Aspect-Oriented Smart Contracts

Saifur Rahman Bhuiyan

TU Dresden

May 30, 2023

Table of Contents

Introduction

Smart Contracts

- Smart contracts are self-executing contracts that are stored on a Blockchain.
- These are consists of pre-determined, mutually decided conditions or parameters which can trigger deterministic outcome when certain conditions are met.
- Smart contracts automate contract execution, providing immediate assurance of outcomes to all parties involved, without intermediaries.
- Smart contracts have diverse real-world applications, primarily in the financial sector for activities such as investing, lending, borrowing, and insurance. However, their potential extends to other domains including gaming, healthcare, digital identity, and more.

Smart Contracts

Challenges in Writing Smart Contract

- **Rigid Behavior**: Once a smart contract is deployed on a blockchain, it cannot be changed because they are immutable and tamper-proof. This can be a problem if there are any errors in the code or if the contract needs to be updated to reflect changes in the law or the business environment.
- **Modularity**: Smart contracts are typically written as a single, monolithic piece of code. This makes them difficult to understand and maintain.[1][2]
- **Extensibility**: It can be difficult to add new features or functionality to existing smart contracts.

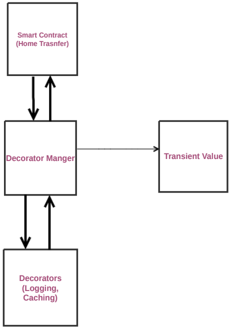
Solution

Decorator Layer

- The decorator[3] enhances an object's functionality without altering its structure
- The decorator can be used to modularize smart contracts by breaking them down into smaller, more manageable parts. Each part can then be implemented as a separate decorator. This makes it easier to understand and maintain the smart contract code
- The decorator can also be used to make smart contracts more extensible. New functionality can be added to a smart contract by simply adding a new decorator. This makes it easy to add new features or to change the behavior of the smart contract without having to modify the original code.

System Architecture

High Level View



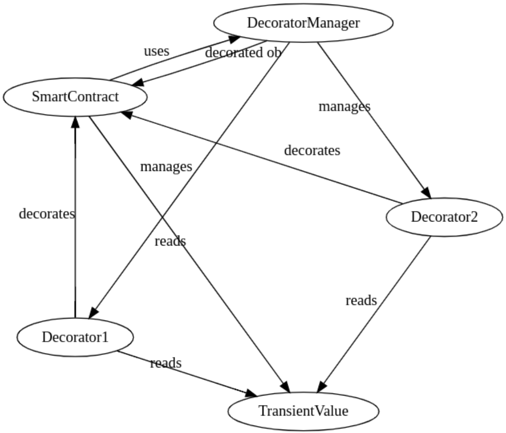
Decorator Manager

- The Decorator Manager is a central component of our dynamic decoration architecture. It manages the application of decorators to our smart contracts and provides a central point of control for dynamic behavior modification.
- It holds a registry of available decorators and controls when and how these decorators are applied to the smart contracts. It can apply a single decorator or a chain of decorators depending on the context and requirements.
- With the Decorator Manager in place, adding new behaviors to our smart contracts is as easy as writing a new decorator and registering it. No modification to the existing codebase is required, enhancing modularity and ease of maintenance.

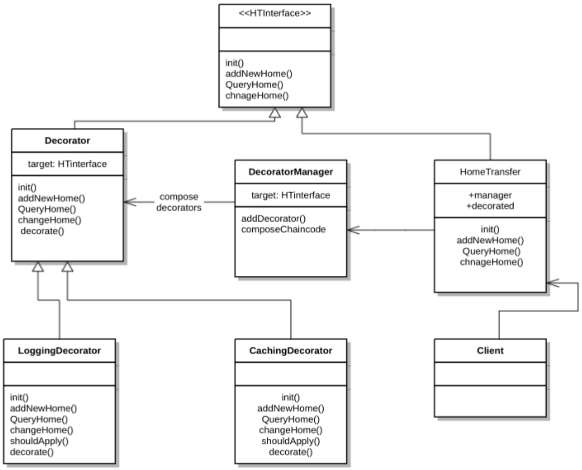
Transient Value

- Transient Values in Hyperledger Fabric are a special type of data that do not get recorded on the ledger but can be used to pass non-persistent data within transactions.
- We use Transient Values as a means to control the dynamic application of decorators. They can be used to pass decoration requirements along with the transactions.
- Transient Values allow us to adjust the behavior of our smart contracts on-the-fly, without changing the recorded data in the ledger. This gives us a powerful tool to adapt to changing needs without compromising the integrity of our stored data.

Conceptual Diagram



UML Diagram


<<HTInterface>>
init()
addNewHome()
QueryHome()
chnageHome()
Decorator
target: HTinterface
init()
addNewHome()
QueryHome()
changeHome()
decorate()
compose decorators
DecoratorManager
target: HTinterface
addDecorator()
composeChaincode
HomeTransfer
+manager
+decorated
init()
addNewHome()
QueryHome()
chnageHome()
LoggingDecorator
init()
addNewHome()
QueryHome()
changeHome()
shouldApply()
decorate()
CachingDecorator
init()
addNewHome()
QueryHome()
changeHome()
shouldApply()
decorate()
Client

Challenges

- **High Maintenance Cost:** Each Decorator conforms to the interface of the object it decorates. If the interface changes, all decorators need to be updated, leading to a high maintenance cost.
- **Difficulty in Removing Added Behavior:** Once a decorator is applied, removing its specific behavior at runtime is challenging.

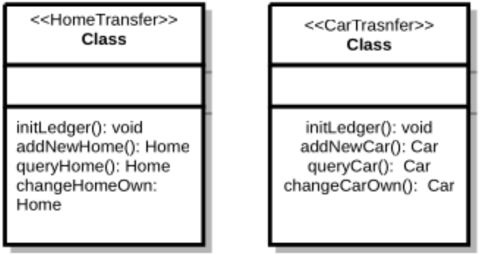
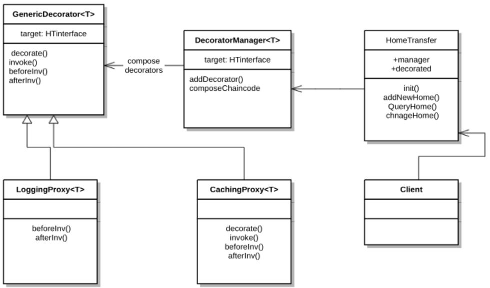


Figure: Multiple Contracts

Modifed Approach: Decorator with Dynamic Proxy

- **Simplified Code:** Don't need to create a separate concrete decorator class for every combination of interface, methods.
- **Reduced Maintenance Cost:** The Proxy is flexible regarding changes in the real object's interface.
- **Dynamic Method Interception:** Dynamic Proxies enable method interception, allowing pre-processing and post-processing of method calls.

Architecture



Architecture

- First method take HomeTransferInterface instance, assigns it to a local variable, and returns decorated object
- Second method takes a generic object target, stores it in a local variable, and then creates a new proxy instance

```
public HomeTransferInterface decorate(HomeTransferInterface chaincodeInterface) {
        this.chaincodeInterface = chaincodeInterface;
        return this;
    }

public T decorate(T target) {
        this.target = target;
        return (T) Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                this);
    }
```

Summary

Summary

- The proposed architecture makes a unique use of dynamic proxies combined with decorators to create an extensible and flexible system that can alter its behavior at runtime based on transient metadata
- This architecture is flexible in terms of handling different interface which is very powerful and reduces huge amount of boilerplate code
- Usage of reflection can lead to performance overhead and security issues

Thank You!

References I

[1] Hung, Chien-Che, Kung Chen, and Chun-Feng Liao. *Modularizing cross-cutting concerns with aspect-oriented extensions for Solidity*. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 176–181, 2019.

[2] Azzopardi, S., Ellul, J., Falzon, R., Pace, G.J. *AspectSol: A Solidity Aspect-Oriented Programming Tool with Applications in Runtime Verification*. In: Proceedings of the 22nd International Conference on Runtime Verification (RV 2022), Tbilisi, Georgia, September 28-30, 2022, pp. 243-252. DOI: 10.1007/978-3-031-17196-3_13.

[3] Rosenmüller, M., Siegmund, N., Apel, S., et al. (2011). *Flexible feature binding in software product lines*. *Autom Softw Eng, 18*(2), 163–197. DOI: 10.1007/s10515-011-0080-5.