

I made a quick sketch of what I think should happen with my app. I read the article https://developer.algorand.org/articles/using-a-smart-contract-to-spawn-additional-smart-contracts-2/ again and here are the questions that came up to my mind:   
With this parent-children relationship the parent contract's logic is more modular and I can update just a single contract that is responsible for one or two methods only. The thing is that the customers of the bank should not be aware of the children contracts that the parent uses. But for each method called specific transactions and parameters can be required. Each method in the parent that is relevant to the customers should be described in a .json artifact so the ABI can do its job. And this method should get the right group of transactions and parameters and delegate the call with them with an inner transaction to the child, where the execution of the logic will happen. So far so good, but now come the questions and problems:

1. If I update the logic of the child so that different transactions and parameters are now required, then I would need to update also the parent contract delegate method, so now it checks for the correctness of the top-level transactions and parameters provided by the customer. This means updating the whole parent contract with a new approval program to point to the right new delegate method. Is this reasoning correct?

**Answer**: More or less it will be like that.

1. When I want to add entirely new logic to the parent contract I can just create a new child contract with the needed logic. But then I should add also a delegate method to this child, which again means updating the parent contract, right? I believe the same goes for deleting a child and removing a delegate call to it? My whole point is that it seems that trying to implement things in this way just makes maintenance harder and not so logical, because either way, the parent contract will have to be updated too. This makes me believe that it will be a lot easier if just everything happens in one contract, namely the parent and all the logic is already there for execution.

**Answer**: First of all, everything can not be reside in a single class, program, or contract in software world. Because when you reach massive amount of code, it will be harder to maintain or preserve entire state (world state) that connects to child state of child contracts. Deposit and Withdraw can be different contracts because they can be called from different external wallet or client application. When you used a client application in order for calling, you can reserve the state upon a contract, which means that either deposit or withdraw. However, if you put these contracts in a single application point, you cannot manage all external client calls when you need to add new business logic. This is totally depends on the use case. Another drawback is that the system would work in a centralized manner, not in a decentralized manner. (This point can be a research question, drawback, advantages)

1. Also if the customer can't know the app id of the children and it should only communicate with the parent contract, then how will the parent know which delegate method to call and where will it store the address or app id of the specific child? Should the address be hardcoded in the inner transaction, or should it be stored in a variable or in a global state? Can the parent hold somehow the addresses of all its children and be able to use the right address for the specific functionality wanted by the customer?

**Answer**: Should the address be hardcoded in the inner transaction, or should it be stored in a variable or in a global state? (This question might be a research question as well. )

A sample link upon your question.

https://stackoverflow.com/questions/42230532/getting-the-address-of-a-contract-deployed-by-another-contract

If you have simpler approach than methods of the above-mentioned links, you can compare and write them in the result and discussion chapters.

I am just confused about how should I implement the whole thing because the logic of a smart contract in Algorand can be updated without changing state, app id/address in contrast to Ethereum where such patterns are the only way to get upgradability.

From the email correspondence that we had so far:

Regarding the proxy pattern, I know what you mean and I already understood what upgradable smart contracts do and what is their purpose. The thing is that Algorand has some advantages when it comes to upgrading smart contracts in contrast to Ethereum. When I have a proxy (storage layer) and a couple of implementation contracts (logic layer) the code is more modular and if we update a certain logic part, we just upgrade one specific implementation contract. The thing is that with Ethereum we have to also update the address in the proxy to point to the new version, in Algorand this is not needed because we can update the approval and clear state programs without having to deploy a new smart contract and the app id and address remain the same, no change needed. The state connected with the updated contract also remains the same. So here comes my question, why do I need the proxy pattern at all, when I can just update the whole bank contract and the state will be still intact, the opt-in accounts won't need to change anything and the logic is upgraded? Also if the updated methods now want different types of arguments or different kinds of transactions needed, then this can be described in the .json artifact of the smart contract, and using the ABI the accounts could continue communicating with the updated logic of the bank. I also correspond with some of the developers of Algorand and they say that using the fact that smart contracts on Algorand are already updatable, makes such proxy patterns not beneficial and are not sure why I will need to design my app in this way to begin with. I believe that's why no example code shows proxy patterns in Algorand.

You asked me how do I plan to do the swapping of the approval and clear state programs? Well, I have an admin account that will create the bank smart contract. After that, the admin will just need to make a transaction to the bank SC with the new approval prog. and clear prog. as parameters, the logic part of the SC will see that the admin wants to update (only the admin account should have this right), and then after it executes the method that is responsible for the Update Transaction it will swap the programs with the new ones. If I implement it like this I also don't need anymore the Money Transfer contract that I used in my diagram.

Regarding the customers, I can't understand why should I module them as SCs and not just regular accounts. Did you mean that their bank accounts associated with them should be smart contracts? If so, this was the way I was planning to do it. What does the shareholder do, what functionality does it have more than a regular customer, I can't think of anything.

One thing that one of the developers sent me is this article right here https://developer.algorand.org/articles/using-a-smart-contract-to-spawn-additional-smart-contracts-2/ , he said that this has a lot in common with what I want to do. This also makes sense with the things that you told me about inheritance (parent-child relationship) and building a modular smart contract and having the upgradable contracts as list of addresses. The thing is that I am not sure how should I implement it exactly. If we go back to the example with the proxy pattern, then parent is the proxy and the contracts with the logic are the children. The thing that bothers me is that the accounts using the bank account should not know about the underlying children of the parent, but when I call a certain method in the bank, it will be delegated to such a contract and a certain group of transactions and parameters should be passed, but if I upgrade the child, then I should also update the parent, because the connection between them should always pass the correct parameters. Which makes the whole thing in my opinion useless again, because we have to update the parent contract every now and then and we are making the overall architecture just more complex than it should be. I am really confused and don't know how to start and 2 weeks have already passed, I will appreciate it if you can be more specific how exactly do you want me to go about this app. Thank you in advance and sorry for the long mail.

 Here is what I got as a message from one of the developers:

"Awesome stuff, thanks for spending so much time looking into this and trying to understand it before asking questions. It's a change from what people usually do. 😄 I do believe that you understand the benefits of just updating the smart contract on Algorand, and if this were coming from a company who were implementing something for Mainnet I would heavily advise them to not use the proxy design unless they could demonstrate a clear advantage of it. That said, I guess the exercise is more about doing it and then writing up about the pros and cons of this approach against the built-in upgradeable smart contract that Algorand offers?"

Is this really the point? I mean a lot better will be if I just describe what upgradable contracts are in blockchain, how it is done in Ethereum with proxy patterns, and then directly show with my application why on Algorand it is a lot simpler and efficient. What do you think?

**Answer**: You can do it. In order to compare them, you need to understand the following patterns as well:

1) Eternal Storage Pattern,

2) Proxy Delegate Pattern

3) Access Restriction Pattern

4) Registry Pattern

However, your priority is to show your use case by means of separated Algorand smart contracts.