Engineering Culture Manifesto

Dalmo

Jan 2019

Culture is not organic, it is cultivated. It ought to be built on top of a foundation, only then be open to diverse points of view and suggestions. These are the fundamental pillars of engineering culture.

1. Systems thinker: The the ecosystem does not behave as isolated components. Changing something has cascading effects. One must be cognizant of how will the system react to an addition, subtraction, or change.

   Each layer of the system should have established software interfaces to communicate of other parts of the system. Those software interfaces are contracts. Period. Full stop. End of the story. Don't even try to argue or say *"but..."*. The contract is either with an external client, or an internal one. If you change something, you will be imposing unscheduled work on some other team. These are man-hours that could have been deployed elsewhere. *"It is a simple change"* is **NOT** an acceptable excuse. If you are still trying to come up with an exception, please stop.

   Do what is right. In the rare cases where a change must be implemented, mark the current interface as deprecated [1] and add an expiration notice (if applicable). Then, increment the major version for that layer, and implement a new version with the necessary changes

2. Building: What does it mean to complete a given feature or product? A clear definition of expectations is necessary, otherwise it becomes impossible to follow up progress or plan for long, or even the short term.

   A clear definition of expectations allows for outside stakeholders to gauge a realistic picture of progress, without incurring on the unnecessary stress of constant inquiring of when *things-will-get-done*?

   This is not a shield against communication and intra-team interaction. On the contrary, it is a layer of dynamic transparency that facilitates the conversation. Rather than being confrontational, conversations become more collaborative.

   The following list constitute the fundamental expectations of building:

   (a) **Architectural design**[2]: Think before acting. Before rolling up your sleeves and start coding, draw activity diagrams, use case diagrams, class diagrams, sketches, anything that will allow you to visualize all the moving parts and how they interact with the ecosystem.

   Talk your design over with another (preferably senior) engineer. They may have a different perspective on something, and the fresh eyes will help to strengthen your design.

   This also is of great importance as the foundation to the documentation. You won't remember all the details in your head. And even if you did, when another person needs to work on the product, they will be able to become acquainted with the system, and productive, without requiring your assistance

[1] Deprecated means the version still works and is fully operational. However, it will not be maintained going forward, until its expiration date (save the case of a severe bug). New software should not be built using this interface, and existing software should plan to migrate to the new version at their earliest convenience.

[2] Before being accepted, the design must be peer reviewed by a senior engineer.

(b) **Documentation**: Express and explain what the product, feature, or subsystem does. Sometimes documentation comes prior to implementation, some other times implementation comes first. However, irrespective of which one comes first, documentation is an essential part of the process.

Furthermore, documentation is not meant to be a *fire-and-forget* solution: it should be a living piece of work that evolves along with what it describes.

Attributes of good documentation[3]:

i. *Requirements*: what is needed in order for the system to work?
ii. *Setup of the environment*: how to configure the dials and knobs?
iii. *Diagrams*: a picture is worth a thousand words[1]
iv. *Interaction*: how does it interact with other subsystems and clients?
v. *How-to*: steps to effectively integrate and use it

(c) **Implementation**: Craftsmanship is the name of the game. Is it built to just work? Or was it built properly? Are there design patterns[2] applied to it? Which algorithms[3] have been used? What data structures[6] is it using? Is the code decoupled? Can it be reused in other parts of the system?

Avoid creating tech debt at almost **all** costs. If you have to create tech debt, it must be discussed with and approved by your manager.

In case you're about to try to point an exception and come up with the question: *"What if you are the CEO?"* The CEO has a countless number of managers: customers (all of them), board of directors, and shareholders, just to name a few. More importantly, customers "discuss" or "approve" with their wallets.

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."[4]

Priorities to consider when implementing (in this specific order of importance):

i. *Security*: the data should be safe
ii. *Stability*: things should run when expected, and no one should need to wake up in the middle of the night to patch something
iii. *Scalability*: does it work for 1 customer? How about 1,000, 1,000,000?
iv. *Architecture and design*: if done properly, tech debt and refactoring should be rare and far in between
v. *Maintainability*: implementation should be modular, clear, decoupled, robust, tolerant in what it receives as parameters, strict in what it sends as response, and extensible[4]
vi. *Runtime speed of the program*: yes, things have to run fast, but only after all the prior items have been satisfied

(d) **Quality control**: Trust, but verify. Test everything. Furthermore, the tests should go beyond verifying the functionality. Implement tests with the intention of breaking things. Test for unlikely scenarios, edge cases, wrong data types, and whatever else comes to mind. Last, but not least, if a bug is reported, immediately write a test case to reproduce it and ensure it won't happen again in the future

(e) **Release process**: Have a protocol for the release process, including a checklist with the key steps

(f) **Monitoring of execution**: Measure (almost) everything. Usability analytics contains valuable information about how the system is performing, being used, and whether it is producing the intended results

3. Mentoring: Don't keep to yourself, share the knowledge. However, there are times and places for that. You must not constantly interrupt your day to mentor people. You still have a job to do.

   Initiatives such as a study group, meet ups and such, are of great benefit to you and your peers

[3]The more of these items, the better. Not all are necessary in all cases. Use your best judgement.
[4]Always have the future in mind, it will be here sooner than you think. Try avoiding having the feature come home to roost.

4. Don't be an a**hole, don't be spineless either. Stand for tactical virtues: courage, honor, strength, and mastery

5. Avoid isolationism (like the plague): Engineering doesn't work alone. Collaboration with other teams is paramount to your health and the health of the company. Conversation among different teams is imperative to everyone's success. Only by being aware of each other we can maximize and optimize the fruits of our labor

6. Say NO: Can you add this feature? Can you join this meeting? This is just a small change, right?

   Unless absolutely necessary, please say NO to interruptions of your time and scope creep. When in doubt, speak to your manager, your manager's manager, and whomever else you need to speak to. Also, build enough arguments to support you view. This will make the situation clear to outsiders and facilitate the decision making process

7. Growth: Always be learning, always be growing. Keep your skills sharp. What areas would you like to study? Engineering? Management? Marketing? Sales?

   Not only you will be benefiting yourself, but also the entire team. Your contributions will always be made by the best of you

8. Empathy: Put yourself in the shoes of who is going to use this. Are you solving a problem or just shifting work to them?

   Giving a customer (external or internal) a data structure and asking them to code the rest to their needs may be a little too much to ask from them. Asking for the upfront investment of countless man-hours to integrate with a system may be a tough sell.

   Avoid falling into the *Stack Fallacy* trap. The occurrence of this misstep is counterintuitively common in high-tech companies. It *"'highlights the tendency of engineers to overweight the value of their own technology and underweight the downstream applications of that technology to solve customer problems '... 'Stack fallacy is the mistaken belief that it is trivial to build the layers above yours.'"*[5]

   There may be cases where the product should be exactly that, as raw as possible. However, that is the exception, **not** the general case. Think about the complete solution, then see how far into it the implementation should go

9. Time and resources: Neither is infinite. Developing a reasonable ability to estimate the effort necessary to implement something is a necessity. We are not going for precision, but for accuracy.

   If you said it was going to take a month and ended up taking two months, that is okay (could be better). However, the project was implemented as intended, involving roughly the team and materials estimated.

   Try thinking about the big picture when trying to estimate: How long will it take? How many people? What equipment is needed? Are other teams involved (e.g. product, marketing, sales, legal, finance)? Will you need to hire personnel or an outside vendor?

10. How would you bring this to market? For what job would someone hire your product or service? Who would use it? And why? How much would anyone pay for it? Is it an internal or external product? Perhaps this is a candidate for an open source tool?

    You may not be selling to an outside customer, but you may need to "sell" to another, internal, team, convincing them to use it.

    There are many possible strategies to go to market. You should consider a few, pick one to try, and iterate over them until you find the most effective one.

References

[1] [Online]. Available: https://en.wikipedia.org/wiki/A_picture_is_worth_a_thousand_words

[2] R. J. J. V. Erich Gamma, Richard Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[3] J. Erickson. (2018) Algorithms. [Online]. Available: http://jeffe.cs.illinois.edu/teaching/algorithms/

[4] P. Morin. Open data structures: An introduction (open paths to enriched learning). [Online]. Available: http://opendatastructures.org

[5] J. F. Woods. (1991). [Online]. Available: https://groups.google.com/forum/#!msg/comp.lang.c++/rYCO5yn4lXw/oITtSkZOtoUJ

[6] T. H. D. S. D. Clayton M. Christensen, Karen Dillon, *Competing Against Luck: The Story of Innovation and Customer Choice*. HarperBusiness, 2016.

[7] W. E. Deming, *The Essential Deming: Leadership Principles from the Father of Quality*. McGraw-Hill Education, 2012.

[8] E. S. Raymond, *The Art of UNIX Programming*. Addison-Wesley, 2003.

Acknowledgements

Thanks to Trevor Masters for his review of and suggestions to the manifesto.