To be successful in verification you not only need to possess the right technical skills, but you also need to possess the right mindset. Possessing the right mindset will lead you to success rapidly. Here are 3 things that I've found very important to keep in mind with everything you do in verification:

Verification vs. Debugging – Verification may be defined as "proving or establishing authenticity or validity" or "evidence that establishes or confirms the accuracy or truth of something". Perhaps the lexical meaning of the word should account for it, but the verification process sometimes wrongly begins with the premise that the design process delivers a more-or-less working RTL and the verification process is merely there as a safety net just to be a 100% sure that everything is working well. Kind of like Quality Control or something. Experienced verifiers and managers are very unlikely to err here, but others may easily fall into this conceptual pitfall, including many designers and project managers. This kind of premise is not only half true but also quite dangerous. In reality, the RTL design process produces **RTL code**. That's it. The verification process produces **RTL code that works**. And there's a big difference.

As a matter of fact, if I had my way, I would split verification into 3 distinct areas – A) VIP Development – that's the part where verification IP is being developed. This is a pure software engineering effort. B) Simulation – that's the part where fresh or immature RTL is being simulated and debugged. C) Closure – that's the part where the RTL is in good shape and regression suites are available to test and cover corner cases. Phase A is pretty much a standalone, self-contained, bounded effort. The other ones (and especially phase B) are totally opposite in nature. Design and Verification during these phases need to work hand in hand. Program the right mindset into your brain – i.e. that verification and design cannot be separated – and you're on your way to success.

Automation – "Manual verification" or "Features verified by visual inspection" are history.

Verification is all about building an **automated** system that eventually will check your design and find bugs in a push-button fashion. With everything you do in verification, you should keep that thought in mind. Interestingly enough, tool vendors promote automation in different ways, always wanting to take it to the next level (e.g. vManager, VMM Planner, etc.) but many users are not so enthusiastic about going there and prefer to keep it simple. Some of their concerns are valid – fully blown automation requires a significant effort to ramp and might be more suitable for big companies. Nevertheless, even in small teams, keeping automation in mind can do wonders to overall productivity. In these cases, Perl and similar scripting languages can do the job well. Don't settle for "visual inspection" or "manual testing". Instead, make your environment smart, make it robust, so that every single test provides a definitive pass/fail indication. Don't leave holes in your checkers and use scripts to help you in launching and handling simulations. Remember, the most valuable asset is your time – let the machine do the hard and repetitive work.

Safety Nets – Verification is and always will be a matter of redundancy. Why? Because it's done by humans and holes may exist in any verification layer. Coverage, for instance, is in many ways a safety net that validates and measures the quality of your generation (In the early days, when coverage had not yet been a metric for progress, it was merely used as a feedback on generation). When you're building a verification environment, keep "safety nets" in the back of your mind. For example, let's say you have a generic scoreboard that compares each item that comes out of the DUT with the respective item that comes in. If a mismatch occurs, an error is issued. But what if for some reason the input channel to the DUT is blocked in such a way that no items ever enter the scoreboard and nothing ever comes of the DUT? If you don't have an orthogonal checker that makes sure that at least one item has entered/exited the DUT you might fall into the infamous pitfall of "0 items in, 0 items out, TEST PASSED". There are, of course, less trivial examples. So what to do? A good methodology (not base class library) should be able to cover up for some user mistakes – for example, sampling coverage from monitors rather than generators will help make sure that the data you're sampling has really entered the DUT. But safety nets can also be much less sophisticated (albeit not less powerful). For example – In the regression phase, add a script that makes sure that no test has been left untested after a full regression run. Simple, but powerful. Remember, the more safety nets you add, the more confidence you have in your verification system.