# SOFTWARE AS AN ENGINEERING DISCIPLINE

Lara Haya Santiago

Alba González Arango

Daniel Fernández Cabrero

Umut Dolangac

# Content

# Introduction

Software engineering is a big deal these days, but it's not just about writing code—it's about building systems that last and work well over time. In this paper, we're diving into what makes software engineering different from basic development, why it's important to take a more disciplined approach, and how engineers can balance business needs with technical complexity. We'll explore everything from the value of planning and managing risk to keeping things flexible (without the chaos). Plus, we'll talk about the role of quality, leadership, and ongoing learning in building strong engineering teams. All the insights in this paper come from our deep dive into the podcast "SE Radio 574: Chad Michel on Software as an Engineering Discipline".

# Introduction & Foundations of Software Engineering

## Chad Michel's background

I'll start by briefly introducing Chad Michel, the guest of the podcast and an important figure in software engineering.

Michel graduated with a degree in Computer Engineering in 2000 and later earned a master's in computer science from the University of Nebraska-Lincoln in 2003.

He is currently the Head of Engineering at Don't Panic Labs and co-author of the book Lean Software Systems Engineering for Developers.

His experience gives us valuable insights into how engineering principles can be applied to software development, which leads us to the next topic: the difference between software development and software engineering.

## Software development vs. software engineering

Although these two concepts sound similar, the key difference is the level of rigor applied.

Software development is easy to get into. With tools like VS Code and JavaScript, anyone can quickly create a functional webpage. However, this often comes at the cost of quality and requirement satisfaction.

On the other hand, software engineering is not just about writing code—it's about applying structured methods to ensure the software is scalable, maintainable, and meets customer needs.

A simple way to put it is:
A developer might make something work fast, but an engineer ensures it's also secure, efficient, and easy to maintain over time.

In short, rigor is the key difference. Without proper planning, software can become fragile and hard to maintain.

### Importance of rigor

This brings us to the difference between scientific rigor and engineering rigor.

In science, the goal is to learn and discover how things work. Scientists create experiments to test hypotheses, even if the outcome is uncertain.

In engineering, the goal isn't just to learn—it's to apply what we already know to build reliable solutions.

As the saying goes:
"In science, we build to learn something. In engineering, we learn how to build things."

This mindset is crucial when creating software that isn't just functional today but sustainable in the long run.

Without proper planning, software may seem to work at first, but over time, it becomes difficult to maintain and modify.

Many developers jump straight into coding without doing prior research or documentation. This often leads to fragile, costly, and unreliable systems.

If we think about construction, we could rush to build a house without blueprints, but would it be safe? Probably not. The same applies to software.

That's why, in software engineering, we need to consider maintainability, cost, and risk from the beginning.

To sum up, good software isn't just about making it work—it's about making it last.

## Business & Complexity in Software Engineering

### Business language in engineering

Most of the software that we're building is for some business purpose. Due to this, it is very important to speak the language of business, which is schedule, cost and risk.

- Schedule: we must deliver the project on time, if we can't, we may need to make different choices.
- Cost: if the cost is much more elevated than what we thought it was going to be, we have also a problem and often we don't realize until it is too late.
- Risk: it is very important to provide certainty to the business. Not being able to provide any level of certainty is a huge risk to the business and can lead to poor financial decisions.

## Planning & trade-offs

It is necessary to make trade-offs in business, no one has unlimited budgets or unlimited time.

It is essential to have a plan, as it actually gives you a target to hit and you have clear what you have to do. Some useful quotes on planning:

- “Plans are imperfect but a good enough plan can be better than no plan at all.” – The plan doesn’t have to be perfect
- "If you fail to plan, you plan to fail."
- "Plans fail, use plans."
- "Plans are worthless; planning is essential." – Eisenhower

Inside your plan you need to specify the estimates. A software estimate predicts how much effort is necessary to create, modify, or maintain software. You give estimates in some unit, usually hours and points.

## Complexity in software engineering

There are three kinds of complexity:

- Objective complexity: helps to keep present the project's true objectives
    - The goals for the customer
    - The measurables that would actually show that we achieved the goals
    - The potential challenges you may run into
- Requirements complexity

It is very important to manage requirements complexity. Requirements very often are duplicated or inconsistent, or even contradictory. Agile tools have improved this area, but more improvement is needed.

- Solution complexity

You cannot design a solution in isolation without being aware of the implications of the delivery. There’s no control on the environment in which the solution will be delivered and that may be a source of additional complexity. To prevent this, you can identify the solution delivery problems that may arise due to this complexity. The goal is to avoid unexpected issues.

## Managing agility & control

Agility comes from managing both requirements and solution complexity, we need to have some control over them. Without management, projects lack control and direction.

Being agile means that we can adapt to changing business needs. We have to assume that our design is going to change and think how are we going to handle these changes. A new requirement shouldn't require big changes to the system and should not ruin the project.

The potential changes can be difficult to identify. In physical systems, for example, it is easier to understand this because you can see and feel them. Seeing software is more complicated, that's why we never get the design right the first time.

We cannot avoid making changes on our system because design is an iterative process. However, we must ensure that it can adapt as needed without complete redesign.

Another good practice is to present your design to someone else so they can identify the weak points and help you get another perspective on your project.

If we can achieve those goals we should be able to continue to provide that agility for our customers, making sure we can keep maintaining and growing this system for a long time.

## Software Design, Quality, & Engineering Constraints

### The importance of early issue detection

When we think about bugs, we usually focus on the ones buried in code. But as Chat Michael pointed out, *"you can have bugs and defects in your requirements and in the design".* We often get so caught up in debugging and fixing code that we forget problems can start much earlier, even before a single line of code is written.

Bugs in production are costly and stressful. Given the *hyper fixation* people tend to have on bugs that come from code, we overlook the fact that around 50% of product bugs come from bad requirements. If those initial mistakes aren't caught early, they snowball into major issues that become much harder to fix, hence more expensive. Constant changes without a solid foundation can lead to structural problems, creating unnecessary headaches for both developers and users.

When a bug is found later in production, we not only have to worry about how much it will cost to fix the product, but also about the increased stress it brings to the team. Sometimes, developers spend hours fixing something only to later find out that the real problem was a misunderstanding from unclear requirements. Worse yet, skipping proper testing can create new issues while trying to fix the old ones. The earlier a problem is found, the easier and cheaper it is to fix, making strong planning and clear requirements just as important as writing good code.

### Refactoring vs. rework

Refactoring and rework may seem similar, but they serve different purposes. Refactoring involves improving code while keeping existing tests intact, whereas rework means fixing deeper structural issues.

While some rework is inevitable, it should be minimized to avoid wasting time on things already built. In software, the more disorganized a system becomes, the harder refactoring gets. Large-scale refactors for new requirements should be avoided whenever possible.

For example, refactoring would be optimizing an algorithm to improve performance without changing its expected outputs or altering existing tests. On the other hand, **rework** would be

realizing that the entire database structure doesn't support a key feature, forcing a major redesign and migration.

### Quality Assurance vs. Quality Control

Quality isn't just about testing at the end—it needs to be built into every step of development. Quality Assurance (QA) is proactive, ensuring good practices from the start, while Quality Control (QC) is reactive, catching issues after they happen.

As Chad Michel put it, *"We have to have engineers writing good code, we have to have designers designing good systems."* If any part of the process fails, it creates bigger problems down the line. That's why quality should be institutionalized, not just left to testers.

Inspired by Toyota's *Andon Cord* concept, engineers should feel empowered to stop development when quality concerns arise, preventing bigger issues later. The *Andon Cord* is a concept from Toyota's production system that empowers employees to stop the production line if they notice a quality issue. It's a way to immediately address problems before they get worse. When pulled, it signals the need for assistance or correction, ensuring that issues are resolved on the spot. The goal is to prevent defects from being passed down the line and to maintain high standards. This concept encourages responsibility and collaboration across all levels of the team.

### Institutionalizing Quality

Quality isn't just the responsibility of testers; it needs to be part of the entire team's mindset. As Michael mentions, we should build a "culture of quality" in means of ensuring that every role, from developers to designers embraces quality from the start. Avoiding last-minute fixes is key; it's much better to address issues early, Project managers, designers and testers all play a role, from developers to designers, embraces quality from the start.

Project managers, designers, and testers all play a role in maintaining quality, and the chief engineer's job is to guide and educate the team. As Michel points out, "education is a large part of that role." In today's fast-paced development, where features can evolve quickly—like a chat feature becoming a side part of a larger app—it's crucial to ensure that quality is integrated at every stage.

## Leadership, Training, & Challenges In Software Engineering

### The İmportance Of Leadership

Leadership is the ability to inspire, guide, and motivate others toward a common goal. True leaders do not just give orders to others; they walk the same path with the team. In the software world, chief engineers are leaders. They understand the requirements, think about the future, and see the whole project as a unified entity. They help new members adapt and educate them throughout the process.

## How to Train ourselves

Training is the process of developing long-term skills, knowledge, and abilities to improve performance in a specific field. It is essential for success. In software world, technology changes rapidly, so in order to stay updated, we need to self-educate ourselves all the time. While training, we should not just learn the tools that are popular, but we should understand the fundamentals of the system, because tools might change, but fundamentals are not likely to change in a long period of time. Also, we can go to workshops or learn from our surrounding to grasp a new concept. Finally, Effective training combines theory and practice, so we need to repeat practices to mastering at any skill.

## Challenges In Software Engineering

Tools are helpful, but using them does not make us software engineers, because Without knowing how to design a system, we cannot solve unsolved problems or update old technologies to the current standards. For example, you start in a new startup, and they are trying to do something that hasn't been done before. How are you creating an algorithm if you do not know the algorithm concepts? Thats why it is important to learn theory of subjects.

# Bibliography


- https://dev.to/victorandcode/understanding-software-estimates-and-how-to-get-better-at-making-them-3poe
- https://www.researchgate.net/publication/333295244_Solution_Architecture_and_Solution_Complexity