Proceedings of the 19th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 249–252
DOI: 10.15439/2024F6780
ISSN 2300-5963 ACSIS, Vol. 39

Strategy Registry: an optimized implementation of the Strategy design pattern in solidity for the Ethereum Blockchain

Hamza Tamenaoul*, Mahmoud El Hamlaoui† and Mahmoud Nassar‡
RABAT IT CENTER
ENSIAS, Mohammed V University in Rabat
Rabat, Morocco
Email: *hamza_tamenaoul@um5.ac.ma, †mahmoud.elhamlaoui@um5.ac.ma, ‡mahmoud.nassar@um5.ac.ma

***Abstract*—The strategy design pattern is an essential behavioral design pattern. When developing software it is relied upon for the development of modular components. Given the constraints inherent to the inner workings of the Blockchain and features offered by Smart Contract languages, some of the design patterns commonly used in software systems development cannot be naively implemented. This paper explores a new pattern implementation of the mentioned pattern in a contract oriented language aimed at Smart-Contracts development, specifically the solidity language. The pattern's implementations discussed in this paper are intended to be deployed and run on the Ethereum Blockchain.**

***Index Terms*—Blockchain, Ethereum, Solidity, Smart Contract, Design Pattern, Strategy pattern, Gas, Optimization**

I. Introduction

THE STRATEGY design pattern [1] tries to solve a very specific behavioral problem. Component are developed to run a fixed set of algorithms. Given the static nature of compiled code, those algorithms could not be defined at a different moment than during compile time. However, in multiple cases, the algorithms to be executed are not necessarily known at that moment, the information would be rather uncovered only during runtime. Therefore a design pattern had to be defined to tackle this specific issue, which is the *Strategy Design Pattern*.

The strategy design pattern defines two elements: the context and the strategy [2]. The context is the component, containing and controlling the algorithms to be executed as well as handling and managing the set of attributes or parameters that impacts the execution flow of said algorithms. From a client perspective, the context is the entity that is called at compile time to run the algorithm.

Decoupling the context from the logic gives us the opportunity to have a modular logic, while enabling us to implement multiple algorithms with a simpler implementation and easier maintainability. This design is essential in a lot of contexts when developing software libraries or software systems. This loosely coupled structure makes it possible to create a context object instance without worrying whether or not the algorithm to be used when executing the logic is known at that exact moment. Moreover, it gives us the opportunity to switch strategies during runtime easily and in a transparent manner. The link between the strategy and the context is created when the algorithm to be used is known, sealing the relationship between the two objects and binding the strategy object lifecycle to the context object. While the strategy object can be discarded while the context is still used, for the context, discarding a strategy would not have any impact on its lifecycle. In fact, this pattern makes the lifecycle of the strategy object completely dependent on the one of the context. This relationship is illustrated in the Fig. 1

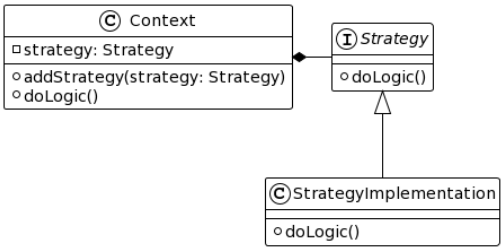

Fig. 1. UML class diagram of the implementation of the strategy design pattern

A. Pattern client implementation

Following the blueprint given by this implementation, the client code logic is required to create a new instance of a strategy every time a new context is instantiated. The following algorithm illustrate this logic.

```
Context context = new Context()
```

The context object is creating without any reference to the strategy object and therefore the algorithm to be used when running the logic.

```
Strategy strategy =
    new StrategyImplementation()
context.addStrategy(strategy)
```

When the strategy is known during runtime, the following algorithm would create the strategy object instance and bind it

to the context. From this moment the context instance has the information on what algorithm should be run forward. This part of the algorithm can be easily executed multiple time during the lifetime of the program with different strategies to accommodate any change of the logic during runtime.

```
context.doLogic()
```

Finally the logic can be called directly from the context, regardless of what logic resides underneath. This transparency when executing the logic, that makes the strategy design pattern an essential design pattern when developing any system.

In the context of non distributed application - *referred to as App* -, this client implementation would lead to the creation of new strategy instance whenever the second part of the algorithm is executed. The footprint of this approach lays mainly on the memory, requiring it to handle the whole lifecycle of the strategy object for every new instance, which is not generally an issue in modern languages.

In such standard applications the memory trade off is worth it, limited and minimal. More generally the memory footprint is negligible in front of the maintainability and the modularity gains, in distributed applications - *commonly referred to as dApps* -, we cannot neglect the memory footprint of this client implementation for cost efficiency.

II. Cost analysis of the strategy pattern implementation in a distributed application

A. The optimization requirements of Blockchain specific code

Distributed applications rely on the Blockchain technology to be deployed and run [4]. They use Smart-contracts to define their business logic, a distributed logic. By design, the Blockchain is a technology that distributes the execution of the logic on all of its users, referred to as nodes. This structure makes also the cost of executing some code on the infrastructure cost on all nodes, leading to a higher resource cost usage cost for the user of some dApp.

This cost is very high and cannot be negligible and therefore kept to a minimum. When running Apps on a non Blockchain environment, the resource cost is the cost of the computing time of the machine that executed the code. On a Blockchain setting, that cost is multiplied by the number of nodes on the Blockchain, since they all need to execute the code for a dApp to work. Those nodes maintain and run the infrastructure of the Blockchain, therefore the Blockchain put a price on every action on the Blockchain to compensate the nodes for their work.

To calculate this compensation, the concept of **Gas** [4] was introduced to evaluate the price equivalent of every action on its infrastructure, the inherent cost of handling a transaction or running an algorithm on the Blockchain network. Gas is a notion that is specific to the Ethereum Virtual Machine (EVM) and similar Blockchains, it represents the computational effort needed to execute a transaction on the Blockchain. When multiplied by the price of a unit of gas we can calculate the price of said transaction. Therefore gas can be used as a measure of the cost of a transaction. To be able to run code on the Blockchain, the caller is required to provide sufficient Gas for its request, or it will not be processed by the network. This requirement forces dApps designers to build and design specifically optimized code for Blockchain applications, code that would require the least amount of memory and processing resources possible to make the cost of using the dApp minimal and acceptable for users.

B. The limitations of the standard strategy pattern implementation

The standard approach to implement the strategy design pattern falls short in memory optimization aspects. In non Blockchain based languages, instantiating new objects is equivalent to allocating a new memory space on the machine running the program. In the context of the Blockchain, creating a new instance of an object, means the creation and deployment of a new Smart-contract, which means allocating resources on all nodes of the infrastructure which is one of the heaviest operations in terms of memory allocation and instructions to execute on the Virtual Machine. This makes the operation require a lot of gas which in turn translates to an extremely high cost on the burden of users. Therefore this approach is not optimal.

III. Name Registry design pattern based optimization

A. Name Registry pattern

On the Ethereum Blockchain, a Smart-contract needs to be deployed first before being used. Once a Smart-contract is deployed, the only way interact with the Smart-contract is by calling through its address on the Blockchain. However loosing the address of a Smart-contract means more or less loosing the ability interact with it. To solve this issue and others related to it the **Name Registry pattern** [5] was introduced.

The Name Registry pattern as its name suggest centers on a registry as the main piece, an address registry. It stores the different addresses of the Smart-Contracts created. It solves the issue of dealing with storing the physical address of Smart-contracts by taking care of this part, and providing keys to retrieve easily the addresses that would be required. The registry pattern can be simplified as a *mapping* from and already defined key - that can be mutable - to a Smart-contract address. While adding addresses inside of the registry has a cost, it is much more cheaper than creating a new Smart-Contract every time on is needed, specially when those Contracts are stateless, but depending on the implementation, the owner of the dApp can implement it a way that the burden of the cost would not fall on the user but on the owner of the Smart-Contract.

A basic key implemented as a string identifier, as shown in Fig. 2 could be used for by mapping in the Name Registry pattern. The key could be much sophisticated depending on the requirements of the project. The most common addition of to the key is the Smart-contract version, to handle versioning which is not native to the Ethereum Blockchain. This part of

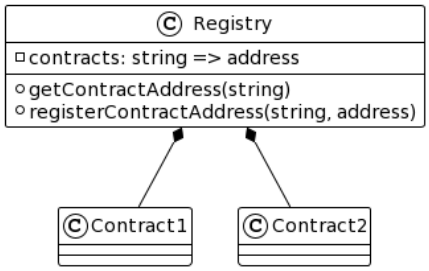

Fig. 2. UML diagram of the Name Regitry pattern

the pattern will not be discussed in this paper and falls out of the scope of the paper.

B. Contract independent strategy pattern implementation

To solve the overhead issue of having to deploy a new Smart-contract every time a new one defining a strategy is deployed, both the Strategy pattern could be merged with the Name Registry pattern to provide the optimization needed for the constraints of the Blockchain.

This paper will refer to the resulting pattern as the **Strategy Registry**. The Strategy Registry optimizes the Strategy pattern by removing the direct link between the context and the strategy. Instead, the context is linked to a Name Registry that contains the different strategies deployed. Strategies are registered into the registry making them context independent and unbinding their lifecycles from the context's. While the registry would contain the strategies that have been deployed by the system's development team, the nature of Blockchain development can make the addition of new strategies more independent.

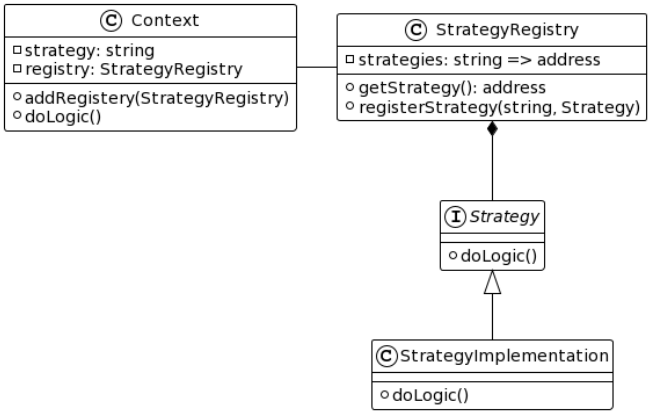

Fig. 3. UML class diagram of the Strategy Registry pattern

This optimization operates on multiple levels:

1) Gas optimization: With the implementation of the strategy design pattern, the number of the strategies that exists in the execution context can reach the total number of strategies defined in the system. Therefore the total number of Strategies that would coexist on the Blockchain is theoretically unlimited, which is an important optimization issue. This would entail that for every execution an enormous amount of gas would be spent every time to repeat the same operation, with transations containing duplicated amount of information that needs to be stored.

Using the Registry strategy pattern caps the number of strategies that could be deployed to the number of strategy implementations that were developed. Having a fixed number of contract to deploy simplifies the cost calculation and reduces enormously the overhead required during every execution. In fine, this would lead to a reduces and acceptable gas price for the pattern.

2) Space optimization: One of the main constraints of the Blockchain is the limited size of the available space. Distributed application designer seek to build space efficient Smart-contract, like during the first days of computing. Using the strategy design pattern breaks this rule. As mentioned previously, deploying strategies with the context only contributes to enlarging the memory size required by the context. If we take into account that the number of strategies is non capped, this means that dApp relying on such architecture could take an unlimited amount of space on the Blockchain, which is far from being ideal, nor acceptable in terms of design or cost.

The Strategy Registry solves perfectly this issues. Capping the number of Smart-contracts that could be possibly deployed means a finite amount of space would be needed for the strategies regardless of the number of executions. Moreover decoupling the Strategy from the context, leads to having lighter Smart-contracts defining the context to deploy, optimizing even further the usage of space.

3) Versioning of strategies: One of the main benefits of using Strategies is the ability to update the logic of some program without impacting the code of the context. This ability can be seen as lost when talking about deploying the strategies beforehand. While at fist glance this might seem true, in reality using the registry enables this feature in a more straight forward manner.

One of the main benefits of the Name Registry pattern in the Ethereum Blockchain is its ability to introduce Smart-contract versioning in a simple and easy way. This capability could be also used to version strategies or even decommission or disable them. In the naive implementation, changing or improving a strategy would have required a double cost. The first and most expensive one, is the redeployment cost of new Smart-contracts representing the strategies as many times as the old versions were deployed. The second cost is related to updating the strategies in the different Smart-contract already in production.

Using the Strategy Registry, the cost would be reduced to one of deploying the new Strategy Smart-contract and registering it in the registry alone. No other data would be changed in any other Smart-contract; the context would still use the same key - that do not have to be changed - to call the strategy, the registry makes sure that the correct Smart-contract will be called, since it holds the information about the statuses of the different Strategies.

IV. Conclusion

The current Blockchain technology comes with new set of constraints, constraints that makes following many of the software principles already used and theorized hard to follow. One of important principles are design patterns such the *Strategy Design Pattern*. The *Strategy Registry Design Pattern* makes such transition. It solves the problem *Strategy Design Pattern* intents solving, while providing a solution that fits the challenges faced during Ethereum Blockchain development.

While using the Strategy Registry design pattern has obvious advantages in terms of production costs for both the owner of the dApp and its user, the performance of such pattern on the Blockchain should be assessed depending on the usage. One fits all solutions rarely exist in the software world, and even more so when working with Ethereum Smart-Contracts.

References

[1] E. Freeman and E. Robson, Head first design patterns: a brain-friendly guide, Second release. in A brain-friendly guide. Beijing Boston Famham Sebastopol Tokyo: O'Reilly, 2014.

[2] Refactoring Guru, "Strategy," Refactoring.guru, 2014. https://refactoring.guru/design-patterns/strategy

[3] "Solidity — Solidity 0.8.26 documentation," docs.soliditylang.org. https://docs.soliditylang.org/en/v0.8.26/ (accessed May 28, 2024).

[4] B. Vitalik, "Ethereum white paper: A next-generation smart contract and decentralized application platform." 2014. [Online]. Available: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_- _Buterin_2014.pdf

[5] "Contract Registry," Blockchain Patterns. https://research.csiro.au/blockchainpatterns/general-patterns/contract-structural-patterns/contract-registry/ (accessed May 28, 2024).