Table of Contents

Introduction

Why this guide?

- the clearer we make where we want to be the easier it will be to get there
- clear expectations
  - helps with performance evaluation and growth paths
- we can share this with anyone interested in working with us - do our values match?
  - we have a culture fit, but not an engineering culture fit

Culture can not be created

We cannot simply "create" the engineering culture that we want[1]. A culture can be fostered, but it is like its own living being that will constantly change and evolve. We can only guide its evolvement into a desired direction.

> [1]. https://dreadfullyposh.com/writing/sorry-but-you-cant-just-create-a-company-culture ↩

Core Principles

Key statements that summarise how we want software engineering to work at celebrate.

We are truly agile

Despite its age, the Agile Manifesto still holds valuable principles that too often are being ignored when companies talk about "being agile". Applying Scrum or similar frameworks will not make us agile if we do not understand the key concepts[8].

We believe in empowered teams

To take pride and joy in the things we finish, we need to be able to shape our work and have full transparency on its effects. We believe that Empowered Teams[7] are the way to achieve exactly that.

We cut scope, not quality

Our work happens in a triangle of quality, time and scope. We deliberately limit time and any cutback on quality will make our lives harder in the future. This leaves us with one variable: scope. We should always strive to find the smallest possible scope to meet the needs of our customers and ourselves. We do this constantly before and also while we work on something [9].

We follow industry standards, as long as we have not discovered something that works better for us

We base our tech stack and our processes on proven standards first. Only when we understand them well enough (which usually means applying and sticking to them for some time), we should look for improvements or alternatives.

We stick to our rules until we decide to change them

Principles and rules we come up with must be meaningful and justified. Breaking them is serious and can have consequences, however challenging the rules is encouraged if done for the right reasons.

We keep in touch with experts and other companies to foster knowledge exchange and keep ourselves up to date

We take responsibility for our workload.

If we see an opportunity to reduce maintenance work, we take it. We avoid unnecessary work to focus on the things we do best.

- example: SaaS > PaaS > self-hosted

Teams work on topics sequentially

A team can only truly work together if they work on the same things. We try to work on as little things in parallel as possible. The value of finishing a single thing is always greater than the value of having multiple things in progress at once (because the value of that is 0).

We work collaboratively

Writing code should be a collaborative process, pair programming, mob programming. We reduce the need for asynchronous Pull Requests and instead have the quickest possible feedback loop through writing code together.

We balance data and intuition

We work data-driven, but we also trust our intuition which is trained by practical experience and theoretical knowledge. Not everything we do can be based on numbers.

Quality Assurance is an integral part of our work

We do not hand over seemingly finished features to QA for testing. Instead, our QA Specialists work with us and help us to apply the right testing strategy before, during and after writing code.

> 7. https://www.svpg.com/books/empowered-ordinary-people-extraordinary-products/ ↩
>
> 8. https://holub.com/heuristics/ ↩

> [9]. https://basecamp.com/shapeup/1.2-chapter-03#fixed-time-variable-scope ↩

Technical Foundation

There are some fundamental practices and techniques that every software engineer at celebrate should have enough experience with to know when to apply them to the problem they are solving. This does not mean we only hire engineers who tick all the boxes on our checklist. It does mean that every software engineer must have the opportunity to learn and apply them at celebrate.

Clean Architecture

There are endless ways to structure code in an application and there is no "golden" architecture that we can apply to each and every software we write. However, most applications can benefit from applying "Clean Architecture", a distillate created by Robert C. Martin from various practices he has experienced (or developed himself) throughout the past decades[5]. Clean Architecture helps us to group our code into layers and adhere to the "D" in SOLID: Dependency Inversion [6], which is one key element for keeping our code componentized should we ever see the need to move something out into its own service. The more we build our applications this way, the easier it will also be to find our way around projects we haven't worked with before.

Trunk-Based Development

This technique encourages (or rather enforces) making many small (we'll call them "atomic" later) commits towards the main branch of a repository instead of working in long-living (i.e. more than a day) feature branches. Since many features can potentially take a larger number of those commits until they are ready for the customer, we need to work with feature branches (also covered later). It also requires

us to have a fast and robust integration process, as otherwise we would risk having (and eventually shipping) broken code in the main branch. On the plus side we avoid "merge hell" when trying to integrate large feature branches.

Test-Driven Development (TDD)

Test-driven development has been practiced for way longer than you may think. The code for the Apollo 11 mission was developed in this way, so it's been around since the 60s (find source). TDD forces us to think about what we want to do before writing code - that sounds simple, but it actually is a very brutal change for many software developers (I intentionally did not write "engineer" here), as many will rather write code as part of their though process. This can lead to suboptimal solutions. TDD has very simple rules to follow[2]:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

And as with every rule, there can be exceptions. Event Kent Beck himself does not follow those rules every single time [3]. The key benefit of TDD is putting the thought process before writing actual code and that along holds a lot of value.

This approach plays along nicely with atomic commits.

Many Much More Smaller Steps

The key to a consistent performance and a high degree of agility is breaking work down into the smallest meaningful (not necessarily valuable!) steps. The smaller the steps, the easier it will be to take them and the better we can react to changes. A step can be translated to a code change that meets all our standards, can go live and does not make the application worse[4].

Extreme Programming

Extreme Programming combines many of the principles mentioned in this section into a process. It showcases how close collaboration, test-driven development and constant exchange with customers lead to better (software) products. It may seem dated, but its fundamentals are still very valid [12].

> 2. https://localheinz.com/articles/2018/02/01/test-coverage-is-a-meaningless-metric/ ↩
>
> 3. https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565 ↩
>
> 4. https://www.geepawhill.org/2021/09/29/many-more-much-smaller-steps-first-sketch/ ↩
>
> 5. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html ↩
>
> 6. https://en.wikipedia.org/wiki/Dependency_inversion_principle ↩
>
> 12. http://www.extremeprogramming.org/ ↩

Roles

For an engineering culture like the one described in this guide to work, a couple of roles are necessary.

Product Manager

- Turns a problem into a solution together with the team
- Knows the "why" and develops the "what", leaving the "how" to the team

Software Engineer

- technical expert
  - usually specialised, e.g. "Backend", "Frontend"
- understands the "why" and the "what" to develop the "how"
  - negotiates scope
- not isolated from customers and stakeholders

Engineering Manager

Taken from http://www.engineeringladders.com[^10]:

> With the relationships that have been built and nurtured within the team, the EM provides great value to the Leadership Team. They implicitly understand the pain points being experienced by the team and should be regularly surfacing and communicating these issues.
>
> Engineering Managers are constantly in the position to identify high performers that need to be nurtured and lower performers that need assistance. Recruitment and capacity planning should start from this position. Not only when looking at bringing new talent into the team but more importantly, recognizing the unique career objectives along with how they can be accommodated and encouraged for each member of the team.
>
> Most importantly of all, the team needs to feel fully and unconditionally supported by the Engineering Manager. They are their safety layer. They provide guidance when there is uncertainty, and protection when there is a threat. This may sound dramatic, but changing scopes and shifting deadlines are a genuine threat to morale and therefore the productiveness of the team. The Engineering Manager is the provider of context when there are changes and the voice of reason when there is debate.

Tech Lead

Taken from http://www.engineeringladders.com[^11]:

> Role also known as dev lead, is the owner of the system and requires a unique balance between hands-on development, architecture knowledge and production support.

Stakeholder

- Are able to represent the customer, so they can tell us if a solution is valuable for the customer
- Are sometimes also the customer as they also raise problems they need a solution for

Customer

- We talk to customers to understand their problems, which we derive the "why" from

Lead

> [10]. http://www.engineeringladders.com/EngineeringManager.html ↩
>
> [11]. http://www.engineeringladders.com/TechLead.html ↩

Metrics

It is the job of the leadership team to build high-performing teams. In order to keep track of how good of a job the leads are doing, we need reasonable metrics. The purpose of those metrics must never be to track individuals but only to track teams.

Possible Metrics

- number of defects in relation to X - tells us something about the quality of a team's work, which is important to keep them performing

Further Reading

The books and articles listed here greatly influenced this guide. While not mandatory, reading them will help you a great deal understanding the reasoning behind many of its sections. Most of them are referenced in the footnotes.

- Kent Beck - Extreme Programming Explained
- Ryan Singer - Shape Up
- Marty Cagan - Empowered
- Daniel H. Pink - Drive
- Richard Sheridan - Joy, Inc.
- http://www.engineeringladders.com/
- Robert C. Martin - Clean Architecture