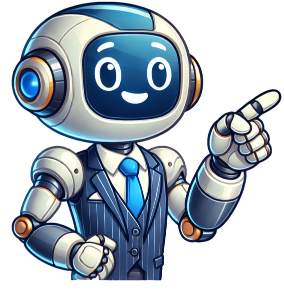
Click Here



When I search about the KISS principle on The Internet, I stumble across many websites defining it in a couple of lines: simplicity is important, lets be simple, the end. They often fail to explain what is simplicity, why simplicity is important, and how to achieve it.Simplicity is one of the driving idea we should keep in mind at all time when designing a system. The problem: its really hard to achieve.You guessed it: well dive into simplicity (and complexity) in this article. I wont write about all the different ways complexity can creep in your codebase but, instead, Ill try to give you a quick overview of the different masks complexity can wear, with many examples. Well go from the business domain itself, through the nitty-gritty (the implementation), to end up in the complexity of software architecture.More precisely, well try to answer these questions:Whats the difference between simple and easy?Why do we need to keep things simple?Can we avoid the complexity created by the business itself?Why should we thrive to delete code?What is cyclomatic complexity?What means simplicity for an architecture?How to manage your dependencies?Why we shouldnt try to outsmart everyone with our code?Simple Or Easy?There are important difference between the concepts of easy and simple we need to understand first. Lets begin with plain definitions. What the dictionary has to say about simple?Plain, basic, or uncomplicated in form, nature, or design; without much decoration or ornamentation.Composed of a single element; not compound.Easy enough: if your system has only a few parts, you have a simple system.We could also try to define simple by looking at the definition of its contrary, complex:Consisting of many different and connected parts.Oho! Thats interesting. A simple system is not a system with one and only one element necessarily. Its a system which has not too many different elements connected to each other.If youre still not convinced, thats great: its really important to have a critical mind. Lets look at a synonym of complex, complicated:Consisting of many interconnecting parts or elements; intricate.I can feel your adrenaline going through your entire body, your neurons doing the dance of understanding, and your whole soul projecting the light of knowledge in the universe. Again the same idea!Now, whats the difference with easy? Lets open the dictionary again:Achieved without great effort; presenting few difficulties.Lets say that you have to do an addition in binary, and you have no idea how to do it: its not easy for you. You lack some knowledge. But your addition is not a system with too many parts. Its not a system with connected parts. As a result, its hard for you, but its also simple.Another example: if you have a codebase with 18237 classes all coupled to each other, you have a complex system: there are too many elements (the classes) entwined with each others. Even if you can understand every single algorithm in your system (theyre easy for you), the system itself stays complex.To summarize, here are the two sins of complexity:Too many parts in the system.Too many interconnected part in the system.Now that we really know what complexity (and, by extension, simplicity) means, you might wonder:Why is it important for a software developer to put his nose in some boring dictionary?Why simplicity should be considered as a God who will save us all?Why Do We Need Simplicity?Why being a software developer can be hard? One word: change.If the application youre building will never change, stop reading this article immediately! Actually, dont read anything about good principles in software development, its useless for you. Just do some procedural code in any language you want, dont follow any principle, and youll succeed. Nobody will put an eye on your codebase anyway.A codebase which doesnt change is a myth. The context around your application will change (libraries, market) and, as a result, your application will need to change, too. To modify your application, youll have to understand how the different parts of your system work together. This representation sitting in your brain is the mental model of your system.Depending on the system, your mental model will be more or less accurate. What about complex systems? Im sure youve guessed it: its difficult and tiring to create an accurate mental model of a complex application.Controlling complexity is the essence of computer programming. We will always be limited by the sheer number of details that we can keep straight in our heads.Brian KernighanSourceLets look at the codebase of Dave, your colleague developer. Its full of hidden dependencies and global mutable state. When you change something, you have no idea of the rippling effects propagating in the whole codebase. Your energy level goes down as youre trying to put everything in your head and, eventually, your anger goes up. At that point, you might surprise yourself cursing Daves family on multiple generations.The result of complex systems? Bugs, unhappy developers, risk of burnout, and frustrated customers at the end of the chain. Nobody wants that.Business ComplexityThe complexity of a system will also be a good indicator of the amount of time developers need to add a new functionality. This is really important, business wise. Nowadays, an organization needs to have a good amount of adaptability and velocity (speed with a direction) to succeed.Yet, we cant always avoid complexity. Its first and foremost the mirror of the business complexity we build an application for. The complexity of the requirements will decide of a good part of the complexity of our systems!For example, if you develop an e-commerce, you will have difficulty to avoid having products, orders, shipments, stock management, or returns. There a lot of moving parts in there.Thats why, as a developer, you need to keep the following in mind:If you can avoid complexity, avoid it, as much as you can.If you cant, you need to make sure that you manage it properly.How the hell do we do that?Simpler RequirementsTo avoid as much business complexity as possible, I suggest attacking the complexity of the business requirements directly. How?If you dont have any planning meeting where developers and business people (project managers and whatnot) discuss the requirements, please ask for it. Insist on that point, till you get it. Dont follow blindly what the top management is throwing at you. They might not understand it, but a developer has a lot of valuable knowledge which can help to reduce the complexity of the requirements.During these meetings, when you see a potential feature which will bring a lot of complexity, ask questions which begin by why. Why do we need this feature? Why is it useful for our customer?If you really need this feature (which will be often the case), you might have a better idea to make it simpler. For example, what about having 10 fields instead of 20 for a user register form? Additionally, a simpler feature to implement will often be simpler for the customer too. This is a good argument to sell your idea to your stakeholders.That said, be aware that reducing complexity directly in the requirements is difficult. Why?First, many stakeholders seem married to their ideas. Thats normal: when we have ideas, we often think its THE idea which will change the world. We often have close relationships with them, associating our ideas with who we are. This is a mistake: these ideas are external entities. They are not an extension of our personalities.Additionally, the more we work on our ideas, the more we will feel the need to get something out of it. Even if the idea is a very bad one, we can stay stuck on it for a long time.If you see somebody too attached to their ideas, knowing how to argue properly can help here.Second, its sometimes not possible to simplify a feature. In that case, try to reduce the complexity has much as you can while implementing it. More on that later.Keep in mind though: if you try to understand the business domain youre working for, you will find simpler solutions. Try to speak with domain experts (people having the knowledge of the business itself) as much as you can, always refresh your knowledge, and avoid being overconfident in what you know.The customers, at the end of the chain, should be the ones discarding ideas and features. Listening to them is utterly important.A Good Implementation is a Deleted OneLess code will often make your system simpler. Big is often synonym with complexity.Indeed, changing your code can have an impact on the surrounding code, and create undesired results. When you think about it, lines of code are potential technical debts, sources of bugs, triggers for headaches, screams, and tears. Show no mercy: remove every useless piece of code you can find.As Antoine de Saint-Exupry famously said:Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.Enough said.Dead CodeDo you want a cemetery full of old functionalities rotting in your shiny system? Me neither.But first, whats dead code? Everything which is not used in the application at runtime: variables initialized but never called, method never used, classes never instantiated.Letting your dead code around will have the following consequences:Developers will be confused while working on the system. Questions and doubts will cripple their mind: whats the purpose of the code? Should I keep it? Is it useful?Your colleagues will possibly maintain it, spending time refactoring it for no benefits.Dead code hide the real implementation. Did you ask yourself already: why, when I modify this method, nothing happens? I did. It gets old, after a while.Simplifying our codebases by removing dead code will simplify our mental models. It means fewer headaches and less energy drained into these useless zombies.The YAGNI PrincipleA specific form of complexity (or dead code), which is often ferociously defended by developers, is the famous code which will be useful one day, in the future, so we should keep it.This is where the YAGNI principle comes from: it means You Arent Gonna Need It. And its right, 99% of the time: you never know what will happen in the future, and the code trying to do some divination will never be used and forgotten.If you stumble across some code written for no actual present purposes, delete it. If you need to implement it in the future, it will be in a different context, and your old code would have become obsolete anyway. You can still come back to your first implementation youve deleted with your version control system (git, anyone?).I dont count how many debates I had about this specific topic during my career. Heres an example:Me: Why the functionality to put a hat on users is implemented in the system? I dont see the option available on the frontend but its everywhere in the backend!Colleague: Its not used for now. But we will use it one day, for sure.Me: But it was implemented in 2016, we are in 2019. If nobody needed it for three years, maybe we wont need it for two years more. We could delete it.Him: No. Its too complicated. We will use it, for sure.Me: but developers will maintain that. I spent hours myself to understand how to manage it, since Im implementing something directly related which could break it.Colleague: We will need that for sure.Me: and what about this functionality which qualify the user as wizard? Colleague: Oh. Its not used. But it will be used one day, for sure.Me: Nobody knows the future. Functionalities can be canceled or staying in the backlog forever. Context change. Meanwhile, useless features will be maintained, sometimes with great efforts, for nothing.Yet, I saw developers spending a crazy amount of time to be as flexible as possible for the hypothetical next features which never come.Only add the complexity you need to answer the present business needs, no more.Were developers, not soothsayers.Global States and BehaviorsUsing global states is not a good idea 99% of the time. I wrote a detailed article about it here, also explaining the dangers of states with growing scopes.Simple ArchitectureComplexity kills. It sucks the life out of developers, it makes products difficult to plan, build and test, it introduces security challenges, and it causes end-user and administrator frustration.Ray OzzieLasagna ArchitectureLasagna is a delicious dish (when its well-made), but a nightmare in a codebase. The problem of a lasagna architecture? Too many layers of indirection.Lets imagine that you have an application which simply takes some inputs from APIs, apply some business logic to transform the data and store it somewhere. Sounds reasonable, does it?Heres what you can do. Each point is a layer:APIs receiving some request.Wrappers of the APIs.Server factories which create the different controller and inject the APIs wrapper.A pool of server factories in case you want to have multiple of them (?).Controllers where the API wrappers are called via interface constructs.A validation layer to validate the inputs.Another layer to share code between every controller (in an attempt to make everything as DRY as possible).A model layer which is as dumb as my feet (anemic models).A persistent layer called via interface constructs from the controllers.Another layer which only get the Ids of entities to fetch the whole entity itself.A layer which does the actual CRUD and implement the real business logic. In case of selects, this layer does some SQL request, get only the IDs and pass it to the layer 10 which will fetch the entity.You didnt get the whole picture? Thats normal.Now, imagine yourself trying to modify some functionality. You might have to:Go through most of the layers.Try to find out if one layer will crash because of your change.Try to remember how everything works in your painful brain.Make your change.Wonder what the layer was doing.Verify everything again.Fix three tests failing on 4 layers. Two days of work only for that.Call your boyfriend, your girlfriend, or your favorite cat, screaming that youre such a fraud.Dream about a little house in the mountain where you can spend your time making some goat cheese instead of wasting it with some nonsense.Youll need to have a mental model spreading on 11 layers to correctly manage the changes of your codebase. Good luck!Now, you might think that Im crazy. You might think that this kind of architecture was created by my sick brain. You might think that such codebases dont exist. But this application is real: Ive worked on it enough to still feel the pain when I think about it.You know what? This codebase was full of good intentions. The developers responsible for this monster didnt want to create a complex system on purpose. They wanted to help everybody by creating something extremely flexible, elegant, DRY, the perfect software.As my good old friend Saint Bernard of Clairvaux was saying the other day:The road to hell is paved with good intentions.The previous example will be useful to explain why unnecessary abstractions can make your system more complex.Abstractions and ComplexityWhats an abstraction? Its a way to deal with complexity by hiding some useless details. I wrote about it in depth in another article.For example, a function is an abstraction: when you call a function in your code, you wont necessarily be aware of its implementation, and, as a result, its complexity. Instead, youll:Look at the signature of the function (its name, input, and output).Decide to use it or to find something better.It sounds great, right? It doesnt matter how much complexity a system has anymore! Lets develop a big Ball of Mud and put layers of beautiful, flowery abstraction on top, to hide all the mess.Well, not really.As we saw, functions can be considered as an abstraction. Objects in OOP too. Even some primitive of your favorite programming language, like arrays for example, can be seen as an abstraction of a more general concept (a mathematical sequence, for example).For your abstractions to work, you need to create cohesive implementations where things which belong together are together. Otherwise, the complexity will just jump back on your face like a mental health leech. Consider this example: