UDC 004.75

DOI: 10.18372/2073-4751.82.20372

Serebriakov R.O.,
orcid.org/0009-0002-1159-4708,
e-mail: r.serebriakov@kpi.ua,
Klymenko I.A., Doctor of Technical Sciences,
orcid.org/0000-0001-5345-8806,
e-mail: klymenko.iryna@lll.kpi.ua

ALTERNATIVE APPROACHES FOR SMART CONTRACT UPGRADEABILITY

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"

Introduction

One of the key requirements for software is upgradeability. This is especially important in the context of decentralized applications based on blockchain technology. Once deployed, smart contracts become immutable, making it impossible to update them directly. This nature of blockchain technology guarantees the security and irreversibility of data, but at the same time creates significant limitations for the applications development [1].

Updating program code is necessary for several reasons. First, even thoroughly tested code can contain bugs or vulnerabilities that need to be fixed after deployment. Secondly, the used approaches may become outdated over time, and the implemented libraries may lose support or require changes in the way they are interacted with. Thirdly, user needs may change over time, so to keep the project relevant, developers must be able to add new functionality and adapt to the current needs of users.

Decentralized applications built on the blockchain should also be able to be updated in the same way as traditional software to stay relevant. Imagine a situation where your favorite apps, messengers, or social networks have not received updates since their launch. You would quickly lose interest in them and start using competitors' products that provide more functionality and are constantly updated to meet user requirements. Therefore, this approach would only lead to a loss of user interest and further displacement from the market by newer products.

Thus, a key problem occurs: how to ensure the ability to update smart contracts in an environment that, by its nature, does not allow changing the program code after its publication? One of the most common approaches to implement the updating process in a blockchain is using the Proxy Pattern [2]. But despite its advantages, this approach still has some limitations, so other approaches are being explored to enhance flexibility, maintainability, and adaptability of smart contracts within decentralized environments.

Literature review and problem statement

Proxy Pattern is a smart contract architecture template that allows you to implement the updating process by dividing the content of a smart contract into a proxy contract and a logic contract [3].

The main idea is that the user interacts only with the proxy contract, which stores the state of the contract but delegates the execution of the logic to another contract (Fig. 1). This delegation takes place through the delegatecall instruction built into the EVM (Ethereum Virtual Machine). An important feature of delegatecall is that it executes the code of an external contract in the context of the caller's memory, which means that all state changes occur in the proxy, not in the logical contract.

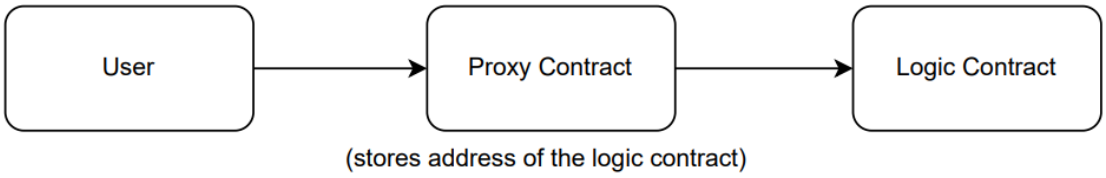

Fig. 1. The general scheme of Proxy Pattern

In the basic implementation, a proxy contract performs the following functions:

- stores the address of the logical contract in a special memory slot;
- has a fallback function that handles all calls and performs delegation via the delegatecall directive;
- allows changing the address of the logic contract while keeping its own state.

This allows developers to update the application's functionality without any need of data migration or changing the address of the main contract, which is especially critical in cases where the contract is already deployed and used by users.

The need in smart contract upgradability became obvious after a series of serious incidents in the Ethereum network. One of the most famous was the Parity Multisig Wallet Hack in 2017, when a hacker exploited a vulnerability in the code of a multisig wallet and stole more than 150,000 ETH [4]. This situation clearly demonstrated that the immutability of smart contracts, which is a key property of the blockchain, can turn into fatal consequences if there is a mistake in the code.

At the same time, the OpenZeppelin platform, which provides tools for developing smart contracts, has begun actively researching and developing patterns that allow separating the contract logic from its state, and thus implementing the possibility of updating.

It all began with upgradeability using Inherited Storage. The main goal was to simply delegate logic through the delegatecall directive, with the logic-contract and proxy-contract having an identical variable storage structure. This approach turned out to be too fragile: even a slight change in the order of variables in a logic contract could lead to a storage collision - a conflict of memory positions, which could cause corruption or loss of data.

To solve this issue and avoid such collisions, the upgradability using Unstructured Storage approach was proposed [5]. The idea was to store critical proxy contract variables, such as the address of the logic contract, in special memory slots that were randomly selected. It provided an almost absolute chance that such slots would not collide with the variables of the logical contract, even if their structure changed, making the chance of a storage collision almost impossible. This provided more convenience during the contract update, as using this approach, there was no need to deal with the problem of placing data inside the contract.

Then another problem arose: how to avoid accidental or malicious calls to proxy contract functions that have the same name as the logic contract functions and vice versa. For example, if a proxy contract had an upgradeTo() function to update the address of a logic contract, and the logic contract had the function with the same name, a common user could call the wrong function and thus cause critical errors. To solve this problem, the Transparent Proxy Pattern was developed [6]. Its essence lies in the fact that common users should have access only to delegated functions, i.e. functions that are inside the logic contract. And only the administrative address specified in the proxy contract constructor has access to the special functions of the proxy contract itself. This approach ensures a clear separation of roles, which makes it impossible for common users to call the administrative functions of the proxy contract, and at the same time forbids the administrator to access the functions of the logic contract.

Even though the Unstructured Storage approach resolves conflicts between a proxy contract and a logic contract, the problem of storage collision has not disappeared completely, as there is still a risk of collisions between different versions of logic contracts.

For example, let's consider a simple case. The initial version of a logic contract contains the address public _owner variable, which is usually placed in the first memory slot. After updating the logic contract, the developer changes the variable structure, and the new version stores the address public _lastContributor variable in the first slot. In this case, any assignment of a value to _lastContributor automatically overwrites the previous value of _owner, even if the variable is no longer directly used. This is a classic example of a storage collision between versions, which can potentially lead to data loss or a violation of the logic of the performed functions [7].

To avoid such consequences, developers should maintain strict compliance when making changes in the logic of contracts. In particular:

- use design patterns that ensure the consistency of variables;
- never change the order or type of variables that have already been used in previous versions;

So, although Proxy Pattern provides a mechanism for updating without losing state, it also imposes strict requirements on the design and compatibility of logic versions, which requires high technical skills from developers.

One of the key advantages of Proxy Pattern is the ability to update logic without losing state. Due to the use of the delegatecall instruction, the state is completely stored in the proxy contract, independently of the logical implementation. This allows developers to update the contract code without the need to migrate data or change the address, which is especially important in decentralized systems with many active users [8].

Another important advantage is transparency for the end user. Interaction always occurs through the same proxy contract address, no matter how many times the logic has changed. This simplifies integration with external services and allows to avoid problems with updating addresses in client applications.

However, despite its advantages, the use of Proxy Pattern is accompanied with significant technical challenges. Implementation of this approach requires a deep understanding of EVM architecture, delegatecall procedure, memory, and internal variable allocation [9]. Incorrect implementation can lead to vulnerabilities such as overwriting variables or inability to access specific functions.

The most common risks when using Proxy Pattern include:

- storage collision between different versions of logical contracts;
- delegation of privileged functions that can be called by malicious users;
- function name conflicts between proxy and logic contract functions.

After all, Proxy Pattern significantly limits the freedom of changes in the logic structure. All new versions of the contract must fully correspond to the original location of the variables in memory. And any deviation, such as changing the variable type, order, or adding variables in the middle of the structure, can cause data corruption.

These problems are especially critical for systems where the logic can change radically, or where it is important to have modularity, strict component isolation, simplicity of updates, and transparency of connections. So, despite its popularity, the Proxy Pattern is not always the best option for implementing blockchain upgradability.

The purpose and objectives of the study

The purpose of this study is to overcome the limitations of smart contract upgradeability mechanisms in decentralized environments, particularly those associated with the traditional Proxy Pattern. To achieve this, the study sets out the following objectives: to develop and propose a more flexible and modular method for updating smart contracts as an alternative to Proxy Pattern, to compare both approaches and to integrate them into a

hybrid architecture that combines their advantages, offering an enhanced level of upgradeability in decentralized environments.

The Router Contract approach as an alternative to the Proxy Pattern

Router Contract is a centralized registry that stores the current addresses of functional components of a decentralized system. Instead of direct or delegated interaction between contracts, all calls to other modules are made through the Router Contract (Fig. 2).

During initialization, each newly created contract receives a link to the router. So when it is necessary to interact with another component, the contract addresses the router with a unique identifier, such as the contract name, and receives the current address of this contract. Such approach allows you to implement dynamic call routing and centrally manage dependencies between contracts, reducing the strict binding of components in the system.

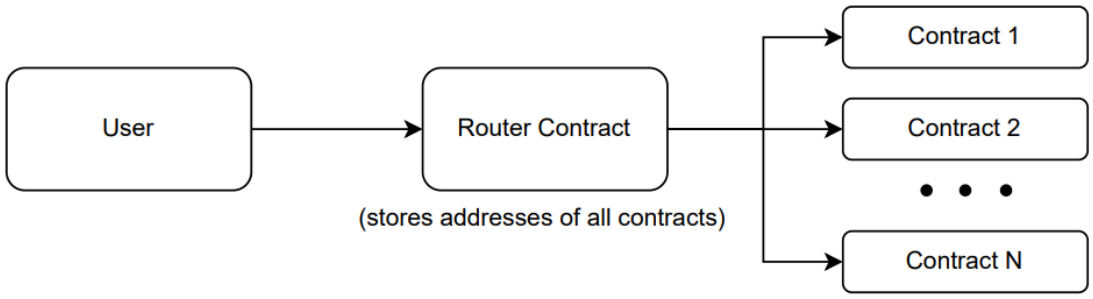

Fig. 2. General scheme of the approach using Router Contract

This implementation includes the following functions (Fig. 3):

- storing current contract addresses in the routes mapping
- adding a new address using the addRoute function. This function checks that there is no record for this name yet to avoid accidentally overwriting an already registered contract.
- changing the contract address using the changeRoute function, which allows you to flexibly update the system without disrupting its structure. Only the administrator has the right to change routes.

To increase security, you can also add to the addRoute and changeRoute functions a check for the existence of a smart contract at the transmitted address using extcodesize(routeAddress). This check will help to avoid registering non-existent or malicious addresses.

All architecture features of Router Contract approach will be:

1. Immutability of the access point - contracts do not interact with each other directly, instead they use a centralized access point - Router Contract. This allows to maintain a stable architecture structure during updates.
2. Centralized dependency management – contract addresses can be changed in a centralized way, without changing the logic of other components. This makes upgrades much easier.
3. Easy to implement - no delegatecall, assembly directives or complex memory makes this approach easy to develop, test, and audit.
4. Control over the system state - key variables, including the system state, can be stored in a centralized way in the Router Contract. This reduces the risk of data loss when updating logical components.

```
contract RouterContract {
    address public admin;
    mapping(string => address) public routes;

    constructor() {
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Not authorized");
        _;
    }

    function addRoute(string memory routeName, address routeAddress) external onlyAdmin {
        require(routes[routeName] == address(0), "Route already exists");
        routes[routeName] = routeAddress;
    }

    function changeRoute(string memory routeName, address newRouteAddress) external onlyAdmin {
        require(routes[routeName] != address(0), "Route does not exist");
        routes[routeName] = newRouteAddress;
    }

    function getRoute(string memory routeName) public view returns (address) {
        return routes[routeName];
    }
}
```

Fig. 3. Basic implementation of the Router Contract

The router approach has a number of advantages that make it an interesting alternative to the Proxy Pattern, especially for modular and dynamic systems. First of all, the router architecture is easy to implement: it doesn't require knowledge of the delegatecall instruction, EVM internal memory features, or the use of assembly directive. This significantly lowers the entry threshold for developers, making it easier to write tests and perform security audits.

Another significant advantage is the high level of flexibility when updating the logic. Since each module stores its own state and is not bound to the strict data structure of the proxy contract, a new version of the logic contract can be implemented with a changed set of variables, different logic, and without respecting the previous memory structure. This opens up the possibility of radical refactoring or functionality extension without the risk of collisions.

Centralized management of contract addresses is also an important advantage. Having a single entry point that controls the connections between system modules allows you to quickly update components, coordinate the integration of new services, and maintain the integrity of the architecture without the need to modify dependent modules.

At the same time, the implementation of Router Contract has certain limitations that must be taken into account when using it. The most important is the lack of a state delegation mechanism. In this architecture, each logic contract has its own memory, so changing it leads to the loss of state unless another way to save it is provided. To solve this problem, developers should design centralized state storage, for example, directly in the router or in a separate State Contract. So if you need to transfer a large amount of data about the state of the contract, this approach is not the best.

Another aspect is the need for preliminary architecture design, since in the Router

Contract approach, contracts do not directly address each other, instead they receive the addresses of target modules through a router. This requires a clear naming scheme for routes, predefined interfaces for interaction between contracts, and standardized call logic.

Gas consumption experiment based on basic implementations of both approaches

Likewise to the study [10] comparing the Proxy and Diamond approaches, lets also consider the gas consumption between the basic Proxy and Router implementations empirically. The Proxy Pattern has a delegatecall mechanism for calling logic from another contract, which is potentially more costly than a direct call to the Router Contract. But Router has a mechanism for getting the address of a contract from route mapping. In order to find out which approach is more economical, lets create implementations of both approaches in the Hardhat environment using the Solidity programming language and empirically obtain data on gas consumption by both approaches. In both cases, the logical contract will implement a complex mathematical operation with a cyclic calculation, where the number of iterations is passed as an argument.

After performing several experiments with a variable number of iterations (Table 1), it turned out that Proxy Pattern consistently consumes about 14.3 thousand units of gas less under any load. This result leads us to the conclusion that the difference in consumption does not depend on the complexity of the logic, but is explained by the difference in the call mechanism, i.e:

- delegatecall in the Proxy Pattern context without external transitions.
- lookup in mapping in Router Contract with an external call, which creates additional cost overhead.

Table 1. Results of gas consumption comparison for Proxy and Router depending on the complexity of calculations

Number of iterations	Proxy Pattern	Router Contract
100	70,934	85,262
500	237,746	252,074
1,000	446,246	460,574
10,000	4,199,246	4,213,574
50,000	20,879,246	20,893,574

Thus, the Proxy Pattern approach demonstrates higher efficiency in terms of gas consumption even in cases with complex logic. At the same time, the difference is stable and becomes insignificant in the context of large computations that require a lot of gas, but can become quite noticeable when calling simple functions.

Comparative analysis of Proxy Pattern and Router Contract

After a detailed review of both approaches, let's create a comparison table (Table 2), where let's highlight the characteristic features of each approach.

Moving on to the conclusions about the comparison of the two approaches, it is possible to highlight that the key advantage of Proxy Pattern is state preservation. Therefore, if the logic update must preserve all internal contract variables, then Proxy Pattern is the best approach, as it does not require additional data management. If the logic contract is lightweight and does not contain critical state, or if there is no problem in storing it in a centralized location, then the Router approach becomes a more convenient and flexible solution, as it provides the ability to radically change the logic contract without the need to inherit the data structure of the old contract.

But if we combine both of these approaches into a hybrid architecture, we will get both the ability to update the logic contract in the proxy contract according to a strictly defined structure and the ability to radically update the contract without any restrictions at the same time.

Table 2. Comparative analysis of Proxy Pattern and Router Contract

Criterion	Proxy Pattern	Router Contract
Logic update mechanism	Changing the address of the logic contract in the proxy contract	Changing the address of contracts in the Router by a unique key
Saving the state	In the proxy contract	In each contract separately, or centralized using the Router Contract or another service contract
Implementation simplicity	High complexity: requires knowledge of delegatecall, EVM memory model etc.	Simple implementation: direct calls without delegation
Logic flexibility	Limited: new logic must have an identical storage layout	High: no dependence on memory structure
Risk of storage collision	High: without proper use of Unstructured Storage	None: contracts do not share memory space
Interaction address	Single: interaction with a proxy	Single: obtaining contract addresses via Router
Gas consumption for calls	Lower: using delegatecall	Higher: searching in mapping before the call
Logic update costs	Average: redeploy logic contract and update address in proxy	Average: redeploy logic contract and update address in Router
Vulnerabilities	Vulnerable to errors in delegatecall	Router is a potential point of failure; risk of incorrect routes
Support	High: OpenZeppelin Upgrades, Hardhat plugins	Limited: requires own implementation

Hybrid architecture that combines both Router Contract and Proxy Pattern approaches

In this approach, the Router Contract serves as a centralized dispatcher that stores the addresses of the proxy contracts for each contract of the system. Each proxy, in turn, delegates a call to the logic contract (Fig. 4). Such a multi-level structure creates a flexible update system in which the logic can be updated at two levels:

1. Partial update - changing a logic contract within a specific proxy contract without losing its state;

2. Full structural update - replacing the entire proxy contract together with the logic contract by changing the address in the Router Contract.

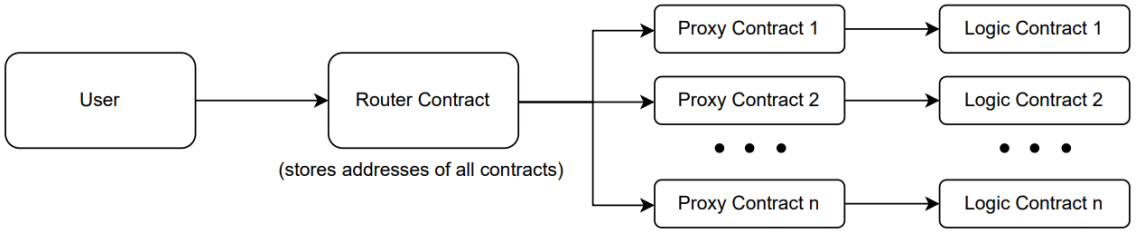

Fig. 4. General scheme of the hybrid approach

This approach allows you to choose the optimal update strategy depending on the scale of changes: small patches, function modification, or complete contract reconstruction.

The router contract remains an unchanged access point to the entire system, which contains all the addresses of the proxy contracts, which in turn delegate the execution of the functions to the logic contracts. Thus, the internal state of the contract is preserved, but it is also possible to completely update the logic along with the proxy contract.

The disadvantage of this approach is the complexity of implementation, since it is necessary to clearly define where the state is stored, which functions are delegated, and which level should be updated in each case.

Thus, the hybrid architecture combines two update models: delegation of logic without loss of state (via Proxy Pattern) and complete module replacement (via Router Contract). This provides high flexibility, but requires additional technical complexity and clear dependency management. This approach can be useful for projects that have a stable architecture and a regular need for updates.

Discussion

One of the fundamental requirements for modern software is the ability to be updated. In case of decentralized applications deployed in the blockchain environment, this requirement becomes especially critical due to the immutability of the code after its publication. Ensuring that the smart contract logic can be updated in a secure and controlled manner is a necessary condition to support continuous development, eliminate vulnerabilities, and adapt to changing user needs.

Today, the most widespread solution in the sphere of updatable smart contracts is the use of the Proxy Pattern. Its architecture allows to change the logic of contract without losing its state, which is ensured by delegating calls through the delegatecall instruction [11]. At the same time, the Proxy Pattern has a number of significant limitations, including the complexity of implementation and the need to inherit the memory structure of the previous contract, as there may be a risk of storage collision.

In this article, an alternative approach of dynamic routing as a Router Contract was proposed, which involves centralized management of logical contract addresses. This approach simplifies implementation and auditing, does not require logic delegation, and allows flexible changes to system components without strict requirements for memory structure. However, it does not automatically save the state when the logic is updated, which requires the implementation of a separate data storage mechanism.

In this paper, it was also proposed a hybrid solution that combines the advantages of both approaches. The architecture, in which the Router Contract stores the addresses of proxy contracts, which delegate logic to external contracts, allows for both partial logic updates without loss of state and complete contract logic replacement if necessary. This approach ensures high flexibility and scalability of the system, while maintaining a stable access point to its components.

As a perspective for further development, the proposed architecture may be enhanced by replacing the manually managed administrative role with an automated upgrade governance system based on on-chain voting. In this model, an additional contract for voting and a separate contract for vote counting would be introduced. Once the community reaches a predefined quorum, the Router Contract would automatically switch the contract logic according to the majority decision. This mechanism would increase the automation and decentralization of the upgrade process, improving trust and security in systems relying on collective governance.

Conclusion

This study addressed the challenge of improving smart contract upgradeability in decentralized environments by focusing on the limitations of the widely used Proxy Pattern. As a result, a Router Contract approach was developed and proposed as a more flexible and modular alternative. Both approaches were compared in terms of architecture and

gas efficiency, and a hybrid model was introduced that combines their advantages – retaining state through proxies while allowing dynamic logic replacement via routing. The proposed hybrid architecture meets the objectives of the study and offers an enhanced and scalable upgrade mechanism for evolving decentralized applications.

References

1. Ebrahimi A.M. et al. UPC sentinel: An accurate approach for detecting upgradeability proxy contracts in Ethereum. *Empirical Software Engineering*. 2025. Vol. 30, no. 61. DOI: 10.1007/s10664-024-10609-7.

2. Li X. et al. Characterizing Ethereum Upgradable Smart Contracts and Their Security Implications. *WWW '24: The ACM Web Conference 2024* : proceedings, Singapore, Singapore, 13–17 May 2024 / SIGWEB. 2024. P. 1847–1858. DOI: 10.1145/3589334.3645640.

3. Proxy Patterns. OpenZeppelin Blog. 2018. URL: https://blog.openzeppelin.com/proxy-patterns (date of access 26.04.2025).

4. Palladino S. The Parity Wallet Hack Explained. 2017. URL: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7 (date of access 26.04.2025).

5. Klinger P., Nguyen L., Bodendorf F. Upgradeability Concept for Collaborative Blockchain-Based Business Process Execution Framework. Lecture Notes in Computer Science. Vol. 12404. Blockchain – ICBC 2020. Third International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, September 18-20, 2020, Proceedings / ed. by Zh. Chen et al. Cham, 2020. P. 127–141. DOI: 10.1007/978-3-030-59638-5_9.

6. Al Amri Sh., Aniello L., Sassone V. A Review of Upgradeable Smart Contract Patterns Based on OpenZeppelin Technique. *The Journal of The British Blockchain Association*. 2023. 8 p. DOI: 10.31585/jbba-6-1-(3)2023.

7. Proxy Upgrade Pattern. OpenZeppelin Documentation. URL: https://docs.openzeppelin.com/upgrades-plugins/proxies (date of access 26.04.2025).

8. Nikhoriya R. Understanding Proxy Patterns in Solidity: A Developer's Guide. 2024. URL: https://medium.com/@riteshnikhoriya/understanding-proxy-patterns-in-solidity-a-developers-guide-019a5d4cbc71 (date of access 26.04.2025).

9. Ebrahimi A.M. et al. A large-scale exploratory study on the proxy pattern in Ethereum. *Empirical Software Engineering*. 2024. Vol. 29, no. 81. DOI: 10.1007/s10664-024-10485-1.

10. Benedetti A., Henry T., Tucci-Piergiovanni S. A Comparative Gas Cost Analysis of Proxy and Diamond Patterns in EVM Blockchains for Trusted Smart Contract Engineering. Lecture Notes in Computer Science. Vol. 14746. Financial Cryptography and Data Security. FC 2024 International Workshops. Voting, DeFI, WTSC, CoDecFin, Willemstad, Curaçao, March 4–8, 2024, Revised Selected Papers / ed. by J. Budurushi et al. Cham, 2025. P. 207–221. DOI: 10.1007/978-3-031-69231-4_14.

11. Salehi M., Clark J., Mannan M. Not so Immutable: Upgradeability of Smart Contracts on Ethereum. Lecture Notes in Computer Science. Vol. 13412. Financial Cryptography and Data Security. FC 2022 International Workshops. CoDecFin, DeFi, Voting, WTSC, Grenada, May 6, 2022, Revised Selected Papers / ed. by S. Matsuo et al. Cham, 2023. P. 539–554. DOI: 10.1007/978-3-031-32415-4_33.

Serebriakov R.O., Klymenko I.A.

ALTERNATIVE APPROACHES FOR SMART CONTRACT UPGRADEABILITY

Smart contracts upgradeability is a critical requirement for modern decentralized applications based on blockchain technology, but its implementation remains a technical challenge due to the immutable nature of smart contracts. The Proxy Pattern has become the most widely

used solution for implementing upgradability into smart contracts, but it comes with some limitations such as implementation complexity and strict memory layout inheritance. This paper introduces an alternative approach based on dynamic routing with Router Contract, which enables modular upgradeability through centralized address management, offering greater flexibility at the cost of requiring external state persistence. Furthermore, a hybrid architecture is proposed, combining both Proxy Pattern and Router Contract approaches to achieve dual-layer upgradeability – supporting both state-preserving updates and full module replacements. The paper offers a comprehensive evaluation of upgradeability strategies and proposes a versatile solution for evolving smart contract systems.

***Keywords**: smart contracts; upgradeability; Proxy Pattern; Router Contract; storage collision.*

Серебряков Р.О., Клименко І.А.

АЛЬТЕРНАТИВНІ ПІДХОДИ ДЛЯ ОНОВЛЮВАНОСТІ СМАРТ-КОНТРАКТІВ

Оновлюваність смарт-контрактів є критично важливою вимогою для сучасних децентралізованих застосунків заснованих на технології блокчейн, однак її реалізація залишається технічним викликом через незмінну природу смарт-контрактів. Шаблон Proxy Pattern став найбільш поширеним рішенням для впровадження оновлень в смарт-контрактах, проте він має певні обмеження, такі як складність реалізації та необхідність дотримання чіткої структури пам'яті. У даній статті запропоновано альтернативний підхід, заснований на динамічній маршрутизації за допомогою Router Contract, який забезпечує модульний підхід до оновлення шляхом централізованого управління адресами, пропонуючи більшу гнучкість за умови зовнішнього збереження стану. Крім того, запропоновано гібридну архітектуру, що поєднує підходи Proxy Pattern та Router Contract для досягнення дворівневої системи оновлень – як зі збереженням стану при оновленні логіки, так і з можливістю повної заміни модулів. У статті проведено комплексне порівняння стратегій оновлення та запропоновано універсальне рішення для розвитку систем смарт-контрактів.

***Ключові слова:** смарт-контракти; можливість оновлення; проксі патерн; роутер контракт; колізія пам'яті.*

.