts though. It's helpful to have a framework so you can more easily organize your thoughts. Below you find the structure that I use, separated into 7 different sections. Beneath each section name, I've added a bullet list of the main points to cover in that section. I also added a few personal notes in each section (mostly some random thoughts and ramblings).

I hope you find this useful and that it helps you in improving your software designs - let's dive in!

01

DEFINE WHAT YOU'RE BUILDING

- What is the software application or feature?
- Who's it intended for?
- What problem does the software solve?
- How is it going to work?
- What are the main concepts that are involved and how are they related?



The very first step is defining what your software application or feature does. There are several standardized processes for this. Personally, I work a lot with Domain-Driven Design (by Eric Evans), even though that happened more as an emergent behavior than from me reading the book and applying the ideas.

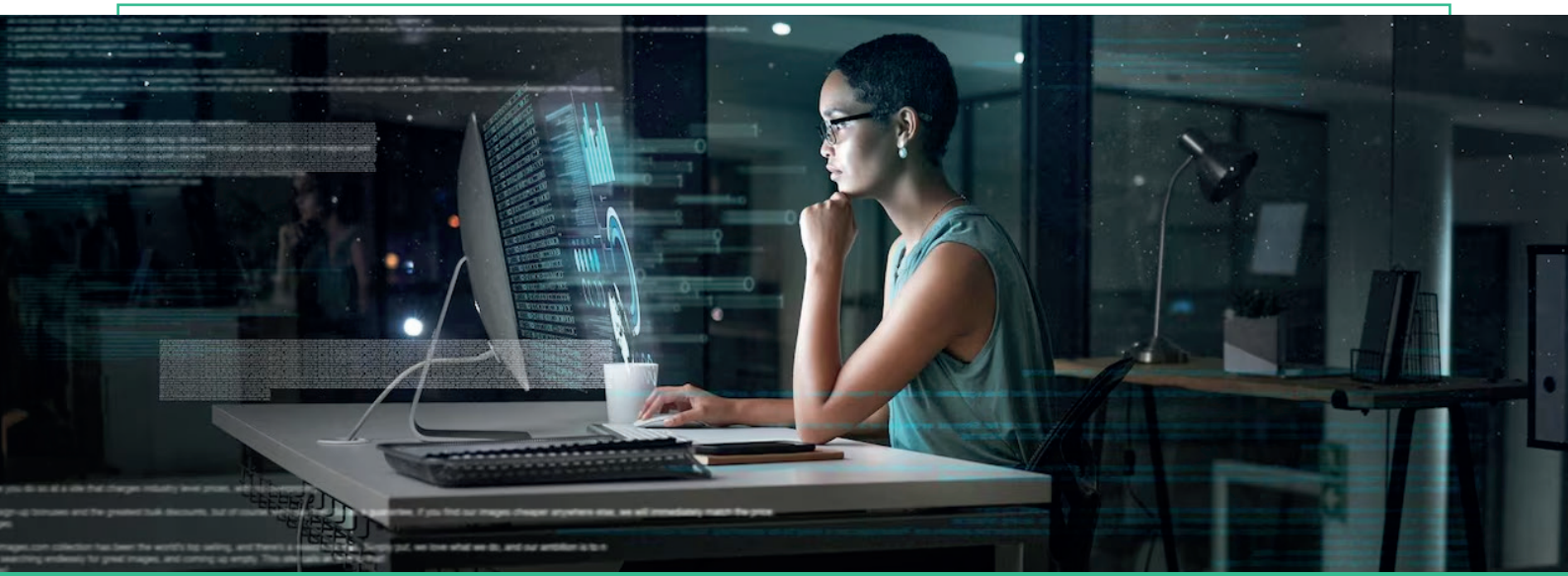
The most important thing about domain-driven design is that it focuses on getting some sort of early implementation, a crude prototype, based on a model of the domain. In practice, don't spend all your time coming up with the perfect design and then start implementing. Instead, do these things in parallel! As you come up with the design for your software by analyzing the domain, it's incredibly helpful to start working on some sort of implementation. This allows you to get a feel for the technical setup of your software. You may uncover technical limitations that affect your software design. At the same time, analyzing the domain might also lead to new insights on how the technology should work.

Don't hesitate to remove things (this is called "distilling the model"). By removing things, you simplify your design and that lets you focus on what's important: the core of what your software needs to do. Remember that the best type of refactoring is removing code.

When you define what a software application or feature does, what concepts are involved and how everything is related, I often follow a "zoom out and zoom in" approach. As a first step, I try to think as broadly as possible about the software and what it would entail ("zoom out"). The funny thing that I've noticed over the years is that as I've become a more experienced software designer and developer, I'm spending more and more time in this conceptual stage, and less time implementing it. The implementation almost becomes an afterthought: it's simply the final step of translating the concepts and relationships that I wrote down into code that a computer can understand and run. However, especially if you adopt the methodology of doing design and implementation in tandem, the boundaries between design and implementation tend to become blurry.

After zooming out, you need to zoom in again. What should we do first? Are there parts we can delay building? What's the low hanging fruit? How does the order of building things impact the complexity of the work? Are there parts we should build first so we can test everything quicker, leading to more efficient development later?

From a business perspective, this is also called defining an MVP (Minimum Viable Product). The aim is to do the minimum work necessary in order to be able to test whether the thing you're building is useful and (if applicable) can form the basis of a business.



02

DESIGN THE USER EXPERIENCE

- What are the main user stories (happy flows + alternative flows)?
- If you're adding a new feature to an existing software application, what impact does the feature have on the overall structure of the interface? (are there big changes in the organization of menus, navigation, and so on?)

You can also provide UI mockups or wireframes here as a part of the user stories to clarify what the flows are going to look like.

When I design software, I always must remind myself that I'm doing this for someone else, not for me to pat myself on the back for finding such a nice design. What you design must make sense to the user. Sometimes, that means you need to introduce an extra concept to clarify how your system works. And sometimes the most generic solution is not the best solution for the user.

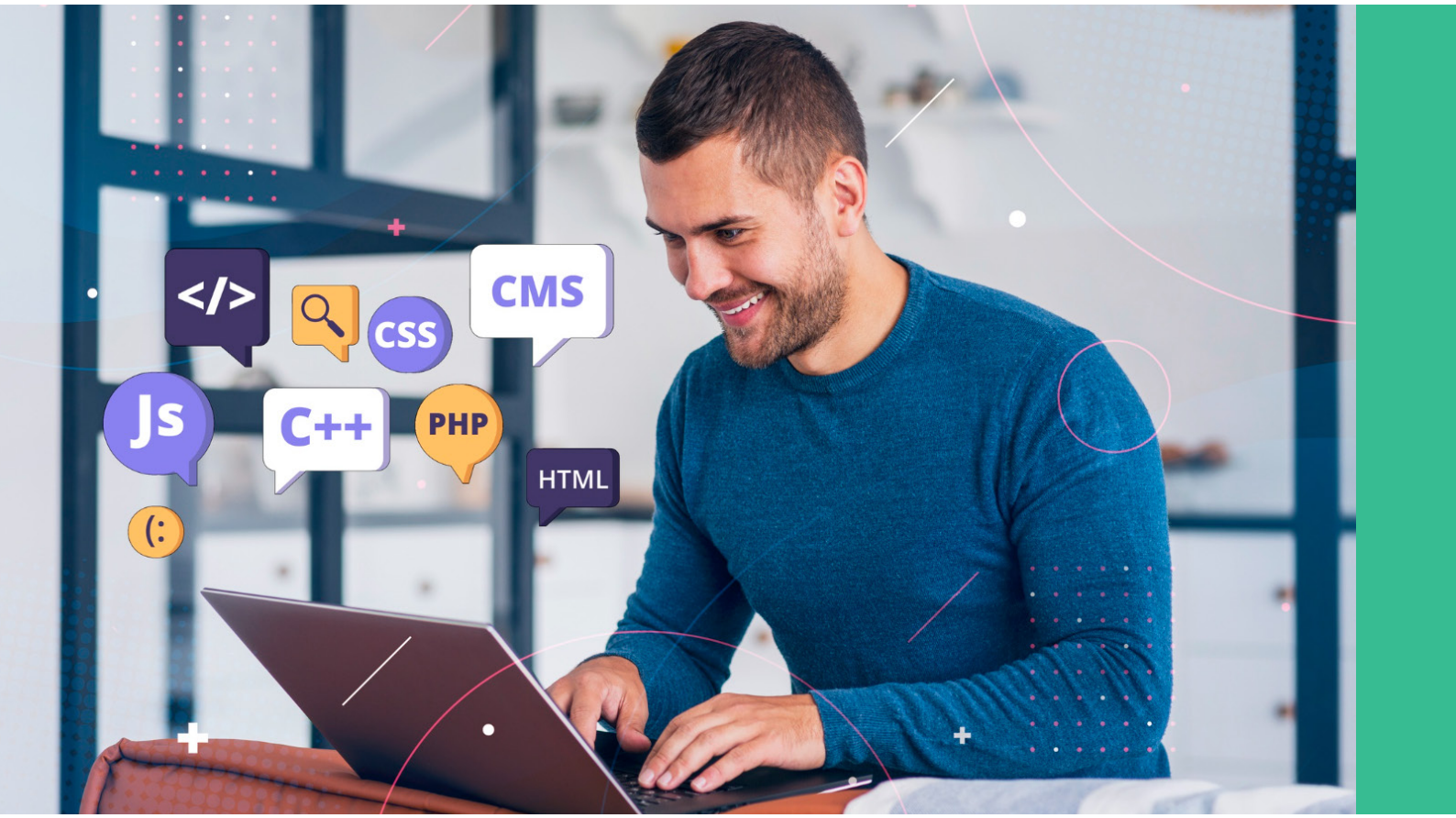
For example, in the past I worked on a software platform for schools and universities. We wanted to add an analytics dashboard to the platform. I created this very generic solution that allowed customers to create their own analytics dashboard and define custom analytics blocks where customers could write their own Python code to do the actual analytics. It was a very generic and powerful solution, but not a single customer used it, because they simply didn't need complex analytics.

My mistake was that I didn't check whether users actually needed something before I built it. Nowadays, I keep things simple and make sure to only spend time on implementing things that I know users are going to benefit from. In my experience, building user interfaces is one of the most time-consuming things of building software. Especially since every user interface change also means writing documentation, communicating about it with your existing users, and potentially introducing bugs that you need to solve in the future.

03

UNDERSTAND THE TECHNICAL NEEDS

- What technical details need developers to know to develop the software or new feature?
- Are there new tables to add to the database? What fields?
- How will the software technically work? Are there particular algorithms or libraries that are important?
- What will be the overall design? Which classes are needed? What design patterns are used to model the concepts and relationships?
- What third-party software is needed to build the software or feature?



You can use UML to draw out the main classes that will be added and how they fit into the rest of the system. I like to use Mermaid for drawing diagrams as this is text-based. Also, many tools for developers such as GitHub or VS Code have support for rendering Mermaid diagrams.

When writing code, there are a few simple things you can keep in mind that will help you write code that's easy to maintain:

a. Use functions over classes.

In my experience, functional code is typically simpler than object-oriented code. Especially in languages like Python or Rust, it's easy to write great functional code as these languages have good support for functional programming concepts. If you notice you need to pass along too many things as arguments to your functions, or if you need some sort of object representation for data, that's a great time to introduce a class.

b. Keep things small and simple.

This is true whether you're thinking about modules, functions, methods, or the number of instance variables in a class. Your code will be easier to read and test.

c. Separate creating the thing from using the thing.

Don't create an object and then immediately call methods on it. Instead, create the object outside of the function and pass it as an argument. This will make your code easier to test.

d. Use abstraction.

In Python, you can use ABCs or Protocols. In Rust, you can use Traits. Either way, these help you remove dependencies, and this reduces the chance of breaking things when you work on your code.

Just doing these things already helps tremendously with keeping your code simple and easy to work on. I often talk about concepts like these in the videos on my YouTube channel, so check those out for more details and code examples.

Finally, something that you might want to address in this step is to describe specific edge cases that you want the system to handle correctly, for example what should happen in case of a network connection error.

04

IMPLEMENT TESTING AND SECURITY MEASURES

- Are there specific coverage goals for the unit tests?
- What kinds of tests are needed (unit, regression, end-to-end, and so on)?
- (new feature only) Are there any potential side-effects on other areas of the application when adding this feature?
- What security checks need to be in place to allow the software to ship?
- (new feature only) How does the feature impact the security of the software? Is there a need for a security audit before the feature is shipped?

Everybody always talks about how important testing is, mainly because it helps you find bugs and produce better, more reliable code. That's part of it, but there's a way more important reason to make testing part of how you operate though: by designing and writing code that is easy to test, you develop a tester's mindset. A mindset that every piece of code potentially breaks something. A mindset that takes edge cases into account by default. If you adopt this mindset, you'll start to automatically organize your code such that edge cases are handled almost automatically. Design interfaces, classes, and methods in such a way that they nudge the users into following the happy flow.

For example, suppose you're writing a function that retrieves users from your database with a particular role. You could use a string argument to specify the name of the role. The problem is that you open the possibility of calling that function with roles that don't exist. And then you'd have to write extra code to handle that edge case, as well as extra tests to make sure you are handling it correctly. Another option is to use an Enum instead. Then, you're relying on the typing system to handle the edge case for you, saving you a lot of extra work. This is why so many developers love languages with strict type checking: they detect problems early on, so you don't have to deal with them later when the code is already shipped, and customers are experiencing bugs.

In general, make sure that your code is easy to work on. It's going to help if you settle on a coding standard and use supporting tools to make your life easier. Don't try to be too fancy. In case of Python, having an automatic formatter like Black or Ruff will already help just like following the best practices for coding in Python (I have many videos on my channel covering this in-depth).

Python is very loose with regards to typing and errors. When you just start coding, this is nice, because you can get started very quickly without having to think too much about edge cases. However, in the long run this does mean that as developer, you need to develop a good sense of how your scripts are going to be used. Rust on the other hand is the complete opposite: it has a sizable learning curve and when you write Rust code for the first time, you might feel like you're constantly fighting with the compiler. However, this does mean that you are forced to deal with edge cases before shipping your code, resulting in more robust software.

Write your tests with the assumption that you're going to change things during your development process: don't focus on reaching 100% test coverage no matter what. Instead, focus on getting a basic version of the feature done so you can evaluate whether this is really what you or your users want. In fact, assume that every line of code you write you're going to throw away at some point: don't get attached to the code you write.

05

PLAN THE WORK

- How much time will it cost to develop the software or feature?
- What are the steps and how much time does step take?
- What are the developmental milestones and in what order?
- Are there any migration scripts that need to be written?
- What are the main risk factors and are there any alternative routes to take if you find out something isn't feasible?
- What parts are absolutely required, and what parts can optionally be done at a later stage? (i.e. the Definition of Done)

Especially if you're a beginning developer, it's hard to estimate how much time developing a piece of software takes. You will get better at this after doing it a couple of times. What helps me is first figuring out what the main risk factors are. For example, if you want to add a feature that allows users to integrate your app with Microsoft Teams, the main risk factor is the process of building the actual integration code (which will probably reside partly in the Teams App store), and not the user interface additions in your app to set it up. The more risk factors you have in the software design, the more time you need to reserve for nasty surprises.

06

IDENTIFY RIPPLE EFFECTS

- What needs to be done outside of designing and implementing the feature?
- What documentation needs to be updated?
- Do you need to communicate something to existing users?
- Are there other external systems that need to be updated? For example, a payment provider, email marketing, sales system?

As a technical person, it's easy to fall into the trap that the only thing that needs to be done is build the actual software or feature. However, I've learned that all the processes and tasks around software development, releasing it to the customer, and making sure customers can successfully use the software, take up a lot of time. Though it might not always be important, and you may not be the person who does these tasks, I find it helpful to already think about these things when I work on software, so they don't come up at the last minute.

Especially for software that has many users, thinking about the ripple effects of changing the software is crucial. This is also why it sometimes feels like larger software companies move incredibly slow. It's because the ripple effects for large companies are huge, so they need to spend a lot of time making sure these are properly handled!

07

UNDERSTAND THE BROADER CONTEXT

- What are limitations of the current design?
- What are possible extensions to think about for the future?
- Any other considerations to take into account such as a budget?

Whenever I design software, I always like to include a few “moonshot” ideas: “It would be really cool if the software could also do X”. The reason this is useful is that this puts you in a very open mindset that sometimes leads to new insights. Also, when you’re developing the software, it’s possible that you get an epiphany that allows you to add one of these moonshot ideas. It’s awesome when that happens!

Finally, knowing the limitations of your design will help you improve the software in the long term as this will give you an immediate starting point whenever you need to make changes to your software in the future.

You may have noticed that I didn't really talk much about design patterns in this guide. This is on purpose. The most important phase of coming up with a great design is understanding the problem you're trying to solve. This is a step that a lot of developers skip, but it's a crucial part of designing great software!

Learning about and becoming comfortable with design patterns and principles is the next phase though, so I promise that I'll discuss them at length with you in the future. In the meantime, I hope this guide gives you something to think about as we begin working together to help you become a software design expert.

So, was this guide helpful to you? I'd love to hear your feedback. Feel free to send me an email!

- Arjan

WHERE TO FIND ME

Website	https://www.arjancodes.com
YouTube	https://www.youtube.com/@arjancodes
LinkedIn	https://www.linkedin.com/company/arjancodes
X	https://x.com/arjancodes
Discord	https://discord.arjan.codes