Lesson4

Design Patterns for Solidity

Some patterns from Fravoli

Behavioural Patterns

- Guard Check: Ensure that the behaviour of a smart contract and its input parameters are as expected.
- State Machine: Enable a contract to go through different stages with different corresponding functionality exposed. To implement a state machine we need to define
  - The states allowed
  - The transitions between those states
  - Function logic that will vary depending on which state we are currently in, or access to functions may depend on our current state.
- Randomness: Generate a random number of a predefined interval in the deterministic environment of a blockchain.

The randomness pattern is now out of date, since we have some randomness in a block field.

Oracle Patterns

See https://dev.to/ahmedmansoor012/ethereum-oracle-design-patterns-5api

- publish-subscribe
  broadcast service for frequently changing data
  when data is changed a flag is set / event
  interested (subscribed) parties poll the flag (or listen for events)
- immediate-read
  single lookup of (fairly fixed) data, probably stored in a contract
- request-response
  This is a comprehensive approach used by chainlink involving on and off chain components

Security Patterns
- Access Restriction: Restrict the access to contract functionality according to suitable criteria. An example is Access Control from Open Zeppelin
- Emergency Stop: Add an option to disable critical contract functionality in case of an emergency. Also known as an escape hatch .See Open Zeppelin pausable

- Checks-Effects-Interactions pattern
  See Solidity Docs

First check that the transaction should proceed (is there sufficient allowance ?)
Next change the state in this contract (reduce the allowance)
Finally interact with other contracts (send ether to an address / contract)

- Pull payments

```language-solidity
function withdrawFund(address recipient, uint amount) external
{
        require(recipient != address(0));
        require(amount > 0);
         (bool sent, bytes memory data) = recipient.call{value: amount}("");
require(sent, "Failed to send ");
          emit PaymentMade(recipient, amount);
}
```

Contract Registry

This can be seen as an anti pattern, if it is being used for upgradability, there are other approaches

Factory Contract

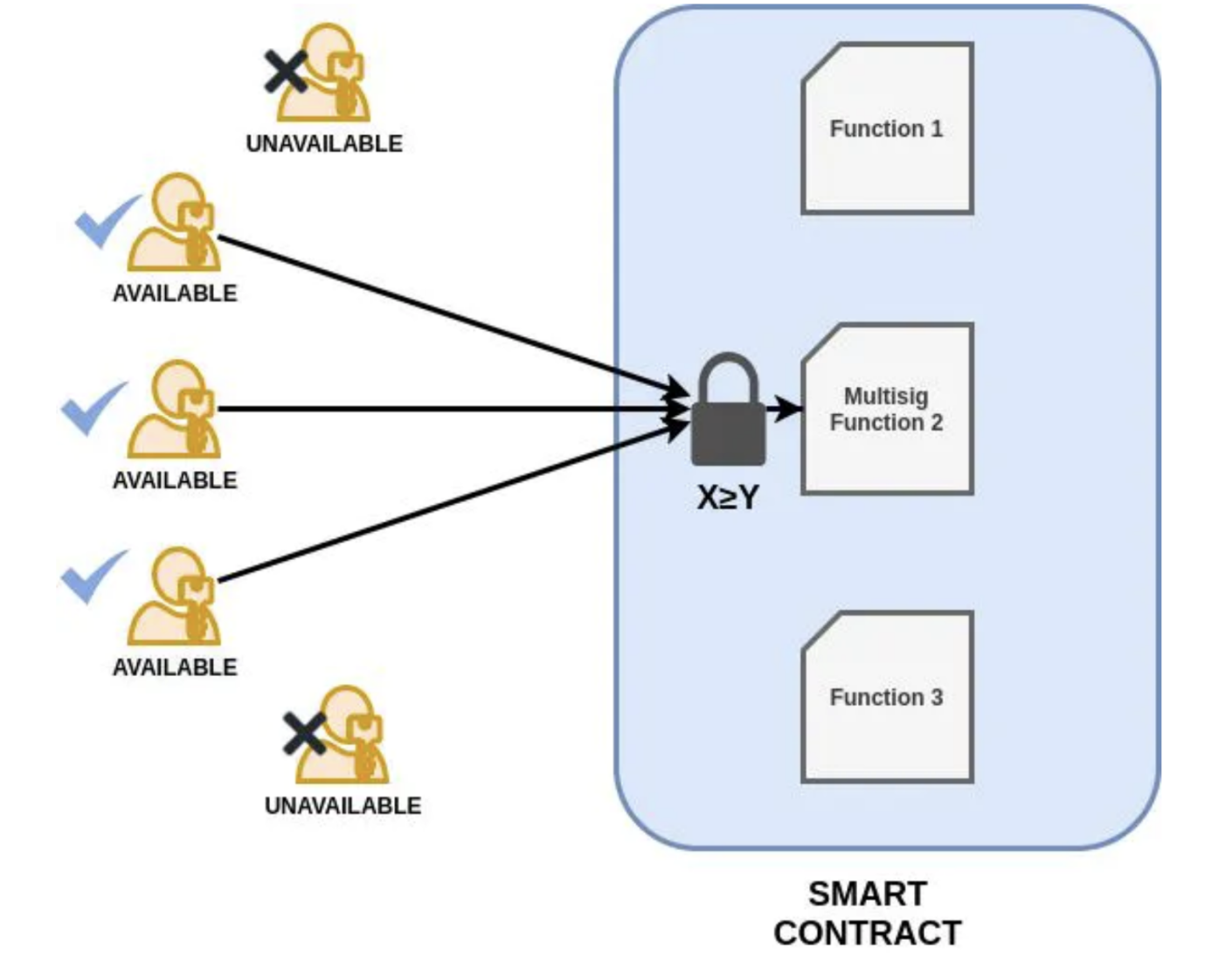
A factory contract is used to produce template contracts at runtime.

"factory.jpg" is not created yet. Click to create.

Incentive execution

Offer other users an incentive for calling for functions
An example is the ethereum alarm clock
For an alternative approach, see Chainlink keepers

Multisig Authorisation



See Gnosis safe for an implementation

Stack too deep workarounds

See repo

- Compile with `--via-ir` flag. This first compiles into Yul before optimisation, and works by moving stack variables into memory. You can pass this flag in both Hardhat and Foundry
- Block Scoping. Variables may not need function lifetime, so you can use blocks of code to keep the declaration close to the place where the code is used. The compiler will be able to handle the code piecemeal and ignore previous variables. This also has the advantage of being more easily verified as correct, so is general good practice.
- Use memory structs. You can pt variables into a struct and put an instance of that in memory, then the only item needed on the stack is a pointer to the in memory struct. This is often needed to reduce the number of function arguments.

We will see more ideas in the optimisation lesson

Upgradability Background

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

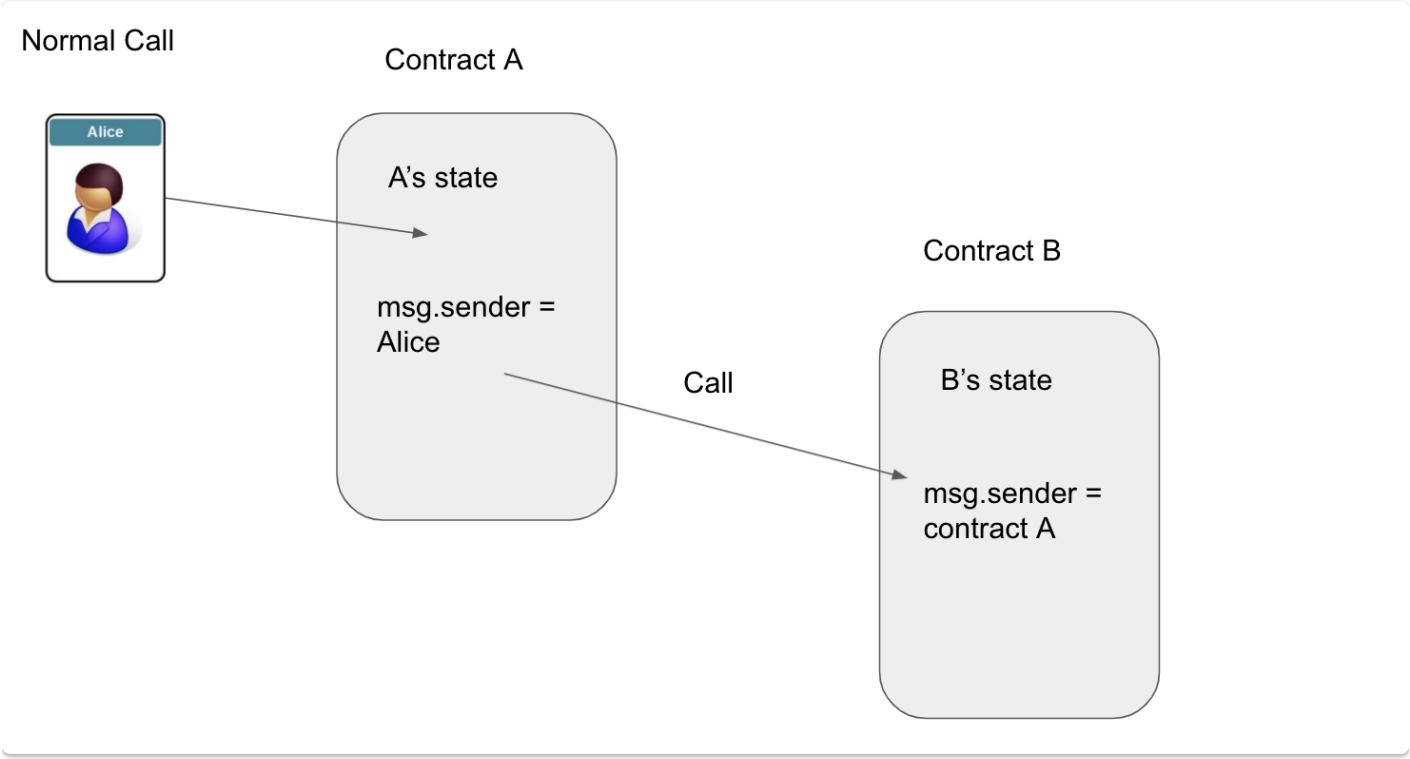
We will look at some of the patterns used to allow upgradability
I have taken examples from this guide, the guide applies to truffle or hardhat.
See State of upgrades

Digression - Message Calls

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

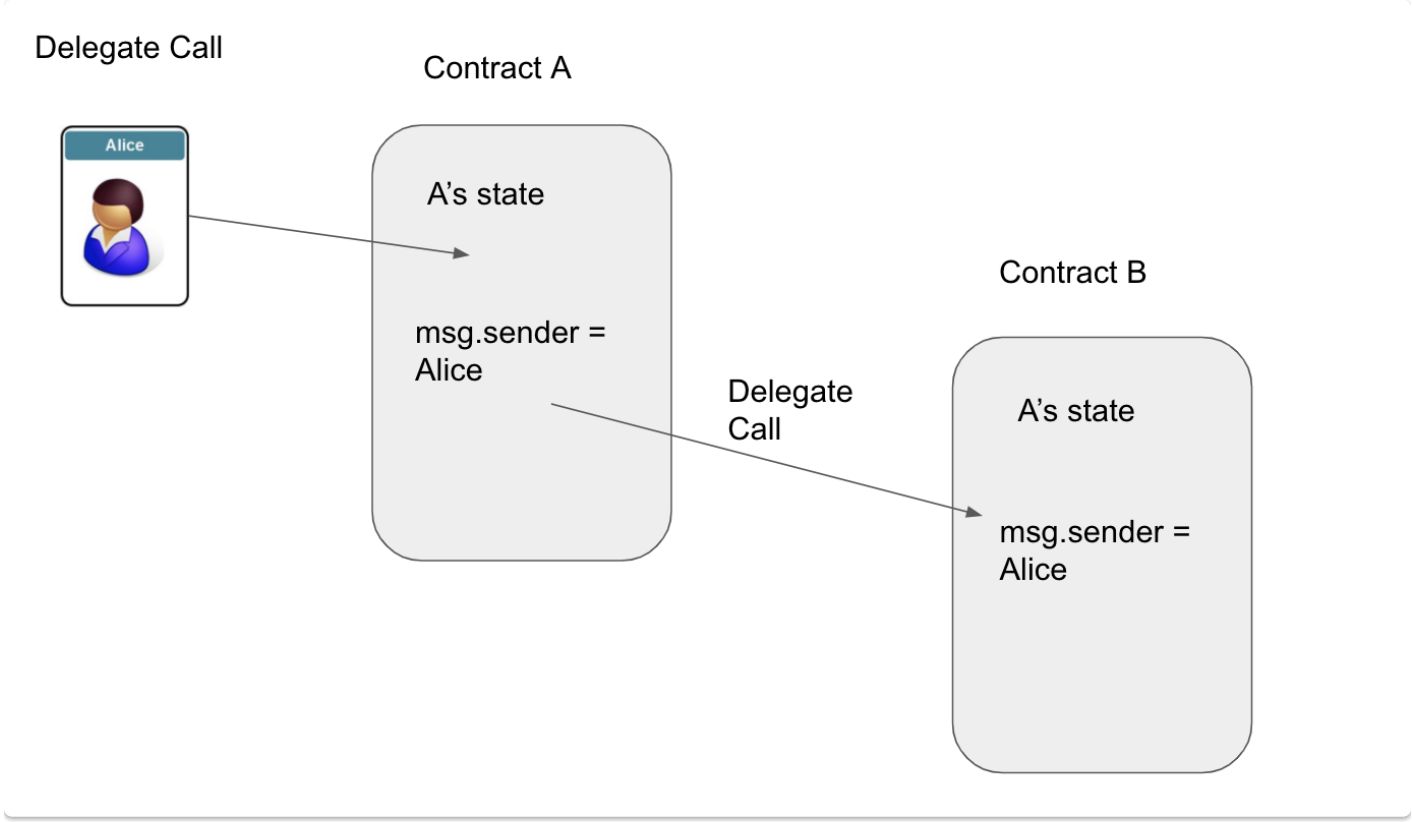
Message calls have a source (this contract), a target (the other contract), data payload, Ether, gas and return data.

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



Delegate Call

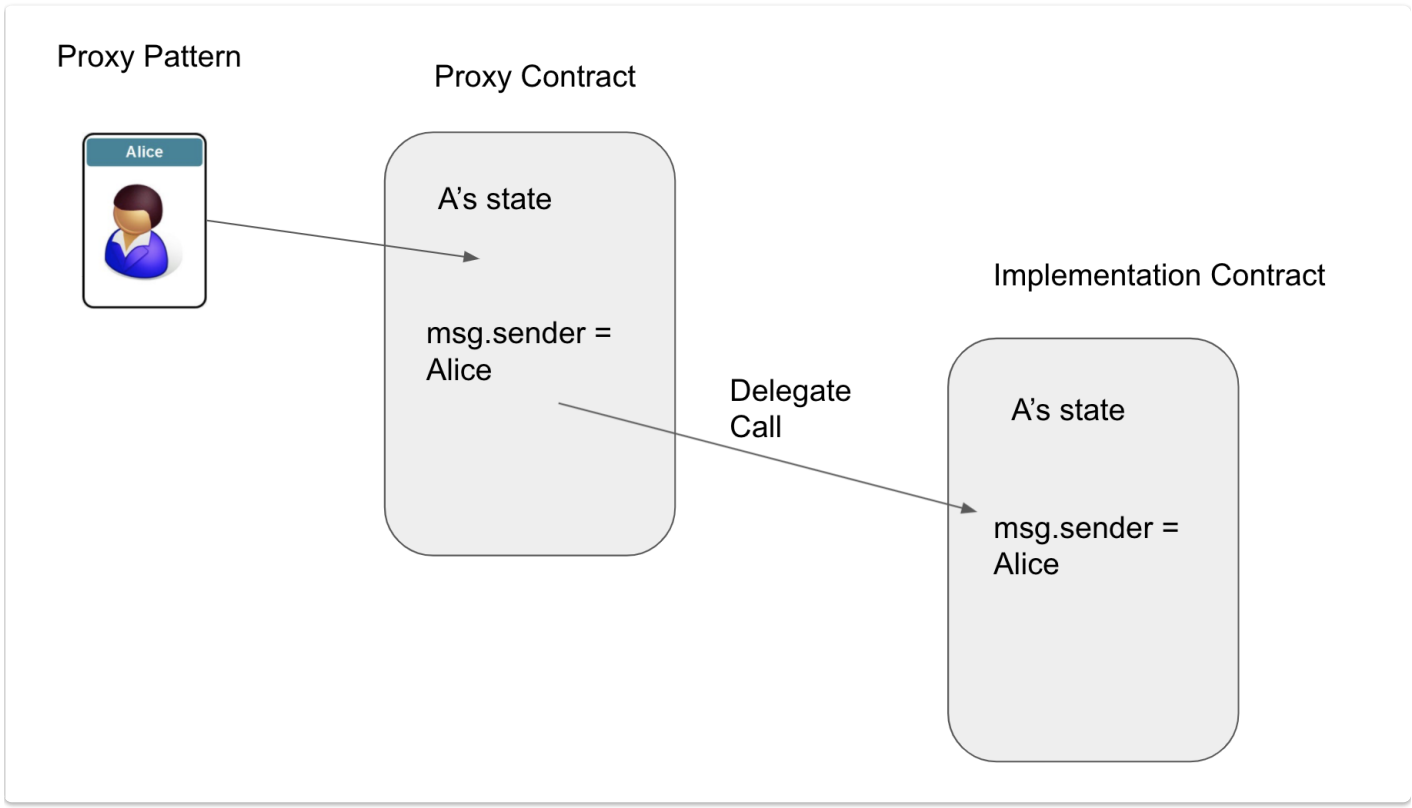
There is a special type of call, Delegate Call which behaves differently in that it executes in the context of the calling contract (and msg.sender and msg.value do not change)



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

Proxy patterns

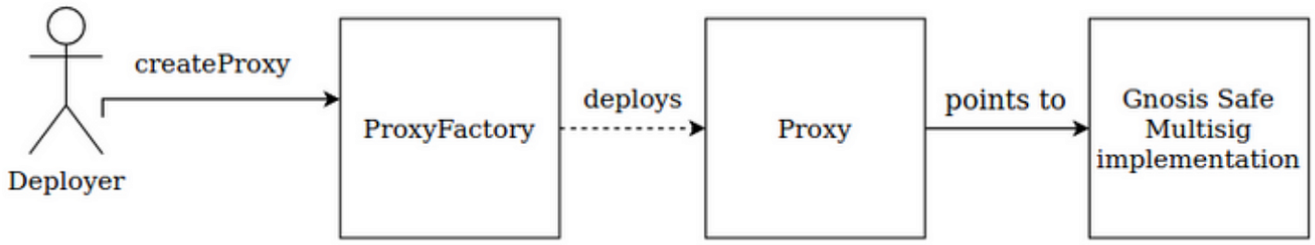
See [EIP 897]



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.

Approaches to Upgradability

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.
- Therefore, upgradeability is a bug.

From the article :
"We strongly advise against the use of these patterns for upgradable smart contracts. Both strategies have the potential for flaws, significantly increase complexity, and introduce bugs, and ultimately decrease trust in your smart contract. Strive for simple, immutable, and secure contracts rather than importing a significant amount of code to postpone feature and security issues."

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

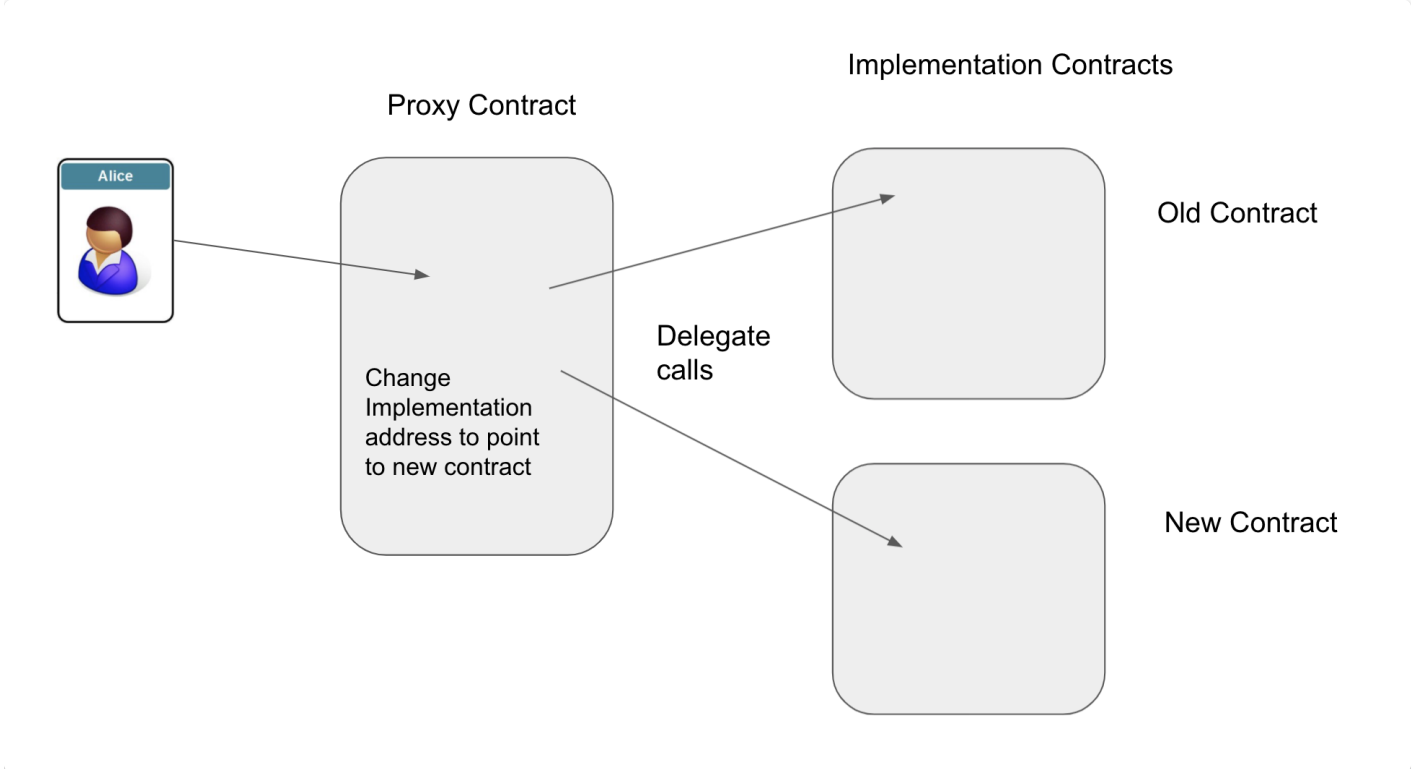
It is difficult to get the security implemented correctly
Fails if your data contract needs to change.

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.

Using Proxy contracts to upgrade

Upgrade Process




```language-none
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

```language-none
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.

```language-none
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.

FiatTokenV2_1 <<Contract>> 0xa2327a938febf5fec13bacfb16ae10ecbc4cbdcf			
slot	type: <inherited contract>.variable (bytes)		
0	unallocated (12)	address: Ownable._owner (20)	
1	unallocated (11)	bool: Pausable.paused (1)	address: Pausable.pauser (20)
2	unallocated (12)	address: Blacklistable.blacklister (20)	
3	mapping(address=>bool): Blacklistable.blacklisted (32)		
4	string: FiatTokenV1.name (32)		
5	string: FiatTokenV1.symbol (32)		
6	unallocated (31)	uint8: FiatTokenV1.decimals (1)	
7	string: FiatTokenV1.currency (32)		
8	unallocated (11)	bool: FiatTokenV1.initialized (1)	address: FiatTokenV1.masterMinter (20)
9	mapping(address=>uint256): FiatTokenV1.balances (32)		
10	mapping(address=>mapping(address=>uint256)): FiatTokenV1.allowed (32)		
11	uint256: FiatTokenV1.totalSupply_ (32)		
12	mapping(address=>bool): FiatTokenV1.minters (32)		
13	mapping(address=>uint256): FiatTokenV1.minterAllowed (32)		
14	unallocated (12)	address: Rescuable._rescuer (20)	
15	bytes32: EIP712Domain.DOMAIN_SEPARATOR (32)		
16	mapping(address=>mapping(bytes32=>bool)): EIP3009._authorizationStates (32)		
17	mapping(address=>uint256): EIP2612._permitNonces (32)		
18	unallocated (31)	uint8: FiatTokenV2._initializedVersion (1)	

If we have our proxy and implementation like this

```language-none
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
));
```
language-none

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
	...

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

```language-none
// 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                                    language-none
// 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
}
```
language-none

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

```language-none
// 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

```language-none
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 :

```language-none
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);
};
```

language-none

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');
  });
});
```

language-none

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.

The closest I have found for Foundry is this project

Diamond pattern

Based on EIP 2535

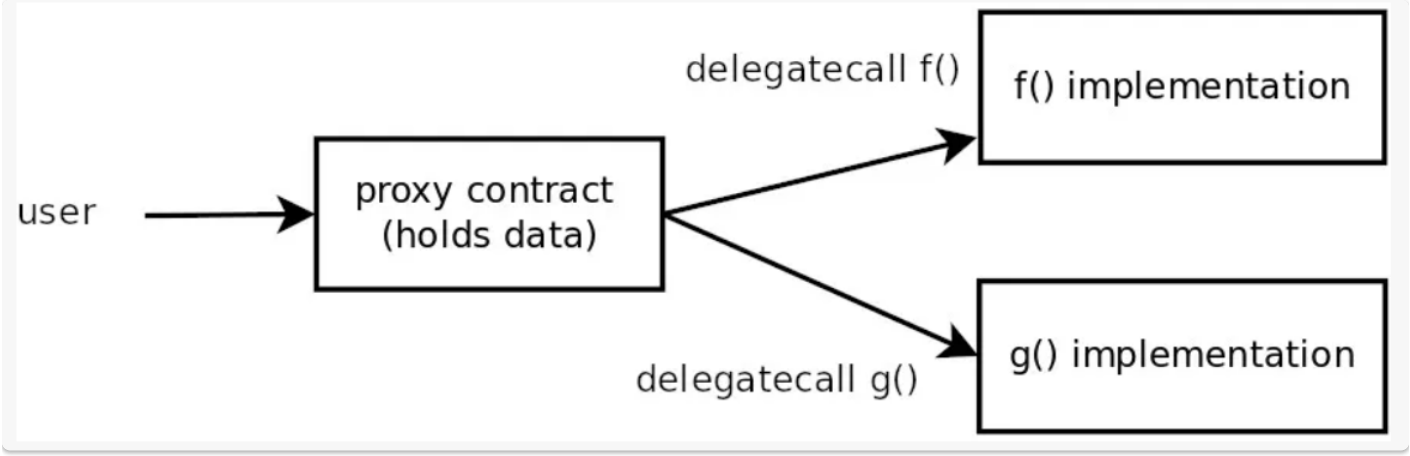
From Aavegotchi (https://docs.aavegotchi.com/overview/diamond-standard)
The diamond pattern is a contract that uses a fallback function to delegate function calls to multiple other contracts called facets. Conceptually a diamond can be thought of as a contract that gets its external functions from other 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

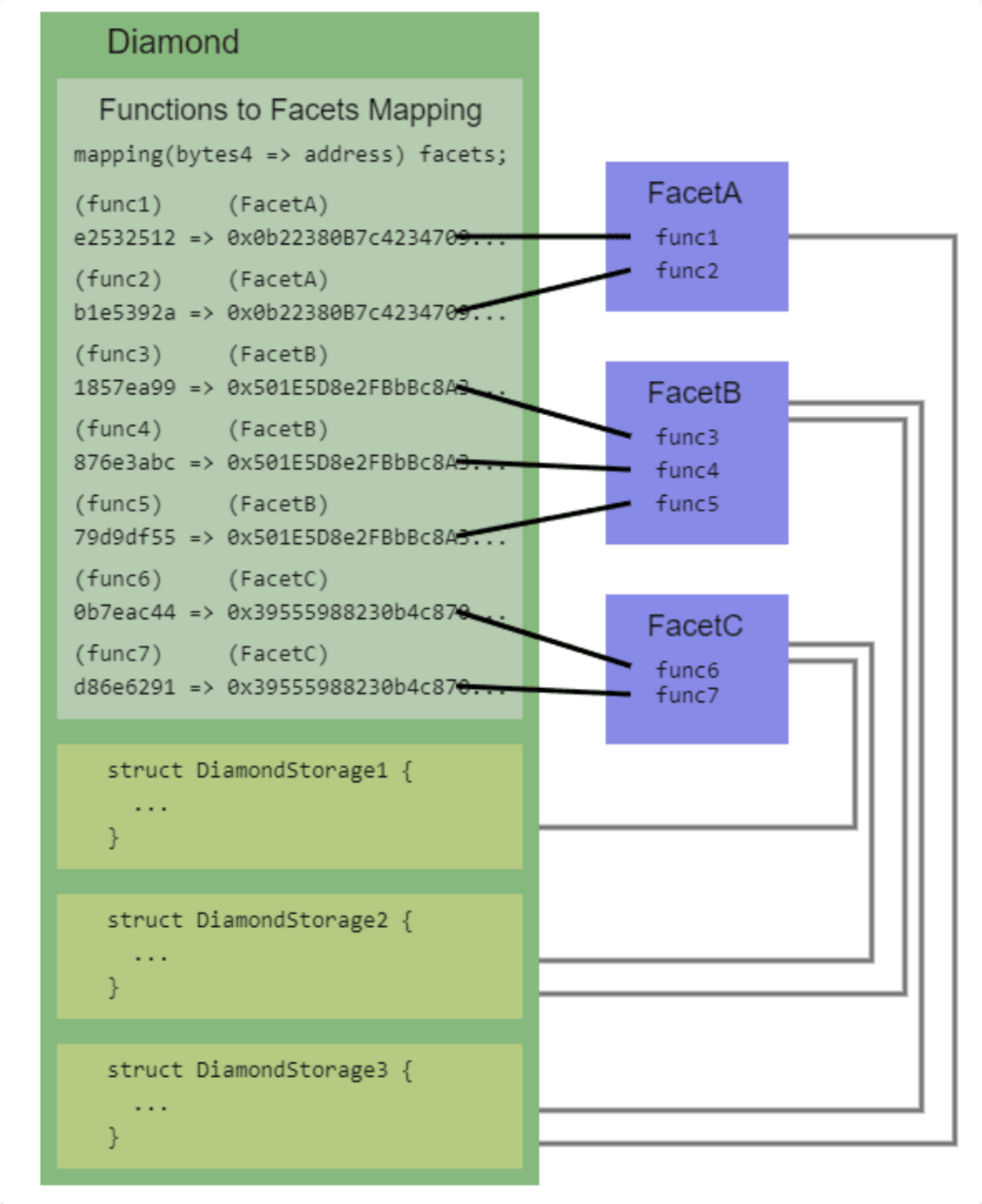



```
  bytes32 constant POSITION = keccak256(
    "some_string"
);

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:]
```
language-none

CREATE2 introduced in Feb 2019

```
keccak256(0xff + deployingAddr + salt + keccak256(bytecode))[12:]
```
language-none

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

IDE General Techniques

Importing from Github in Remix

See Documentation
You can import directly from github or npm

```language-none
import "https://github.com/OpenZeppelin/openzeppelin-contracts
/contracts/access/Ownable.sol";
```

or

```language-none
import "@openzeppelin/contracts@4.2.0/token/ERC20/ERC20.sol";
```

Logging in Remix

https://remix-ide.readthedocs.io/en/latest/hardhat_console.html

Remix IDE supports hardhat console library while using `JavaScript VM`. It can be used while making a transaction or running unit tests.
To try it out, you need to put an import statement and use `console.log` to print the value as shown in image.

Home | 2_Owner.sol

```
 * @dev Set & change owner
 */

import "hardhat/console.sol";

contract Owner {

    address private owner;

    // event for EVM logging
    event OwnerSet(address indexed oldOwner, address indexed newOwner);

    // modifier to check if caller is owner
    modifier isOwner() {
        // If the first argument of 'require' evaluates to 'false', execution terminates and all
        // changes to the state and to Ether balances are reverted.
        // This used to consume all gas in old EVM versions, but not anymore.
        // It is often a good idea to use 'require' to check if functions are called correctly.
        // As a second argument, you can also provide an explanation about what went wrong.
        require(msg.sender == owner, "Caller is not owner");
        _;
    }

    /**
     * @dev Set contract deployer as owner
     */
    constructor() {
        owner = msg.sender; // 'msg.sender' is sender of current call, contract deployer for a constructor
        emit OwnerSet(address(0), owner);
    }

    /**
     * @dev Change owner
     * @param newOwner address of new owner
     */
    function changeOwner(address newOwner) public isOwner {
        console.log('msg.sender :', msg.sender);
        emit OwnerSet(owner, newOwner);
        owner = newOwner;
    }
```

Deployed Contracts

OWNER AT 0XD91...39138 (MEMORY)

changeOwner | 0xAb8483F64d9C6d1EcF9b

getOwner

Low level interactions

CALLDATA

Transact

0 listen on network | Search with transaction hash or address

[vm] from: 0x5B3...eddC4 to: Owner.(constructor) value: 0 wei data: 0x608...70033 logs: 1 hash: 0x7c5...7e0aa

transact to Owner.changeOwner pending ...

[vm] from: 0x5B3...eddC4 to: Owner.changeOwner(address) 0xd91...39138 value: 0 wei data: 0xa6f...35cb2 logs: 1 hash: 0xaf1...8ed3c

```
console.log:
 msg.sender : 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
```

>

Forking from mainnet

The easiest way to try this feature is to start a node from the command line:

```
npx hardhat node --fork https://eth-mainnet.alchemyapi.io/v2/<key>
```

You can also configure Hardhat Network to always do this:

```js
networks: {
  hardhat: {
    forking: {
      url: "https://eth-mainnet.alchemyapi.io/v2/<key>",
    }
  }
}
```

Solidity Templates

Paul Berg has created some useful templates

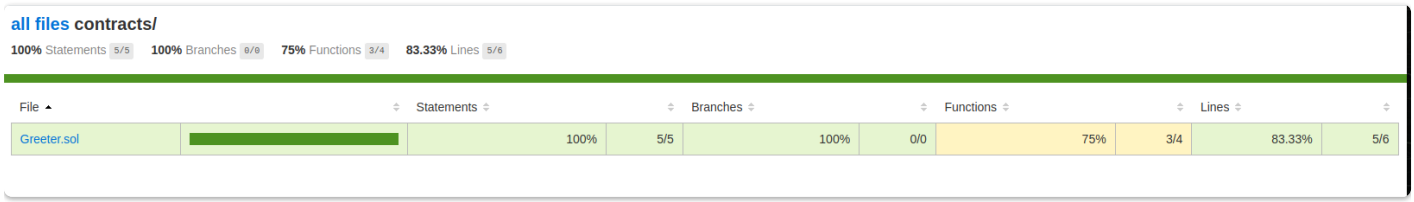
Solidity Template

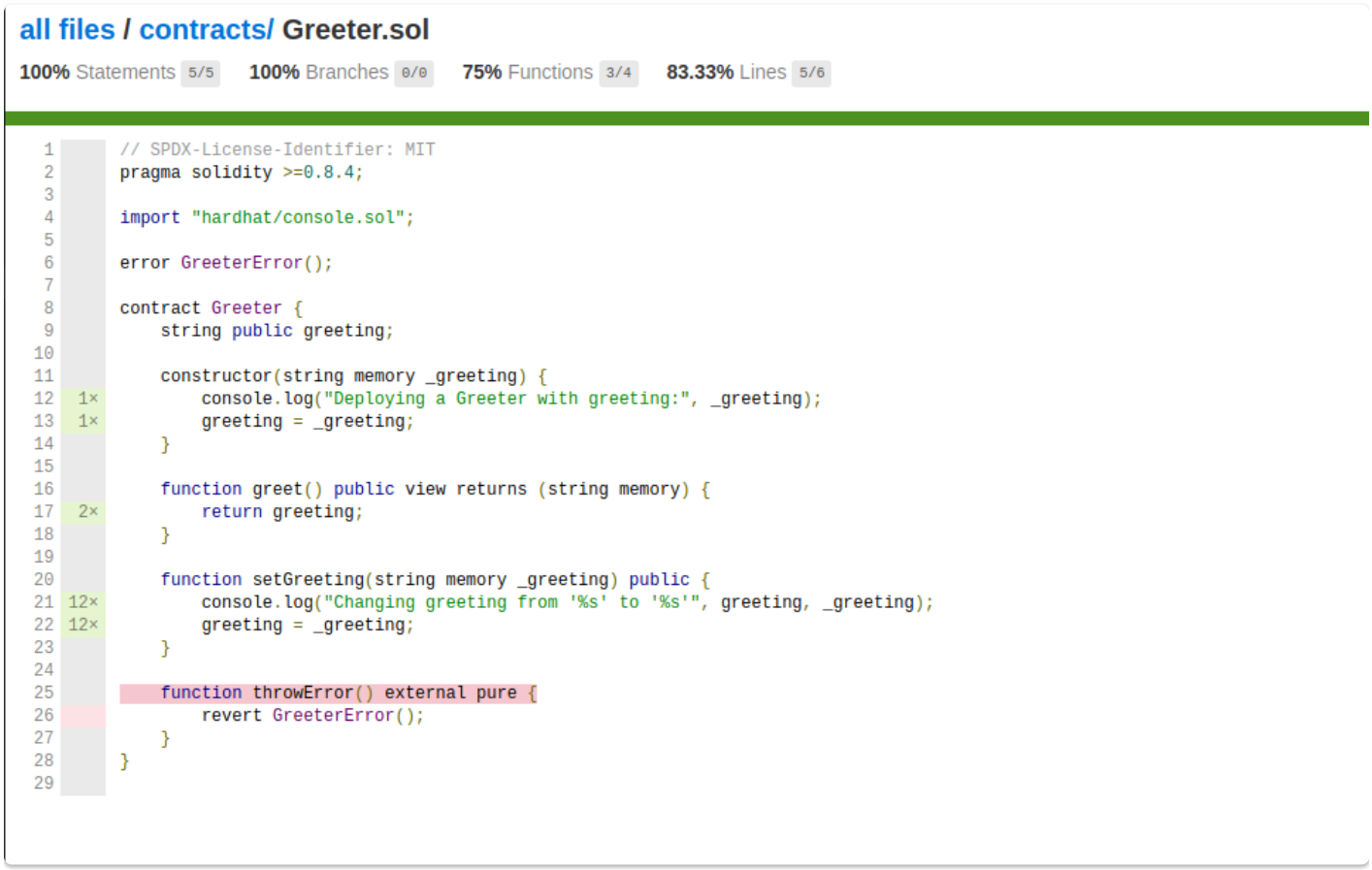
Repo
This will give you

- Hardhat: compile, run and test smart contracts
- TypeChain: generate TypeScript bindings for smart contracts
- Ethers: renowned Ethereum library and wallet implementation
- Solhint: code linter
- Solcover: code coverage
- Prettier Plugin Solidity: code formatter

COVERAGE REPORTS

```
yarn coverage
```





Gas report

```
REPORT_GAS=true yarn test
```

Foundry Template

Repo
This gives you :

- Forge: compile, test, fuzz, debug and deploy smart contracts
- PRBTest: modern collection of testing assertions and logging utilities
- Forge Std: collection of helpful contracts and cheatcodes for testing
- Solhint: code linter
- Prettier Plugin Solidity: code formatter

You can install from the github template button, or from the command line with

```
forge init my-project --template https://github.com/PaulRBerg/foundry-template
cd my-project
yarn install # install solhint and prettier etc.
```