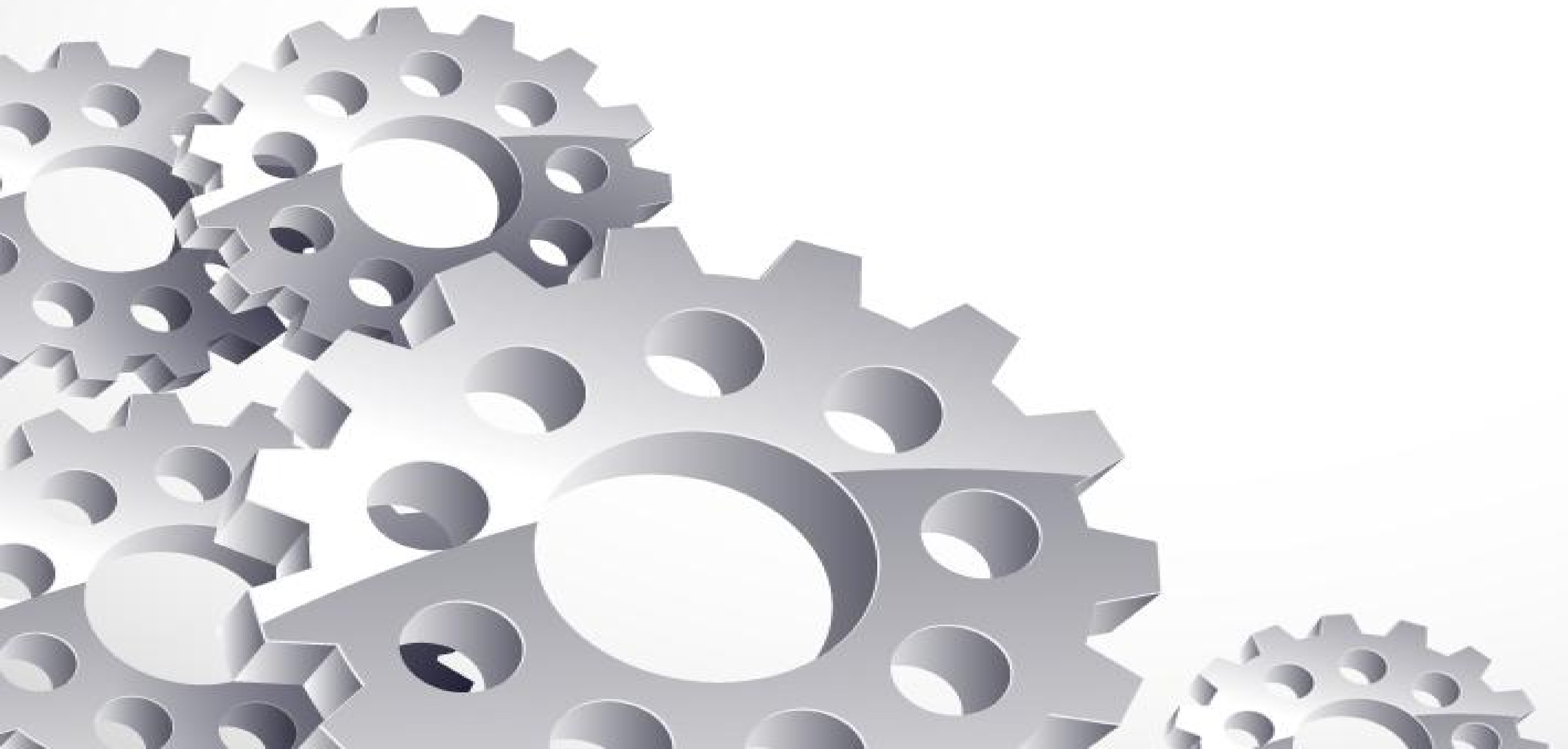
# Upgradability Patterns

for Ethereum Smart Contracts



## Upgradability Patterns

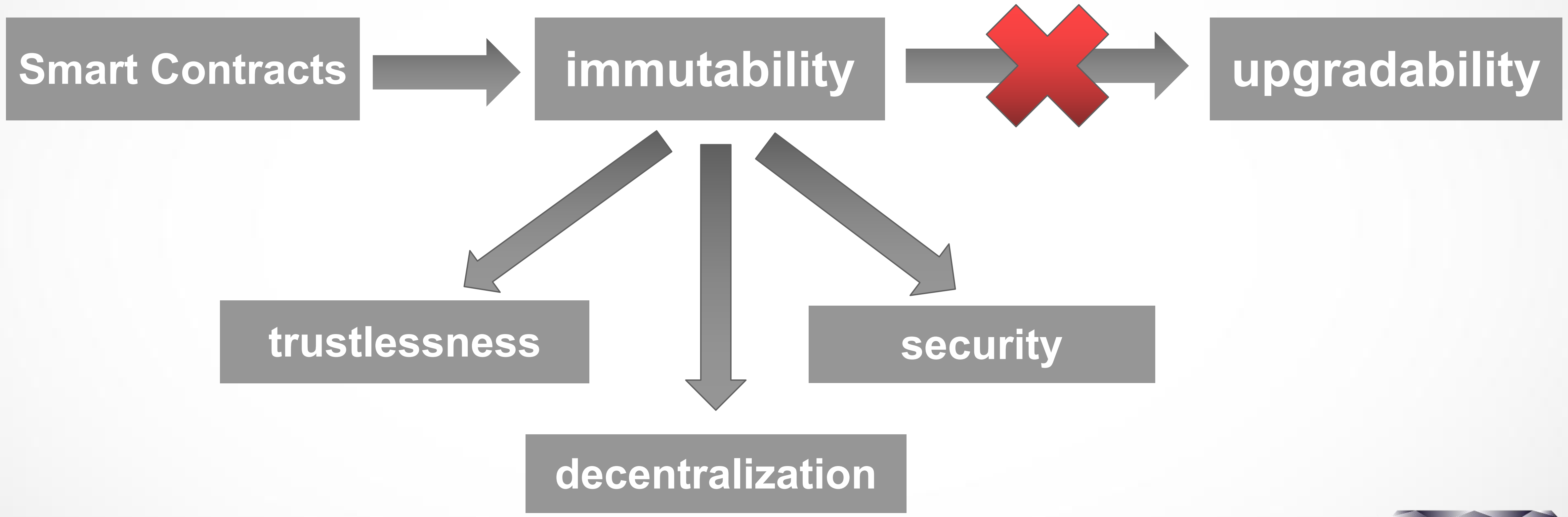
1. **Proxy pattern**
2. **Diamond pattern**

## Upgradability Patterns

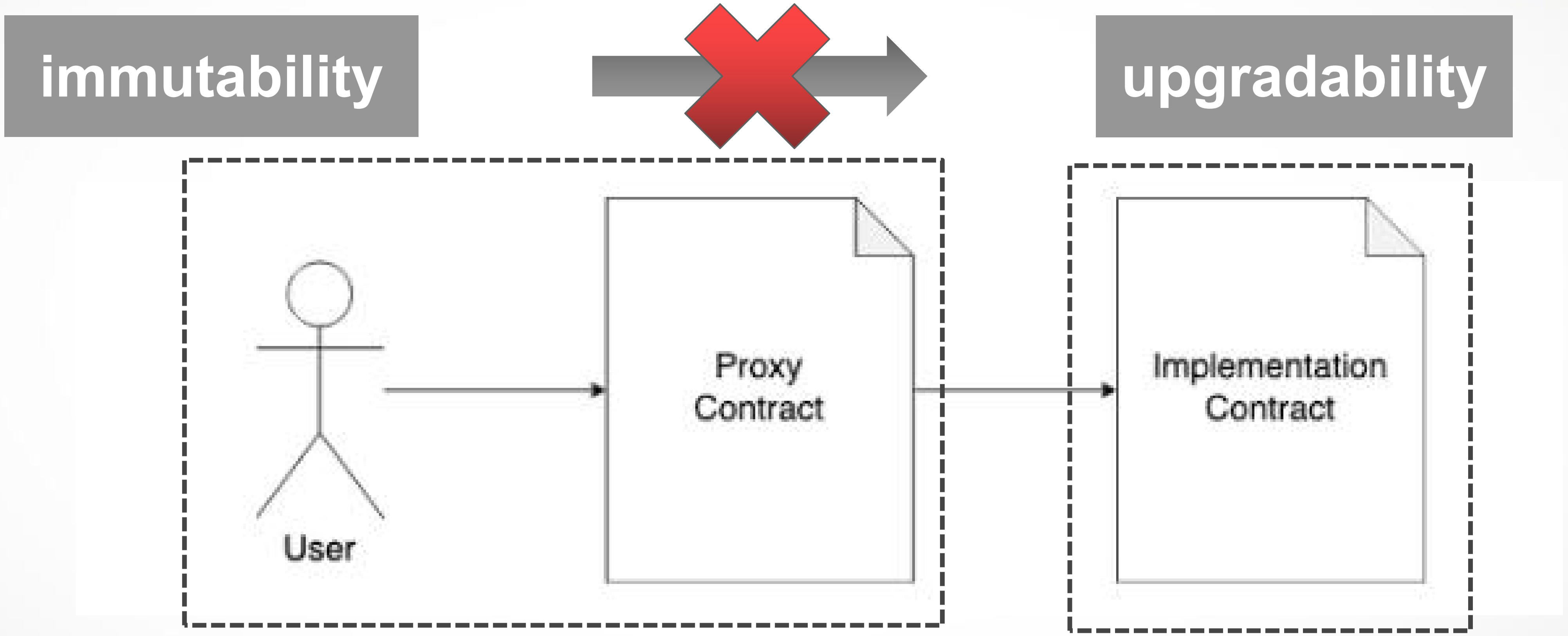
The diamond pattern can be considered an improvement on the proxy pattern. Diamond patterns differ from proxy patterns because the diamond proxy contract can delegates function calls to more than one logic contract.

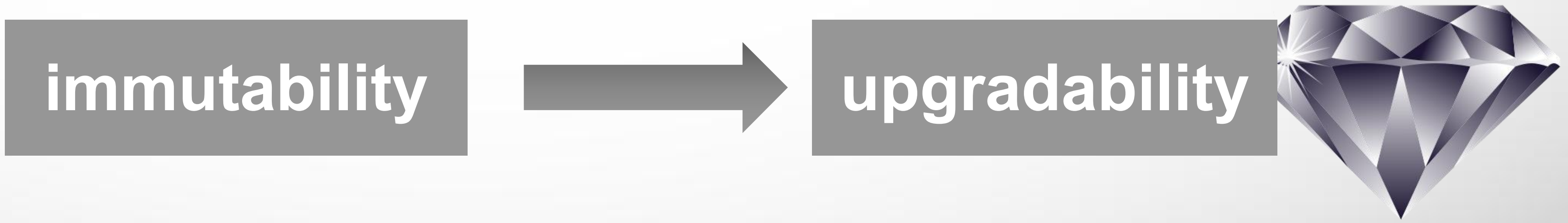
Therefore, we are going to focus on diamond contracts - the most interesting upgradability pattern.

# WHAT IS A SMART CONTRACT UPGRADE?



## WHAT IS A SMART CONTRACT UPGRADE?





## HOW IS IT HELPFUL?

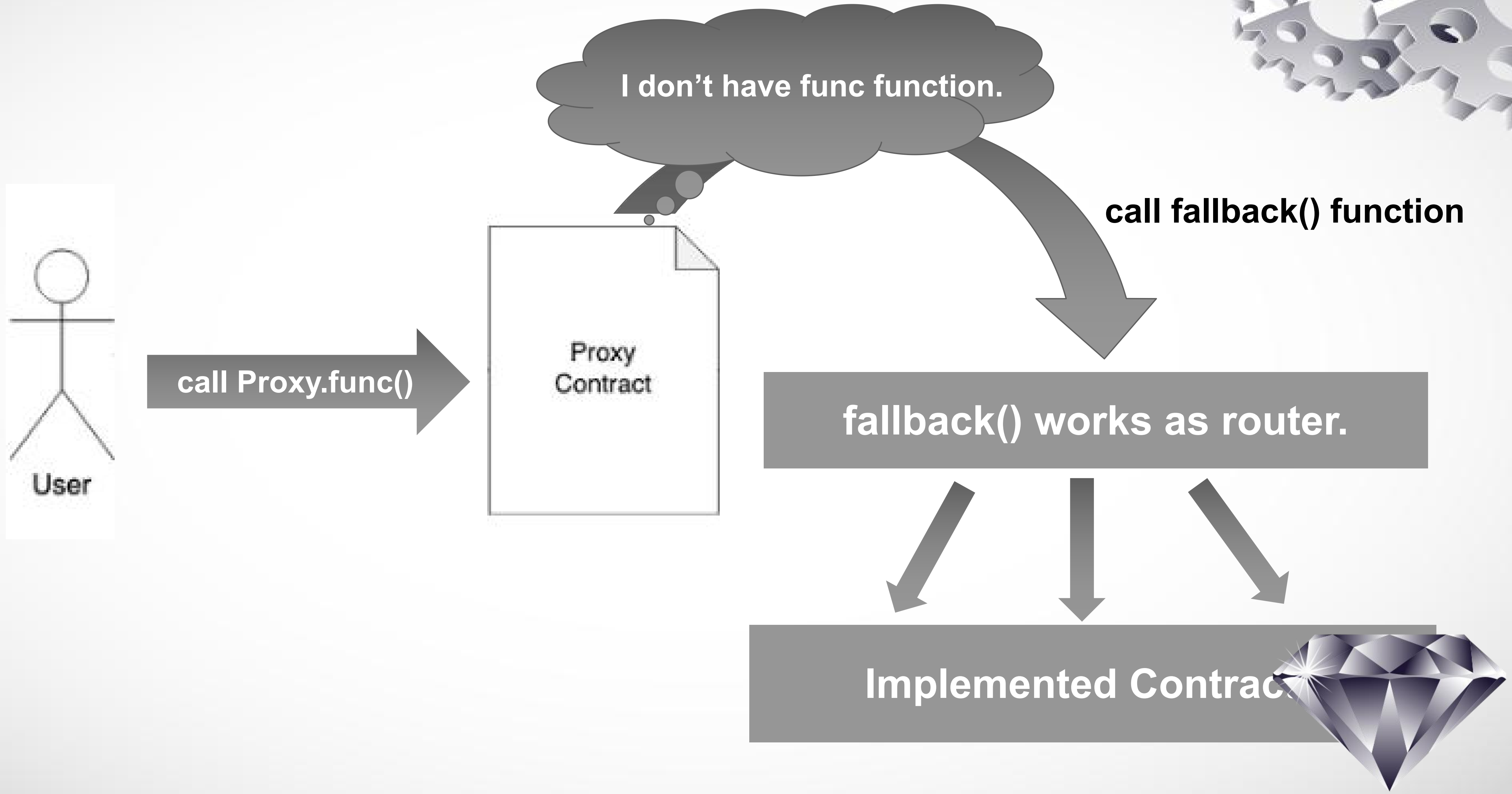
**Write smart contracts with virtually no size limit**

- **A single contract can’t exceed the 24KB maximum contract size.**

**A single address for unlimited contract functionality**

- **No need to redeploy existing functionality.**
- **Part of a contract can be added/replaced/removed while leaving**

## Technical Background



## Technical Background

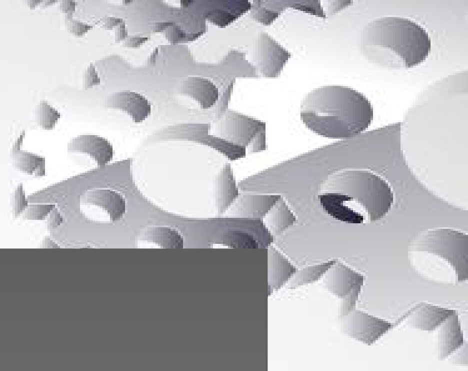
How to call another contract function as its internal function

Solution: delegatecall

It doesn't affect msg.sender, msg.value.

## Technical Background

```
fallback() external payable {
  address facet = selectorTofacet[msg.sig];
  require(facet != address(0));
  assembly {
    calldatacopy(0, 0, calldatasize())
    let result := delegatecall(gas(), facet, 0, calldatasize(), 0, 0)
    returndatacopy(0, 0, returndatasize())
    switch result
      case 0 {revert(0, returndatasize())}
      default {return (0, returndatasize())}
  }
}
```



## Facets, State Variables and Diamond Storage

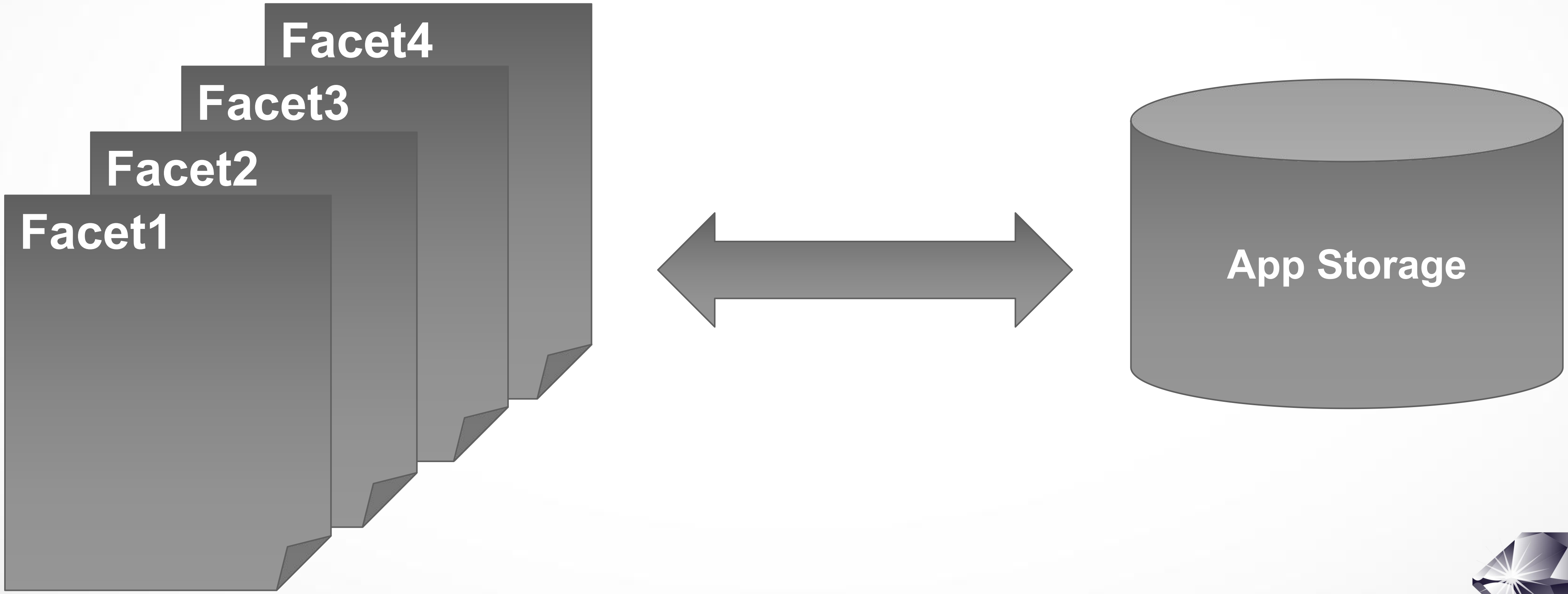
```
library LibA {
  struct DiamondStorage {
    address owner;
    bytes32 dataA;
  }
  function diamondStorage() returns (DiamondStorage storage ds) {
    bytes32 storagePosition =
           keccak256("diamond.storage.LibA");
    assembly {ds.slot := storagePosition}
  }
}
```

**The storage locations of state variables do not conflict.**

## Facets, State Variables and Diamond Storage

```
contract FacetA {

  function setDataA(bytes32 _dataA) external {
    LibA.DiamondStorage storage ds = LibA.diamondStorage();
    require(ds.owner == msg.sender, "Must be owner.");
    ds.dataA = _dataA;
  }

  function getDataA() external view returns (bytes32) {
    return LibA.diamondStorage().dataA;
  }
}
```

# AppStorage



## AppStorage

```
import "./AppStorage.sol"

contract StakingFacet {
  AppStorage internal s;

  function myFacetFunction(uint256 _nextVar) external {
    s.total = s.firstVar + _nextVar;
  }
}
```

AppStorage is always located at position 0 in contract storag

## Features

Diamonds Can Use Other Contract Storage Strategies.

Facets Can Share State and Functionality.

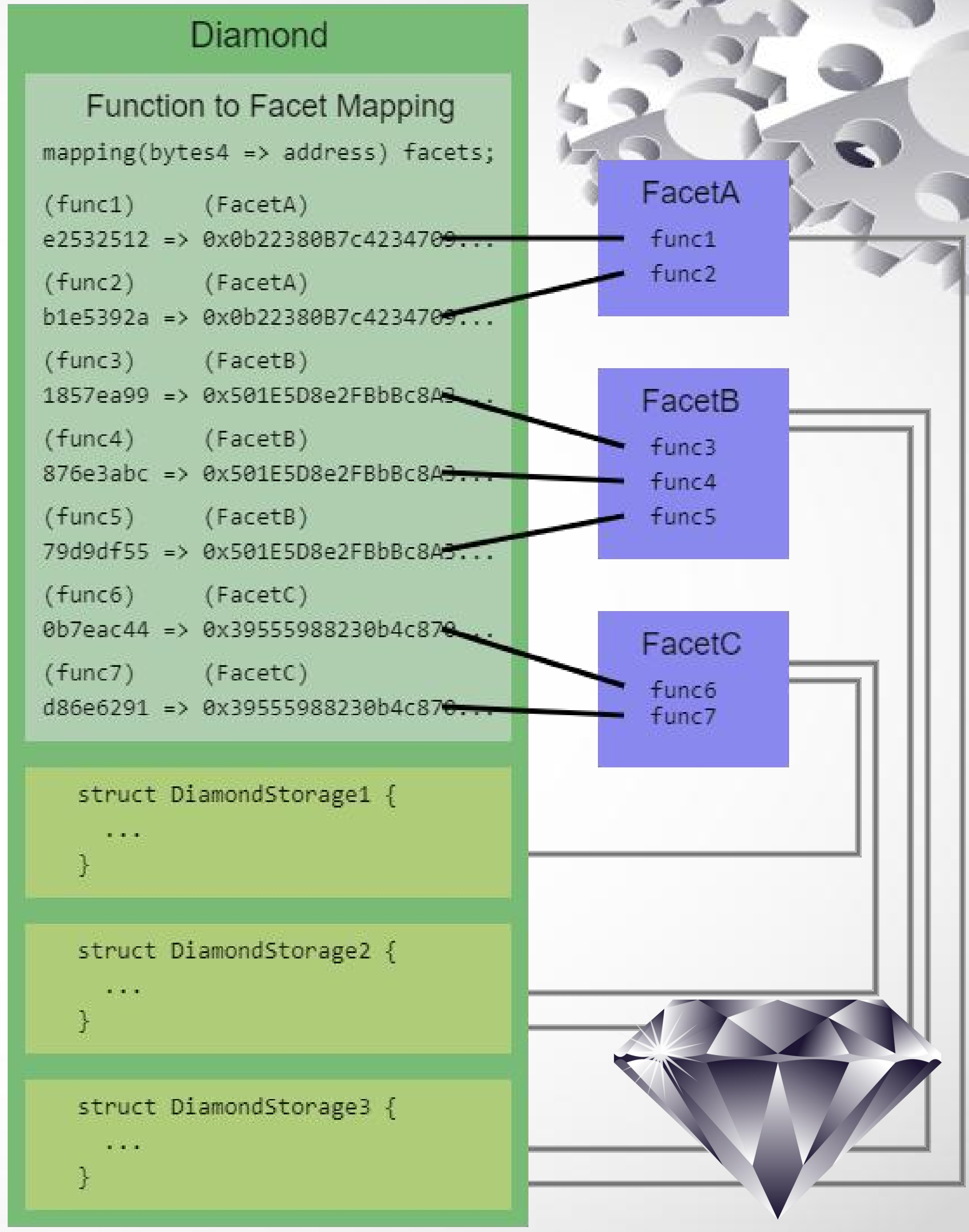
Facets are Reusable and Composable.

## Confused?

## Let’s take it easy.

**Diagrams of Diamond Contract**

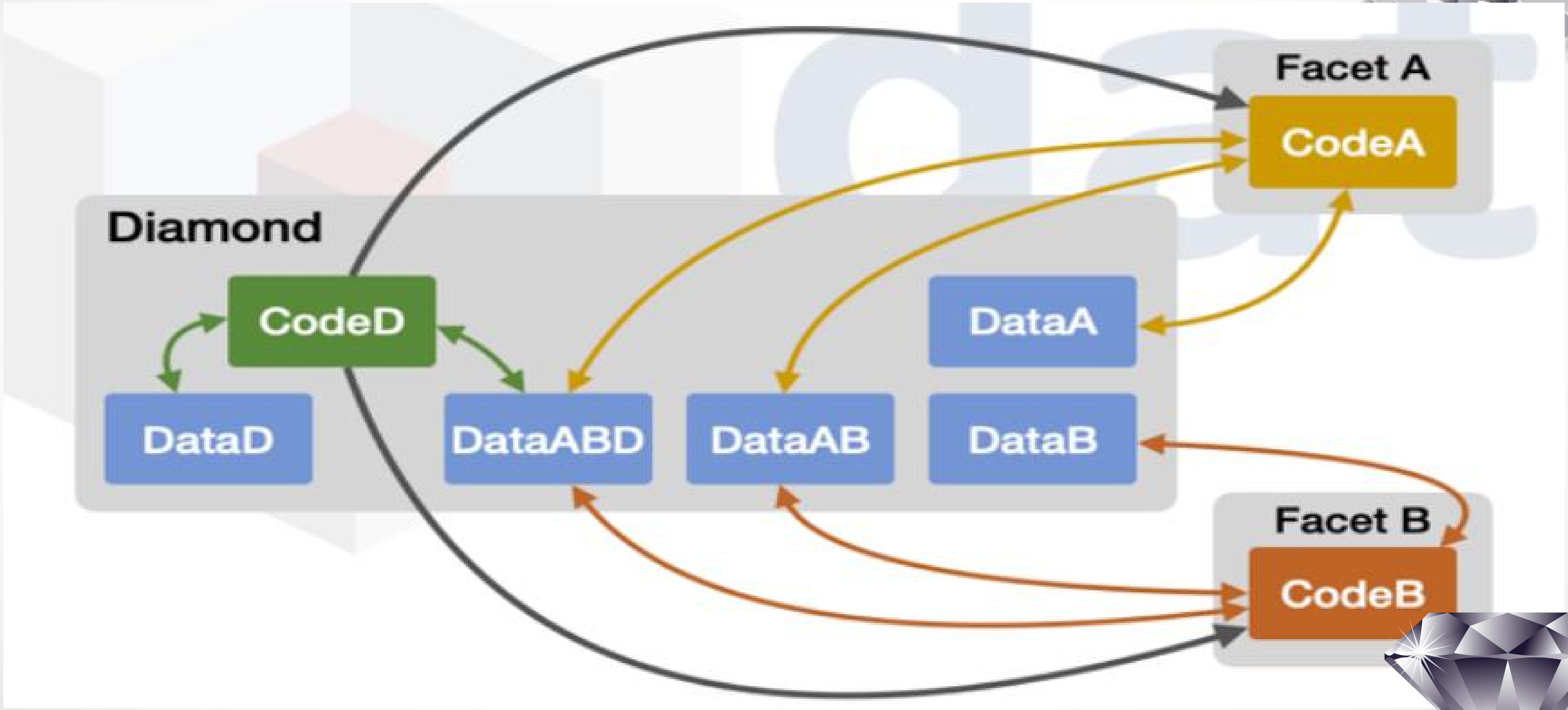
**The Proxy contract contains a dictionary.**



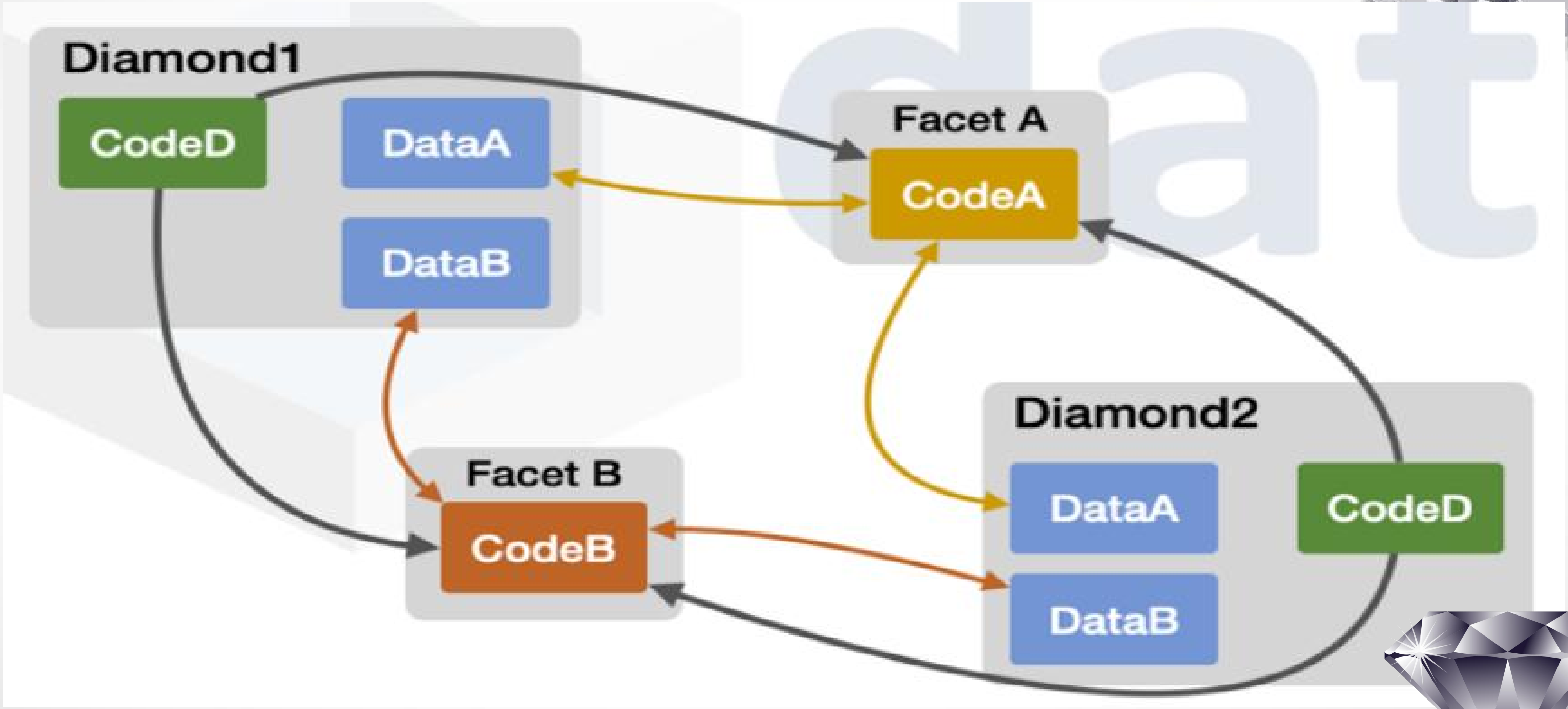
```
mapping(bytes4 => address) facets;

(func1)     (FacetA)
e2532512 => 0x0b22380B7c423470...
(func2)     (FacetA)
b1e5392a => 0x0b22380B7c423470...
(func3)     (FacetB)
1857ea99 => 0x501E5D8e2FBbBc8A3...
(func4)     (FacetB)
876e3abc => 0x501E5D8e2FBbBc8A3...
(func5)     (FacetB)
79d9df55 => 0x501E5D8e2FBbBc8A3...
(func6)     (FacetC)
0b7eac44 => 0x39555988230b4c870...
(func7)     (FacetC)
d86e6291 => 0x39555988230b4c870...
```

```
struct DiamondStorage1 {
  ...
}

struct DiamondStorage2 {
  ...
}

struct DiamondStorage3 {
  ...
}
```

## Sample Diamond Contract Diagram 1



# Sample Diamond Contract Diagram 2



# Different Kinds of Diamonds

**Upgradeable Diamond**

↳ **Can upgrade any time.**

**Finished Diamond**

↳ **The number of upgrades are limited.**

**Single Cut Diamond**

↳ **Not upgradable, Why we need this?**

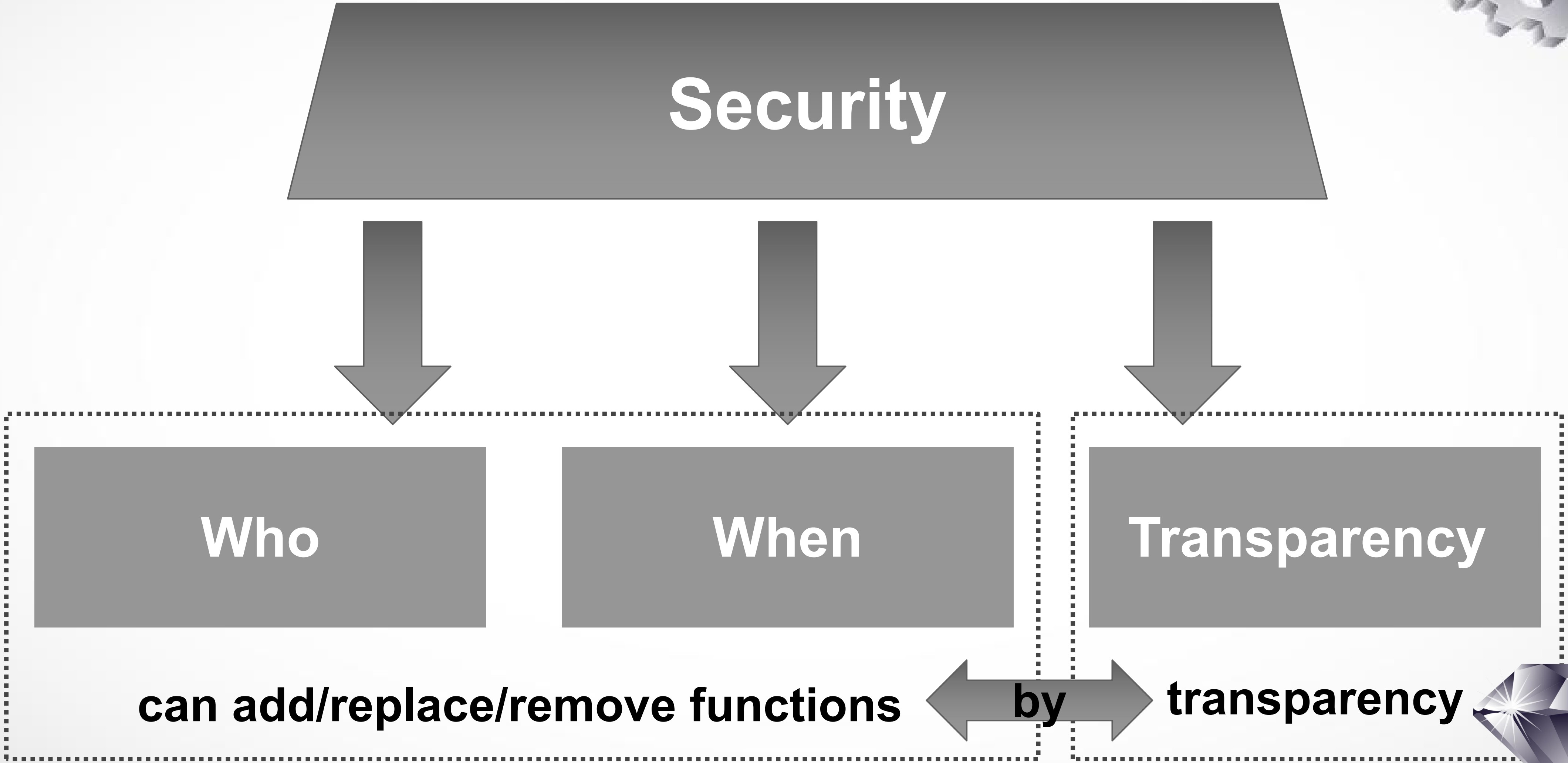
## Cons of Single Cut Diamond

To solve maximum contract size limit

To modularize large project

To deploy on mainnet after finished working on testnet

## Security Considerations



# Security Considerations

## Several ways who can be limited

1. Only allow a trusted individual or organization to make diamond upgrades.
2. Only allow a distributed autonomous organization to make diamond upgrades.
3. Only allow multi-signature upgrades.
4. Only allow the end user who owns his own diamond to make upgrades.
5. Don’t allow anybody to make upgrades by making a single cut di

## Security Considerations

### Several ways when can be limited

1. Make a single cut diamond for main network release.
2. Use an upgradeable diamond until it is certain that no new features are needed and then make it a finished diamond by removing the ability to add/replace/remove functions.
3. Only in certain periods of time, the diamond can be upgraded.

## Security Considerations

### Several ways to provide transparency

1. **Publish and make available verified source code used by diamonds and facets.**
2. **Provide documentation for diamonds, facets, upgrade plans, and results of upgrades.**
3. **Provide tools and/or user interfaces that make your diamonds more visible and understandable.**

# Learning & References

## Diamond Articles

1. Introduction to EIP-2535 Diamonds
2. Awesome Diamonds
3. How to Share Functions Between Facets of a Diamond
4. Smart Contract Security Audits for EIP-2535 Diamonds Implementations
5. Why Ethereum Diamonds Need A Standard
6. Ethereum's Maximum Contract Size Limit is Solved with EIP-2535 Diamonds
7. Understanding Diamonds

## Learning & References

### Diamond Storage Articles

1. How Diamond Storage Works
2. AppStorage Pattern for State Variables in Solidity
3. New Storage Layout For Proxy Contracts and Diamonds
4. Solidity Libraries Can’t Have State Variables – Oh Yes They Can!
5. Rules for Diamond Upgrades
6. Example of Adding New State Variables in Diamond Upgrade
7. Constructor Functions Don’t Work in Facets

# Learning & References

## Implementations

1. Diamond reference implementations
2. GHST Staking
3. pie-dao / ExperiPie
4. Nayms Contracts
5. Aavegotchi Diamond

# Thanks for your time.

*Oliver Chang*