Upgrading Smart Contracts

The great advantage to smart contract is that they're immutable, no one can hack them or change their terms once they are deployed

The great drawback to smart contracts is that they're immutable, you can't fix them once they're deployed.

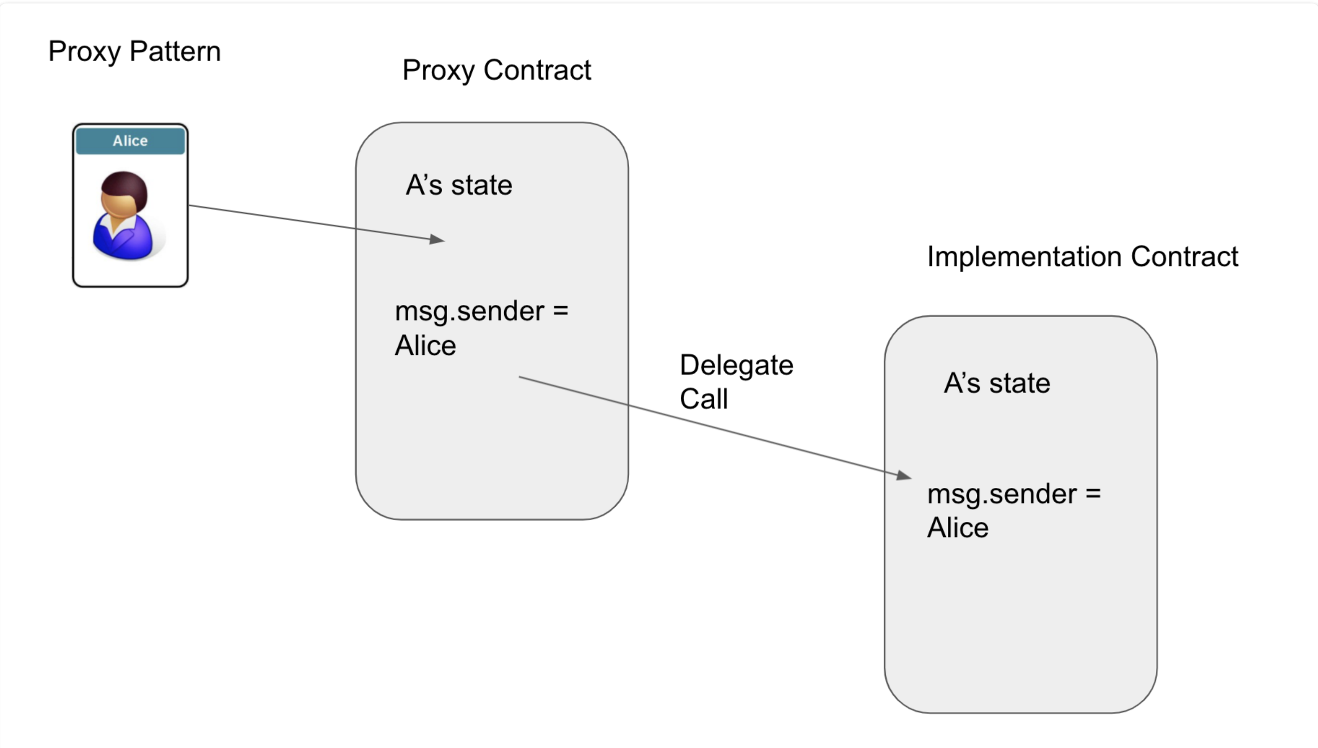
The problems we need to solve are

1. How to change the functionality in the contract
2. How to migrate data if necessary

We will look at some of the patterns used to allow upgradability
I have taken examples from this guide, the guide applies to truffle or hardhat.
See State of upgrades

Background - Proxy patterns

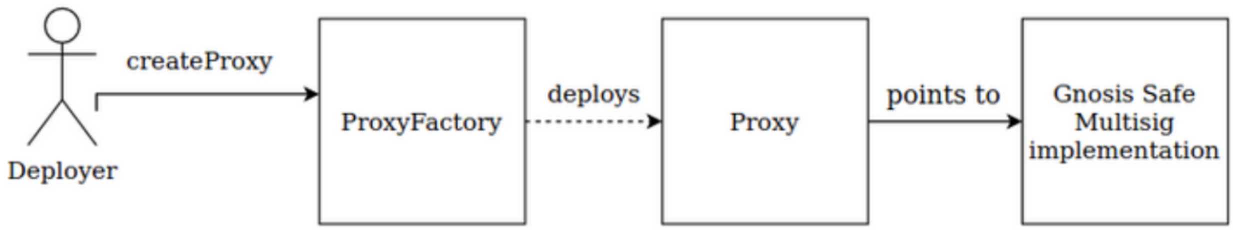
See EIP 897



We have a proxy contract and an implementation contract
Users always interact with the Proxy contract and need not be aware of the implementation contract.
It is also possible for multiple proxy contracts to use the same implementation contract (This can be a way of deploying multiple instances of a contract cheaply, see EIP1167).

Open Zeppelin Clones Library

We are cheaply creating a proxy for another contract



Deploying a Gnosis Safe Multisig wallet with the ProxyFactory contract

This function uses the create2 opcode and a `salt` to deterministically deploy the clone. Using the same `implementation` and `salt` multiple time will revert, since the clones cannot be deployed twice at the same address.

```
function cloneDeterministic(address implementation, bytes32 salt) internal
returns (address instance) {

        assembly {
                let ptr := mload(0x40)
                mstore(ptr,
0x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
                mstore(add(ptr, 0x14), shl(0x60, implementation))
                mstore(add(ptr, 0x28),
0x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
                instance := create2(0, ptr, 0x37, salt)
        }
        require(instance != address(0), "ERC1167: create2 failed");

}
```

Approaches

Upgrading is an anti pattern - don't do it

There is an argument for this approach, it favours decentralisation
See Upgradability is a bug

- Smart contracts are useful because they're trustless.
- Immutability is a critical feature to achieve trustlessness.
- Upgradeability undermines a contract's immutability.

It may be sufficient to parameterise your contract and adjust those parameters instead of upgrading

For example Maker DAO's stability fee, or a farming reward rate that can be adjusted by the an administrator (or a DAO, or some governance mechanism)

Migrate the data manually

Deploy your V2 contract, and migrate manually any existing data

Advantages

Conceptually simple.
No reliance on libraries.

Disadvantages

Can be difficult and costly (gas and time) in practice, and if the amount of data to migrate is large, it may hit gas limits when migrating.

Use a Registry contract

A registry (similar to ENS) holds the address of the latest version of the contract.
DApps should read this registry to get the correct address

Advantages

Simple to implement

Disadvantages

We rely on the DApp code to choose the correct contract
There is trust involved in the developers, not to switch out a contract with reasonable terms for an unfavourable one.
"Keep in mind that users of your smart contract don’t trust you, and that’s why you wrote a smart contract in the first place."

This doesn't solve the data migration problem

Separate code into function and data contracts

Advantages

Maybe simple to implement
Partially solves data migration

Disadvantages

It is difficult to get the security implemented correctly
Fails if your data contract needs to change.

Choose an function at runtime in other contracts or libraries

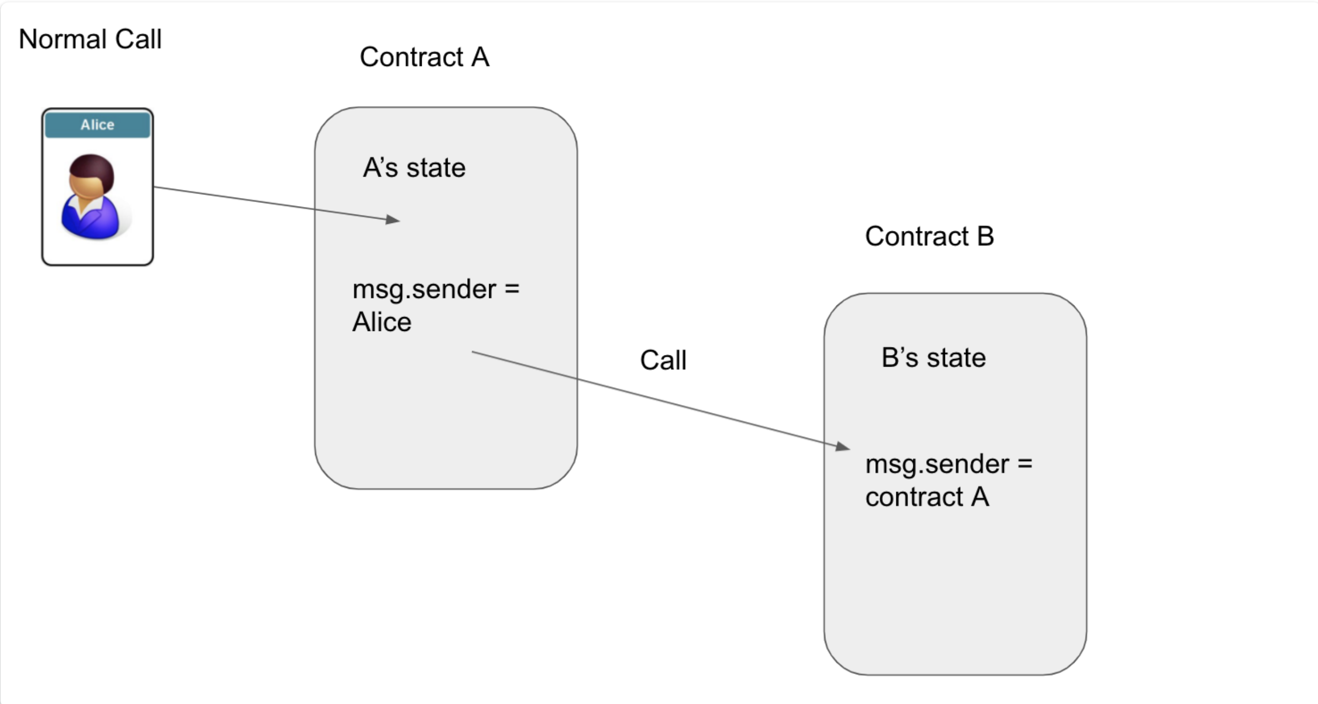
This is moving towards the Proxy patterns and the Diamond pattern
Essentially the Strategy Pattern
Compound use this approach with their interest rate model
A variant of this is the use of pluggable modules
such as in Gnosis Safe
The module approach is additive, if there is a bug in the core code this approach won't fix it.

Digression - Message Calls

There are a number of ways for contracts to call each other
See : Read the Docs : Message Calls

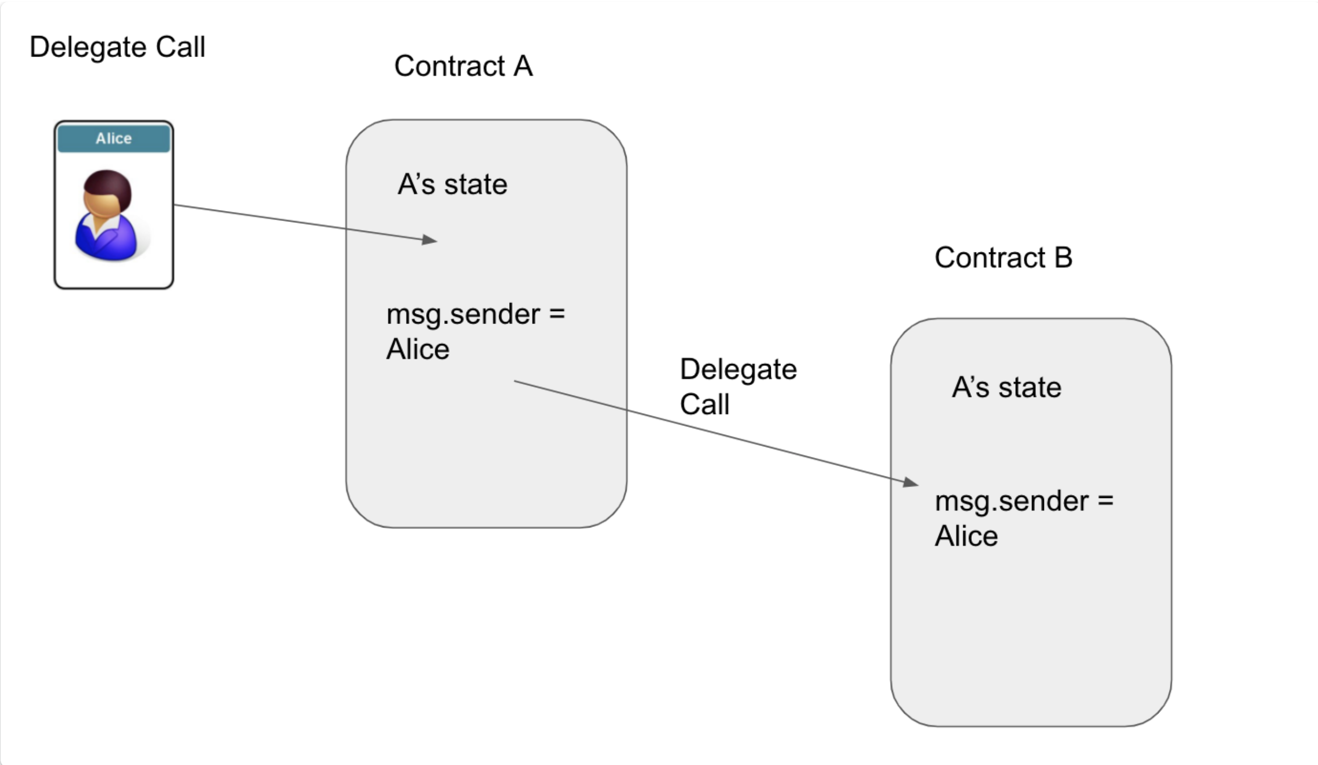
Message calls have a source (this contract), a target (the other contract), data payload, Ether, gas and return data.

The other contract gets a fresh context to work in, its own contract state as you would expect.



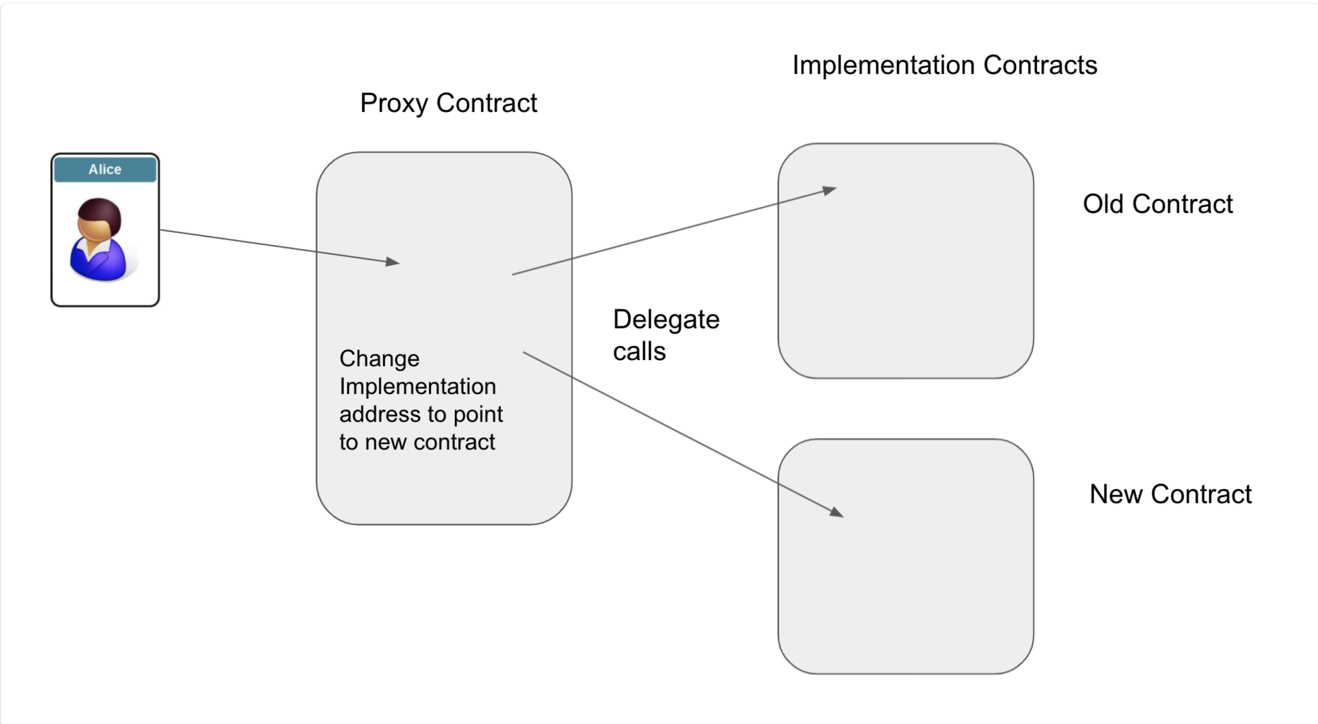
Delegate Call

There is a special type of call, **Delegate Call** which behaves differently in that it executes in the context of the calling contract (and msg.sender and msg.value do not change)



Another way to look at this is to think of contract A loading and executing contract B's code.

Upgrade Process




```
contract AdminUpgradeableProxy {
    address implementation;
    address admin;

    fallback() external payable {
        // delegate here
```

```
    }

    function upgrade(address newImplementation) external {
        require(msg.sender == admin);
        implementation = newImplementation;
    }
}
```

This can be open to vulnerabilities, instead the
Transparent Proxy Contract can be used

```
contract TransparentAdminUpgradeableProxy {
    address implementation;
    address admin;

    fallback() external payable {
        require(msg.sender != admin);
        // delegate here
    }

    function upgrade(address newImplementation) external {
        if (msg.sender != admin) fallback();
        implementation = newImplementation;
    }
}
```

This pattern is widely used, but comes at a cost because of the additional lookup of the implementation address and admin address

A cheaper and more recent alternative is the universal upgradeable proxy standard (UUPS)

In this pattern, the upgrade logic is placed in the implementation contract.

```
contract UUPSProxy {
    address implementation;

    fallback() external payable {
       // delegate here
    }
}

abstract contract UUPSProxiable {
    address implementation;
    address admin;

    function upgrade(address newImplementation) external {
```

```
            require(msg.sender == admin);
            implementation = newImplementation;
        }
    }
```

COST COMPARISON

	Transparent	UUPS
Proxy Deployment	740k + 480k ProxyAdmin	390k
Implementation Deployment	+ 0	+ 320k
Runtime Overhead	7.3k	4.9k

Overwriting data

But what about the data is there a possibility of overwriting data in our proxy contract unintentionally ?

LAYOUT OF VARIABLES IN STORAGE

From Solidity Documentation

State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings, data is stored contiguously item after item starting with the first state variable, which is stored in slot 0. For each variable, a size in bytes is determined according to its type.

MAPPINGS AND DYNAMIC ARRAYS

Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored “in between” the state variables preceding and following them. Instead, they are considered to occupy only 32 bytes with regards to the rules above and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.

If we have our proxy and implementation like this

```
contract UUPSProxy {
    address implementation;

    fallback() external payable {
       // delegate here
    }
}

abstract contract UUPSProxiable {
    uint256 counter;
    address implementation;
    address admin;

function foo() public {
    counter ++;
}

    function upgrade(address newImplementation) external {
        require(msg.sender == admin);
        implementation = newImplementation;
    }
}
```

If our implementation contract writes to the slot that it sees as counter, then it will overwrite the implementation variable.

To prevent this we use **Unstructured storage**
Open Zeppelin puts the implementation address at a 'random' address in storage

```
bytes32 private constant implementationPosition = bytes32(uint256(
  keccak256('eip1967.proxy.implementation')) - 1
));
```

This solves the problem for the implementation variable, but what about our other values in storage ?

Say our versions of the implementation contracts looks like this

Implementation_v0	Implementation_v1
address owner	address lastContributor
mapping balances	address owner
uint256 supply	mapping balances
...	uint256 supply
	...

Then we will have a storage collision when writing to lastContributor

The correct approach is only to append to the storage when we upgrade
Implementation_v0	Implementation_v1
address owner	address owner
mapping balances	mapping balances
uint256 supply	uint256 supply
... |address lastContributor	<=== Storage extension.
	...

Question - What about the constructor in the implementation contract ?

For upgradeable contracts we use an initialiser function rather than the constructor.

ETERNAL STORAGE

An alternative to the above is to split up our data types and store them in mappings

```
// Sample code, do not use in production!
contract EternalStorage {
    mapping(bytes32 => uint256) internal uintStorage;
    mapping(bytes32 => string) internal stringStorage;
    mapping(bytes32 => address) internal addressStorage;
    mapping(bytes32 => bytes) internal bytesStorage;
    mapping(bytes32 => bool) internal boolStorage;
    mapping(bytes32 => int256) internal intStorage;
}

contract Box is EternalStorage {
    function setValue(uint256 newValue) public {
        uintStorage['value'] = newValue;
    }
}
```

I include this for completeness but do not recommend it.

Using the UUPS plugin

What the plugins do

Both plugins provide two main functions, deployProxy and upgradeProxy, which take care of managing upgradeable deployments of your contracts. In the case of deployProxy, this means:

- Validate that the implementation is upgrade safe.
- Deploy a proxy admin for your project.
- Deploy the implementation contract.
- Create and initialize the proxy contract.

And when you call upgradeProxy:

- Validate that the new implementation is upgrade safe and is compatible with the previous one.

- Check if there is an implementation contract deployed with the same bytecode, and deploy one if not.

- Upgrade the proxy to use the new implementation contract.

Writing your contract

1. You need to include an initialising function and ensure it is called only once.
   Open Zeppelin provide a base contract to do this for you, you just need to inherit from it.

```
// contracts/MyContract.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

import "@openzeppelin/contracts-upgradeable/
proxy/utils/Initializable.sol";

contract MyContract is Initializable,UUPSUpgradeable {
    uint256 public x;

    function initialize(uint256 _x) public initializer {
        x = _x;
    }
}
```

Since this is not a constructor, the constructors of parent contracts will not be called, you will need to do this manually.

This also applies to initial values applied to variables (but constant is ok)
e.g.

```
contract MyContract {
    uint256 public hasInitialValue = 42;
    // equivalent to setting in the constructor
}
```

2. If you are using standard Open Zeppelin libraries, you need to switch to their upgradeable versions. It is recommended that your new version of the implementation contract inherits from your previous version

3. Use the plugins to deploy and upgrade your contracts for you
   Instead of the usual migration scripts in hardhat / truffle you will need something like this

HARDHAT

```
// In Hardhat config
require('@openzeppelin/hardhat-upgrades');


// scripts/create-box.js
const { ethers, upgrades } = require("hardhat");

async function main() {
  const Box = await ethers.getContractFactory("Box");
  const box = await upgrades.deployProxy(Box, [42]);
  await box.deployed();
  console.log("Box deployed to:", box.address);
}

main();
```

TRUFFLE

```
const { deployProxy } = require('@openzeppelin/truffle-upgrades');

const Box = artifacts.require('Box');

module.exports = async function (deployer) {
  const instance = await deployProxy(Box, [42],{ kind: 'uups' });
  console.log('Deployed', instance.address);
};
```

The deployProxy function has a number of options :

```
async function deployProxy(
  Contract: ContractClass,
  args: unknown[] = [],
  opts: {
    deployer: Deployer,
    initializer: string | false,
    unsafeAllow: ValidationError[],
```

```
    kind: 'uups' | 'transparent',
  } = {},
): Promise<ContractInstance>
```

Performing the upgrade

See documentation

```
const { upgradeProxy } = require('@openzeppelin/truffle-upgrades');

const Box = artifacts.require('Box');
const BoxV2 = artifacts.require('BoxV2');

module.exports = async function (deployer) {
  const existing = await Box.deployed();
  const instance = await upgradeProxy(existing.address, BoxV2, { kind:
'uups' });
  console.log("Upgraded", instance.address);
};
```

Testing the upgrade process

You can add the upgrade process to a unit test

```
const { deployProxy, upgradeProxy } = require('@openzeppelin/truffle-
upgrades');

const Box = artifacts.require('Box');
const BoxV2 = artifacts.require('BoxV2');

describe('upgrades', () => {
  it('works', async () => {
    const box = await deployProxy(Box, [42] { kind: 'uups' });
    const box2 = await upgradeProxy(box.address, BoxV2);

    const value = await box2.value();
    assert.equal(value.toString(), '42');
  });
});
```

OTHER FUNCTIONS

prepareUpgrade

Use this to allow the plugin to check that your contracts are upgrade safe and deploy a new implementation contract ` admin.changeAdminForProxy

Use this to change the administrator of the proxy contract.

SECURITY CONSIDERATIONS

1. Always initialise your contract
2. Do not allow self destruct or delegatecall in your implementation contracts.

Diamond pattern

Based on EIP 2535

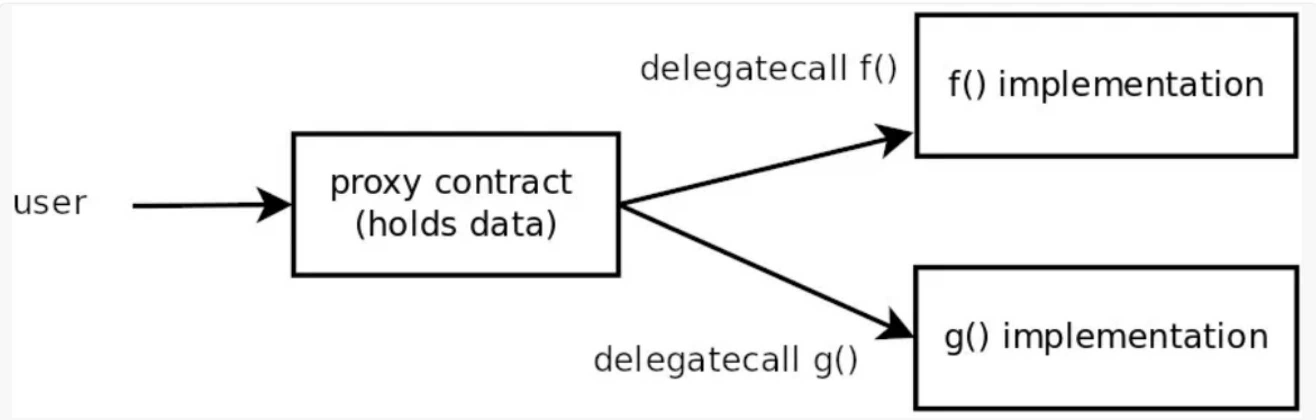
From Aavegotchi (https://docs.aavegotchi.com/overview/diamond-standard)
The diamond pattern is a contract that uses a fallback function to delegate function calls to multiple other contracts called facets. Conceptually a diamond can be thought of as a contract that gets its external functions from other contracts. A diamond has four standard functions (called the loupe) that report what functions and facets a diamond has. A diamond has a `DiamondCut` event that reports all functions/facets that are added/replaced/removed on a diamond, making upgrades on diamonds transparent.

The diamond pattern is a code implementation and organization strategy. The diamond pattern makes it possible to implement a lot of contract functionality that is compartmented into separate areas of functionality, but still using the same Ethereum address. The code is further simplified and saves gas because state variables are shared between facets.

Diamonds are not limited by the maximum contract size which is 24KB.

Facets can be deployed once and reused by any number of diamonds.

From Trail of Bits Audit

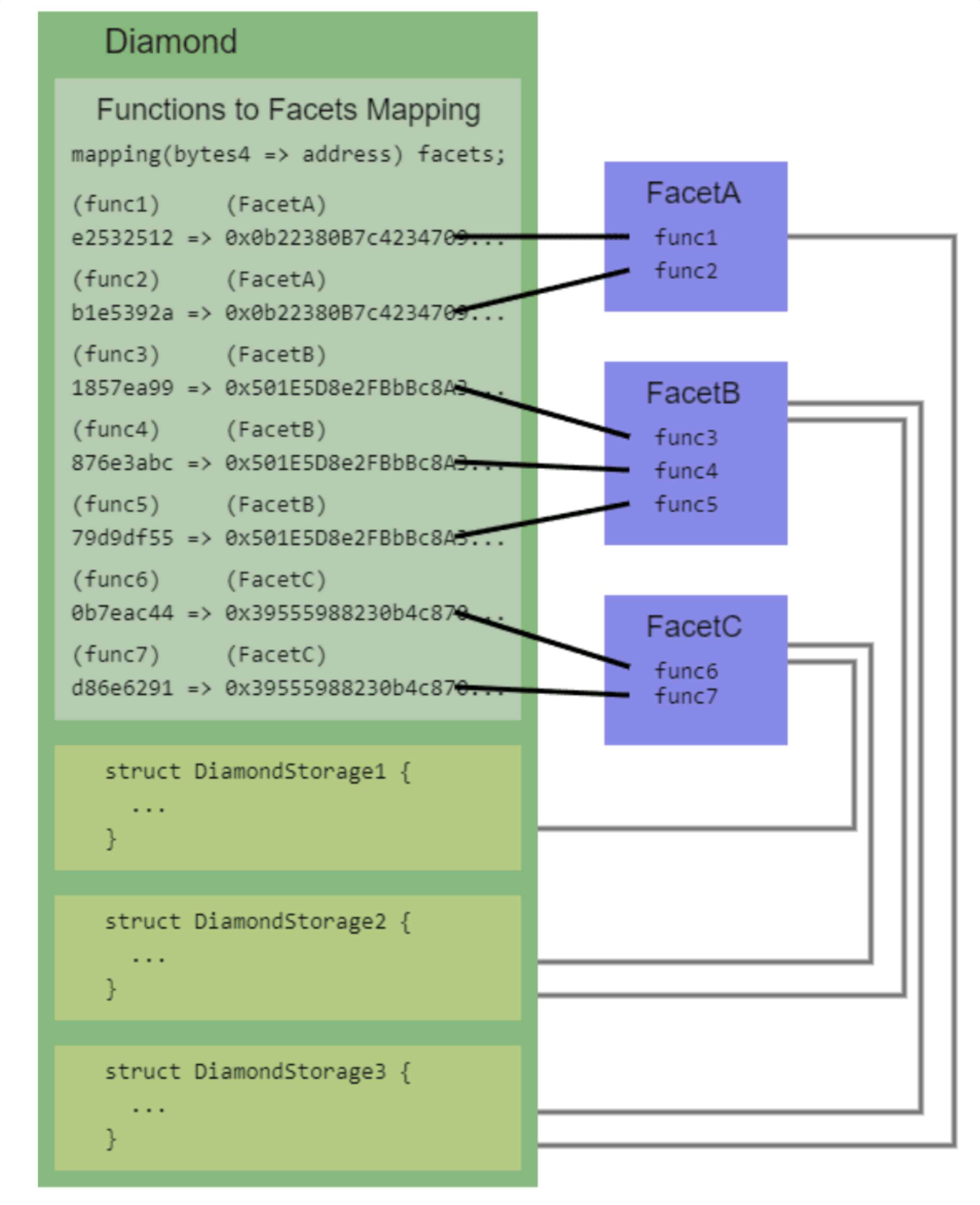



```
bytes32 constant POSITION = keccak256(
      "some_string"
 );
```

```
struct MyStruct {
    uint var1;
    uint var2;
}

function get_struct() internal pure returns(MyStruct storage ds) {
    bytes32 position = POSITION;
    assembly { ds_slot := position }
}
```

(The _slot suffix gives the storage address)



Trail of Bits Audit - Good idea, bad design

https://blog.trailofbits.com/2020/10/30/good-idea-bad-design-how-the-diamond-standard-falls-short/
"The code is over-engineered, with lots of unnecessary complexities, and we can't recommend it at this time."

But.. projects are using it
For example Aavegotchi

"AavegotchiDiamond provides a single Ethereum address for Aavegotchi functionality. All contract interaction with Aavegotchis is done with AavegotchiDiamond."

From Nick Mudge article
Diamonds solve these problems:

1. The maximum size a smart contract can be on Ethereum is 24kb. But sometimes larger smart contracts are needed or desired. Diamonds solve that problem.
2. Provides a structure to systematically and logically organize and extend larger smart contract systems so they don't turn into a spaghetti code mess.
3. Provides fine-grained upgrades. Other upgrade approaches require replacing functionality in bulk. With a diamond you can add, replace, or remove just the functionality that needs to be added, replaced or removed without affecting or touching other smart contract functionality.
4. Provides a single address for a lot of smart contract functionality. This makes integration with smart contracts and user interfaces and other software easier.

Question - How is this different to using a library ?

Metamorphic Contracts

CREATE AND CREATE2 AND SELFDESTRUCT

CREATE gives the address that a contract will be deployed to

```
keccak256(rlp.encode(deployingAddress, nonce))[12:]
```

CREATE2 introduced in Feb 2019

```
keccak256(0xff + deployingAddr + salt + keccak256(bytecode))[12:]
```

Contracts can be deleted from the blockchain by calling selfdestruct.

selfdestruct sends all remaining Ether stored in the contract to a designated address.

This can be used to preserve the contract address among upgrades, but not its state. It relies on the contract calling selfdestruct then being re deployed via CREATE2

Seen as 'an abomination' by some
See Metamorphic contracts
and
Abusing CREATE2 with Metamorphic Contracts
and
Efficient Storage