Upgradeability: Good, Bad, Ugly

Mehdi Salehi, Jeremy Clark, and Mohammad Mannan

Concordia University

Abstract.

1 Introductory Remarks

2 Classification

2.1 Retail Changes

This is not a standardized pattern. The development team must consider the ways to upgrade the contracts before deploying the smart contract. Known patterns are different on the level of intervention to change the logic that they need to change in the future. The amount of changes are limited and system design can not be changed after deployment and just some system variables can be changed. We will describe three famous patterns here:

Parameter Configuration . The easiest way to upgrade the logic of the smart contract is to have some critical parameter that can change the whole logic of the system. For instance, in economy we have different variables which have effect in the interest rates. By changing those variables the governors will response to changes needed for the system. In this model we have a setter function to change the upgradable parameters if the system needs upgrades. The best example for this type of upgradeability is MakeDao project. In Maker there are some variables like Dai Saving Rate (DSR) or Stability fee that can be changed through governance vote. The logic behind the smart contract and the tokenomic of the Dapp completely depends on these variables.

Strategy pattern . The strategy pattern is an easy way for changing part of the code in a contract responsible for a specific feature. Instead of implementing a function in your contract to take care of a specific task, you call into a separate contract to take care of that and by switching implementations of that contract, you can switch between different strategies. An example for this pattern is Compound project and how they used strategy pattern for their interest rate model. There is a interest rate model contract in Compound that can be changed during the time.

Pluggable Modules . In this pattern we have a core contract that have some immutable features and then new contracts generated by the main contract and each have some or all features of the main contract. This pattern is mostly used in wallets and DeFi services like DeFi saver and InstaDapp. Users can decide to add new features into their wallet.

2.2 Wholesale Changes

In contrast to previous session, sometime we need to change the whole or a big part of the logic of our smart contract. This update could be a response to an incident happen to the smart contract or a planed upgrade of the system. Before deployment, the core developer team should have a plan for the upgrade events. We categorized these types into two main classes:

Contract Migration In the migration plan we should write a completely new contract with our desired new logic. In migration method our new version contract doesn't have any communication with the previous versions. The challenges we face in migration method are:

1. Grab the needed data (from previous contract or new data): It depends on the data type. It is easy for simple data structures (*e.g.,* uint, address, or even arrays) to collect the data just by reading storage slots from the 0 slot. we should take care of complex structures (*e.g.,* mapping) in the latest versions of our contract by adding event updates whenever a data added to a mapping variable. In case of an upgrade we can use Logs to find storage slot (using key hash) and collect the data. Sometime we need to push new data into our upgraded smart contract. For example in airdrops we need new coin to be added to some specific addresses.
2. Push the data into the new contract: Using Constructor, we can use batch transfer function with arrays of addresses and amounts as inputs. This way we can push lots of data using a single transaction. One limitation here is block gas limit. If we exceeds the block gas limit we need to push all data in different blocks (pausing in the first block and unpause at the end). Recently, Devs are using merkle distribution tree to push data on to the smart contracts. The most important thing here is the cost of pushing data to the new version. It depends on 1)the number of storage slots to be updated and 2)Method used to push the data on-chain. (Can be tested).
3. Stop the previous contract: Suppose that we have a token contract and we want to migrate to a new version. We should be confident that nobody can use the previous contract. If not, it is possible that a person sell a token from previous contract (which should be valueless after migration) to a person who doesn't aware of the migration plan.Because of the decentralized nature of the blockchain and Dapps you cannot reach to your customers to alert them from using the previous contract. One way to do that is have a pause option the your contracts and pause the old versions before migration.

Contract Migration is less riskier than other types of upgrades, not cost effective compare to some upgradeability types but more decentralized to the other solutions. Also, it's not good for frequent updates.The other advantage of this method is that it removes transaction gas cost needed for patterns like proxy, registry or call-based methods.

Data Separation patterns The other type of wholesale methods is to separate data and logic part of our codes. In the case of the upgrade we can keep the storage contract and just upgrade the logic contract and link the new version of logic contract to the storage contract. There is a debate on whether this type of upgradeability is cheaper or not in comparison to migration method. But, this method is more efficient for Dapps in which we need frequent updates. The other important issue here is who decides on the changes we need for the system which we will discuss on further sections. Here we have 2 different choices using Call method and Delegate Call method to link storage and logic contracts together.

Call based patterns In this type the interaction between logic and storage contact is handle by Call opcode in Ethereum Virtual Machine. In call based patterns user is supposed to call the logic contract and the logic contract will call the storage contract. The logic contract is the one that can be upgraded.

There are two concerns in this approach: how to store data and how to perform an upgrade.

Storage. The easiest way to store data in storage contract is to have a modifier on the setter functions in the storage contract that allow just the logic contract to change the variables. The owner of the contract can change the address of logic contract for the modifier. In this approach for adding a new persistent variable, a new data contract should be deployed which may be costly in case that the application needs lots of upgrades.

The other way to store data is so called Eternal storage (ERC930). Eternal storage uses mapping (key-value pair) to store data, using one mapping per type of variable. The EVM storage layout and how it handles mapping helps the Eternal storage pattern to be more amenable to evolution but also more complex.

Upgrade implementation. There are three main ways to implement upgrades using data separation pattern. The easiest way is to change the ownership of storage contract into new upgraded logic contract and then **Pause** the old contract or set its pointer to 0x0 address. The other solution is to forward the calls receive by the old contract into the new logic contract. The last option is to set a proxy contract that just keeps the address of logic contract and call into logic contract.

DelegateCall-based upgrades Similar to call based patterns here we have two contracts, Storage and Logic contract. we may have more than one logic contract. The difference here is that the user is calling storage contract first(called proxy contract), and the proxy contract DelegateCalls to the logic contract(s). The main difference between delegatecall and call-based approach is that in delegatecall proxy pattern the storage layout of proxy and logic contract should be the same. The difference between storage layouts will result in storage clashes. There are three different methods to mitigate the risk of storage clashes:

Inherited Storage. In this method the proxy contract and all logic contracts are inherited from a storage contract that contains storage variables. Using this method we are confident that the proxy and logic contracts are using the same

storage layout and storage clashes will be mitigated. After deployment if we need new logic contract with new storage variables, we should deploy a new storage contract that inherits the previous storage contract. Then the new logic contract must inherit the new version of the storage contract.

This method is not efficient because of variables that declared but not used in some logic contracts. On the other hand, each logic contract is coupled with a storage contract and it is hard to take care of this track.

Eternal Storage. In this pattern, we defined mappings for all variable types that we need to use in our logic smart contract. For storing mapping variables EVM selects random slots on the storage based on the variable's name so we can mitigate the clashes using this randomness.

The main problem of this type is that the logic contract and all other contracts that are using the storage must use the mapping structure to access the storage variables and use complex syntax whenever they want to access a variable. This also results in the gas usage inefficiency because we need to call and update a mapping each time we need to change a variable.

Also it is hard to use eternal storage for complex variables like mappings and structure (need mapping of mapping pattern). Also finding a state variable of the proxy smart contract is hard because we store them in arbitrary slots of the storage.

Unstructured Storage. The other way of mitigating the storage clashes is to assign some randomly selected slots to critical variables like address of logic contract. For instance, open zeppling uses hash of "org.zeppelinos.proxy.implementation" to store the address of the logic contract in this slot.

The downside of this approach is that we need getter and setter function for each variable. We also can use unstructured storage for simple variables and not for mapping and structures.

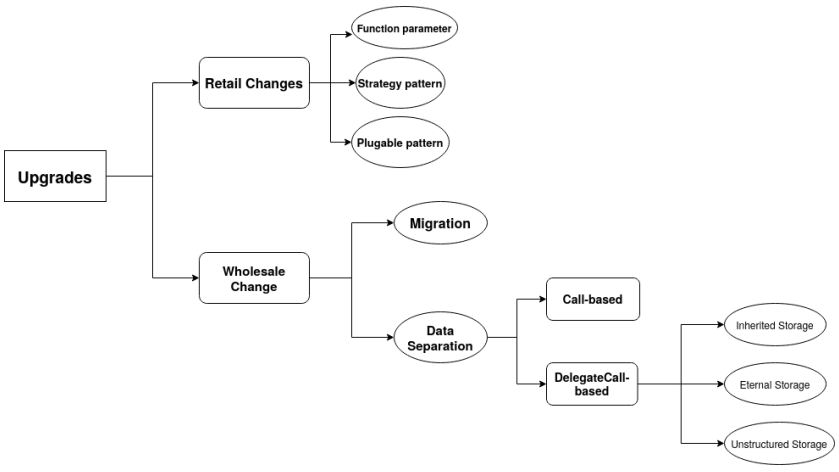

Fig. 1. Classification

3 Evaluation of different methods

4 Upgrading process

4.1 decision maker(s)

There is a debate on who is responsible for upgrading a Dapp. Different systems can choose one of these schemes to upgrade their Dapp depending on the complexity of the system, frequency of the changes needed for the system and how fast does the system need to upgrade in the incidents.

Externally owned Address The easiest and the fastest way to upgrade a system is through a single address which is the owner of smart contract. This is the most centralized solution we have for upgrading a system. The main problem with this issue is the security of the system because it only depends on a single private key hold by the owner. In case of malicious party or if an attacker find the owner's private or if the owner lose the key the entire system is on the risk.

First Dapps on the ethereum blockchain used this method for the upgrade but it is not used these days because it is far from the idea of *Decentralization*.

Multi-sig A *m out of n* Multi-sig wallet is a smart contract that can manage a transactions only if m number out of a specified n EOAs agree and sign the transaction. We can use address of a Multi-sig wallet as the owner of the system. In case of a upgrade or responding to an incident m number of the governors can permit to upgrade the system.

This is a better answer to the decision making of the upgrade compare to using an EOA in case of centrality while keeping the speed of an upgrade process. However, it is not decentralized. One way to reduce the level of centralization is to use different trusted teams who are stakeholders of the system in the multi-sig wallet.

Governance Voting The most decentralized way to decide on a system change is to do it using decentralized voting. This can be done by distributing governance voting tokens to the community and then they can vote on a change proposal by staking their voting token.

There are some critique to this method. Governance by voting has an inherit time delay to the upgrading process. This raises a problem when the system needs an instant upgrade (*e.g.,* responding to an incident). This means we need another mechanism to quickly fix bugs and upgrade the system on the event of incidents in conjunction with the voting process (*e.g.,* Global shutdown in MakerDAO).

It is also not cost-efficient for the voters because all token holders must send a transaction and needs to pay network fee.

The other problem with this method is fair distribution of the tokens. If the governance token does not distribute fairly and the majority of tokens granted

to the limited number of users, then it is very similar to the multi-sig method which is more costly and complex. Because, whales of the governance token can vote to any desired change of the system similar to the multi-sig.

4.2 Mitigating risks

Timelocks

Thereshold

Pausable

Escape Hatches

Front-Runnign

5 Measurement study

References