Principles for better design

7-9 minutes

---

credit

Design is a broad topic. This post won't approach the adjective qualifying a style with simple forms and a pure appearance. No. Design is broader than a sleek car or an harmonious living room.

Design is a creative activity that aims to invent, improve and solve problems. Life is full of them, in various domains, which pushes people to eventually become a designer of something. Whether you're conceiving a recipe, an apartment layout, a software or a rocket you are designing. By learning the fundamentals, you will arm yourself for various situations. Here is a list of the best pragmatic principles I've learned to better design. In spite of their obviousness and renown, my experience has shown me that some of these principles are often forgotten.

Fun fact: In english, *to design* means both *to draw* and *to conceive according to a plan*. Similarly in French, the word *dessin* (a drawing) derives its etymology from the word *dessein* (have the intention).

Incremental improvement

Believing you can do it right the first time is naive at best, pretentious at worst. **You will fail and have to accept it**. Do not postpone but instead **release fast**, get feedback early and iterate to make it better. You will find many references, notably on Lean Startup, DevOps culture or Site Reliability Engineering, which encourage **a quick redesign capability rather than a perfect design at the first place**.

> To write clean code, you must first write dirty code and then clean it. […] Learning to write clean code is hard work. It requires more than just the knowledge of principles and patterns. You must sweat over it. You must practice it yourself, and watch yourself fail.
> Robert C. Martin — Clean Code: A Handbook of Agile Software Craftsmanship

Reuse what exists

It is unlikely that the problem you face is fundamentally new. Before making any effort, look for **existing solutions** or proven practices. **Be humble** and accept you can't compete with years of expertise on a subject. If so, the motivation to develop the new model must come from reaching the limits of the previous one. In software engineering for instance, design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Don't limit yourself to the area of the original problem, sometimes a similar problem has been found and solved long before in **a different field**. Learn about queuing theory will help you to write an operating system scheduler as well as being frustrated in a supermarket queue. Solutions come from fields of experiences and theories that you would not necessarily expect.

Five whys

The Five whys technique, originally developed by Toyota, has been mostly used for exploring the root cause of a defect in a system. It involves asking the relevant question starting with a why in order to find the source which cause the failure. Adapt this technique to find out what really matter in our design. For example, "*We need a program which exports our documentation to PDF*":

1. Why? - So the export can go on the website.
2. Why? - Because our customers has to read it.
3. Why? - Because they require it to maintain the software.
4. Why (don't they have access to the documentation like everyone else)? - Because they haven't been integrated into our platform.
5. Why? Because they have never been on-board on the platform.

Decision: instead of developing a complex program which exports PDF, write a simple on-boarding guide for the customers.

This situation often occurs when the given requirement includes the design. Before answering the question *"How do we do it?"* directly, the first and most important step is to remember "*Why do we do it?*". Once you have understood the **root reasons** and constraints for the requirement, it clears the path on how to get there and can **prevent** you from taking **very long detours** in addition to **simplifying** the design.

Keep it simple stupid

Resolve problems, don't create them. You should be suspicious of providing a solution that makes the overall system more complicated than before. Remember: You'll have to maintain everything you build, so you'd better build as little and simply as possible.

Simple designs are easy to **learn**, easy to **teach**, easy to **use** and therefore easily **appeal** to customers. It forces the design to be intuitive and obvious, reducing the learning curve and facilitates the adoption. Like Denis Diderot says in Pensées sur l'interprétation de la nature:

> These books [of Sthal and Newton] were just waiting to be heard, to be valued for what they were worth; and it would not have cost their authors more than a month to make them clear; that month would have spared three years of labor and exhaustion to a thousand good spirits.

But what is a simple design? In the Simplicity chapter of the Site Reliability Engineering book of Google, they mention a great quote from the French poet Antoine de Saint Exupery about perfection which could fit very well with simplicity:

> perfection is finally attained not when there is no longer more to add, but when there is no longer anything to take away - A. de Saint Exupéry, Terre des Hommes

Perfect is the enemy of good enough

The moka coffee maker may not produce perfect coffee, but it requires so little maintenance compared to a large coffee machine with radiators, pipes, grinder, etc. that it makes the compromise "complexity / coffee taste" great.

Remember the Pareto principle: in general **80% of a things can be done in 20%** of the total allocated time. Conversely, the hardest **20% left takes 80%** of the time. Perfection requires infinite time and energy. This is impossible and therefore should not be part of your design.

> Beware of the perfection pitfall and design your architecture only as good as necessary and not as good as ultimate possible. - Ralf S. Engelschall

Postpone complexity

Simplicity and incremental improvement principles give birth to a another principle: **Postpone complexity**. Complexity lies in early optimization, as Donald E. Knuth points out in his article Computer Programming as Art with "*premature optimization is the root of all evil*". Focus on designing something that works instead of something finely tune that does not.

Complexity lies also in early generalization. Focus on designing something that works in a single use case rather than something that will not work in many. When designing a coffee machine, first design something that makes a simple black coffee. If you plan ahead to include all extra options for latte, cappuccino or irish coffee you will end up not making a simple back coffee. As elegantly summarized by Ralf S. Engelschall: "*Use before reuse*".

Conclusion

Do you think there's a principle missing? Send me your comments! This list will certainly be extended and refined, subscribe to the newsletter if you wish to be notified about it. In the meantime, how are you going to ensure that you do not forget to implement the principles you have just learned?