SPECIFIC VS. GENERIC

What we've just done is get an understanding the core of OTP (conceptually speaking). This is what OTP really is all about: taking all the generic components, extracting them in libraries, making sure they work well and then reusing that code when possible. Then all that's left to do is focus on the specific stuff, things that will always change from application to application.

Obviously, there isn't much to save by doing things that way with only the kitty server. It looks a bit like abstraction for abstraction's sake. If the app we had to ship to a customer were nothing but the kitty server, then the first version might be fine. If you're going to have larger applications then it might be worth it to separate generic parts of your code from the specific sections.

Let's imagine for a moment that we have some Erlang software running on a server. Our software has a few kitty servers running, a veterinary process (you send your broken kitties and it returns them fixed), a kitty beauty salon, a server for pet food, supplies, etc. Most of these can be implemented with a client-server pattern. As time goes, your complex system becomes full of different servers running around.

Adding servers adds complexity in terms of code, but also in terms of testing, maintenance and understanding. Each implementation might be different, programmed in different styles by different people, and so on. However, if all these servers share the same common `my_server` abstraction, you substantially reduce that complexity. You understand the basic concept of the module instantly ("oh, it's a server!"), there's a single generic implementation of it to test, document, etc. The rest of the effort can be put on each specific implementation of it.

This means you reduce a lot of time tracking and solving bugs (just do it at one place for all servers). It also means that you reduce the number of bugs you introduce. If you were to re-write the `my_server:call/3` or the process' main loop all the time, not only would it be more time consuming, but chances of forgetting one step or the other would skyrocket, and so would bugs. Fewer bugs mean fewer calls during the night to go fix something, which is definitely good for all of us. Your mileage may vary, but I'll bet you don't appreciate going to the office on days off to fix bugs either.
Another interesting thing about what we did when separating the generic from the specific is that we instantly made it much easier to test our individual modules. If you wanted to unit test the old kitty server implementation, you'd need to spawn one process per test, give it

the right state, send your messages and hope for the reply you expected. On the other hand, our second kitty server only requires us to run the function calls over the 'handle_call/3' and 'handle_cast/2' functions and see what they output as a new state. No need to set up servers, manipulate the state. Just pass it in as a function parameter. Note that this also means the generic aspect of the server is much easier to test given you can just implement very simple functions that do nothing else than let you focus on the behaviour you want to observe, without the rest.

A much more 'hidden' advantage of using common abstractions in that way is that if everyone uses the exact same backend for their processes, when someone optimizes that single backend to make it a little bit faster, every process using it out there will run a little bit faster too. For this principle to work in practice, it's usually necessary to have a whole lot of people using the same abstractions and putting effort on them. Luckily for the Erlang community, that's what happens with the OTP framework.

Back to our modules. There are a bunch of things we haven't yet addressed: named processes, configuring the timeouts, adding debug information, what to do with unexpected messages, how to tie in hot code loading, handling specific errors, abstracting away the need to write most replies, handling most ways to shut a server down, making sure the server plays nice with supervisors, etc. Going over all of this is superfluous for this text, but would be necessary in real products that need to be shipped. Again, you might see why doing all of this by yourself is a bit of a risky task. Luckily for you (and the people who'll support your applications), the Erlang/OTP team managed to handle all of that for you with the gen_server behaviour. `gen_server` is a bit like `my_server` on steroids, except it has years and years of testing and production use behind it.

Source : http://learnyousomeerlang.com/what-is-otp