antirez.com /news/112

The mythical 10x programmer -

9-12 minutes

antirez 1132 days ago. 631525 views.

A 10x programmer is, in the mythology of programming, a programmer that can do ten times the work of another normal programmer, where for normal programmer we can imagine one good at doing its work, but without the magical abilities of the 10x programmer. Actually to better characterize the “normal programmer” it is better to say that it represents the one having the average programming output, among the programmers that are professionals in this discipline.

The programming community is extremely polarized about the existence or not of such a beast: who says there is no such a thing as the 10x programmer, who says it actually does not just exist, but there are even 100x programmers if you know where to look for.

If you see programming as a “linear” discipline, it is clear that the 10x programmer looks like an irrational possibility. How can a runner run 10x faster than another one? Or a construction worker build 10x the things another worker can build in the same time? However programming is a design discipline, in a very special way. Even when a programmer does not participate in the actual architectural design of a program, the act of implementing it still requires a sub-design of the implementation strategy.

So if the design and implementation of a program are not linear abilities, things like experience, coding abilities, knowledge, recognition of useless parts, are, in my opinion, not just linear advantages, they work together in a multiplicative way in the act of creating a program. Of course this phenomenon happens much more when a programmer can both handle the design and the implementation of a program. The more “goal oriented” is the task, the more a potential 10x programmer can exploit her/his abilities in order to reach the goal with a lot less efforts. When the task at hand is much more rigid, with specific guidelines about what tools to use and how to implement things, the ability of a 10x programmer to perform a lot of work in less time is weakened: it can still exploit “local” design possibilities to do a much better work, but cannot change in more profound ways the path used to reach the goal, that may include, possibly, even eliminating part of the specification completely from the project, so that the goal to be reached looks almost the same but the efforts to reach it are reduced by a big factor.

In twenty years of working as a programmer I observed other programmers working with me, as coworkers, guided by me in order to reach a given goal, providing patches to Redis and other projects. At the same time many people told me that they believe I’m a very fast programmer. Considering I’m far from being a workaholic, I’ll also use myself as a reference of coding things fast.

The following is a list of qualities that I believe make the most difference in programmers productivity.

* Bare programming abilities: getting sub-tasks done

One of the most obvious limits, or strengths, of a programmer is to deal with the sub-task of actually implementing part of a program: a function, an algorithm or whatever. Surprisingly the ability to use basic imperative programming constructs very efficiently in order to implement something is, in my experience, not as widespread as one may think. In a team sometimes I observed very incompetent programmers, that were not even aware of a simple sorting algorithm, to get more work done than graduated programmers that were in theory extremely competent but very poor in the practice of implementing solutions.

* Experience: pattern matching

By experience I mean the set of already explored solutions for a number of recurring tasks. An experienced programmer eventually knows how to deal with a variety of sub tasks. This avoids both a lot of design work, but especially, is an extremely powerful weapon against design errors, that are in turn among the biggest enemies of simplicity.

* Focus: actual time VS hypothetical time

The number of hours spent writing code is irrelevant without looking at the quality of the time. Lack of focus can be generated by internal and external factors. Internal factors are procrastination, lack of interest in the project at hand (you can't be good doing things you do not love), lack of exercise / well-being, poor or little sleeping. External factors are frequent meetings, work environments without actual offices, coworkers interrupting often and so forth. It seems natural that trying to improve focus and to reduce interruptions is going to have a non marginal effect on the programming productivity. Sometimes in order to gain focus, extreme measures are needed. For instance I only read emails from time to time and do not reply to most of them.

* Design sacrifice: killing 5% to get 90%

Often complexity is generated when there is no willingness to recognized that a non fundamental goal of a project is accounting for a very large amount of design complexity, or is making another more important goal very hard to reach, because there is a design tension among a fundamental feature and a non fundamental one. It is very important for a designer to recognize all the parts of a design that are not easy wins, that is, there is no proportionality between the effort and the advantages. A project that is executed in order to maximize the output, is going to focus exactly on the aspects that matter and that can be implemented in a reasonable amount of time. For example when designing Disque, a message broker, at some point I realized that by providing just best-effort ordering for the messages, all the other aspects of the project could be substantially improved: availability, query language and clients interaction, simplicity and performances.

* Simplicity

This is an obvious point that means all and nothing. In order to understand what simplicity is, it is worth to check how complexity is often generated. I believe that the two main drivers of complexity are the unwillingness to perform design sacrifices, and the accumulation of errors in the design activity.

If you think at the design process, each time a wrong path is pursued, we get more and more far from the optimal solution. An initial design error, in the wrong hands, will not generate a re-design of the same system, but will lead to the design of another complex solution in order to cope with the initial error. The project, thus, becomes more complex and less efficient at every wrong step.

The way simplicity can be achieved is to reason in terms of small metal "proof of concepts", so that a large amount of simple designs can be explored in the mind of the programmer, to start working from something that looks the most viable and direct solution. Later, experience and personal design abilities will allow to improve the design and find sensible solutions for the sub-designs that need to be resolved.

However each time a complex solution is needed, it’s important to reason for a long time about how the complexity can be avoided, and only continue in that direction if no better possibility is found even considering completely different alternatives.

* Perfectionism, or how to kill your productivity and bias your designs

Perfectionism comes in two variants: an engineering culture of reaching the best possible measurable performance in a program, and as a personality trait. In both the instances, I see this as one of the biggest barriers for a programmer to deliver things fast. Perfectionism and fear of external judice insert a designing bias that will result in poor choices in order to refine a design only according to psychological or trivially measurable parameters, where things like robustness, simplicity, ability to deliver in time, are often never accounted for.

* Knowledge: some theory is going to help

When dealing with complex tasks, knowledge about data structures, fundamental limits of computation, non trivial algorithms that are very suitable to model certain tasks, are going to have an impact in the ability to find a suitable design. To be a super expert of everything is not required, but to be at least aware of a multitude of potential solutions for a problem certainly is. For instance applying design sacrifice (accept some error percentage) and being aware of probabilistic set cardinality estimators, can be combined together in order to avoid a complex, slow and memory inefficient solution in order to count unique items in a stream.

* Low level: understanding the machine

A number of issues in programs, even when using high level languages, arise from the misunderstanding of how the computer is going to perform a given task. This may even lead to the need of re-designing and re-implementing again from scratch a project because there is a fundamental problem in the tools or algorithms used. Good competence of C, the understanding of how CPUs work and clear ideas about how the kernel operates and how system calls are implemented, can save from bad late-stage surprises.

* Debugging skills

It is very easy to spend an enormous amount of work in order to find bugs. The sum of being good at gaining state about a bug, incrementally, in order to fix it with a rational set of steps, and the attitude of writing simple code that is unlikely to contain too many bugs, can have a great effect on the programmer efficiency.

It is no surprising to me to see how the above qualities of a programmer can have a 10x impact on the output. Combined they allow for good implementations of designs that start from a viable model and can be several times simpler than alternatives. There is a way to stress simplicity that I like to call “opportunistic programming”. Basically at every development step, the set of features to implement is chosen in order to have the maximum impact on the user base of the program, with the minimum requirement of efforts.