Lecture 29 - Distributed Autonomous Organizations (DAO) and Proxy Contracts

What is a DAO?

A DAO (Decentralized Autonomous Organization) as an organization represented by rules encoded as a transparent computer program, controlled by the organization members, and not influenced by a centralized control structure. As the rules are embedded into the code, no managers are needed, thus removing any bureaucracy or hierarchy hurdles.

Paul Grahm, founder of Ycombinator, figures that getting rid of the “pointy haired” manager improves productivity by a 2 to 4x factor.

Wyoming has “enshrined” this into law...

The Wyoming DAO Law: https://www.wyoleg.gov/Legislation/2021/SF0038

This is similar to another Wyoming invention, the Limited Liability Corporation (LLC).

DAO’s do not obligate you from any of the requirements of a corporation or LLC. They do provide a new way to “manage” an organization.

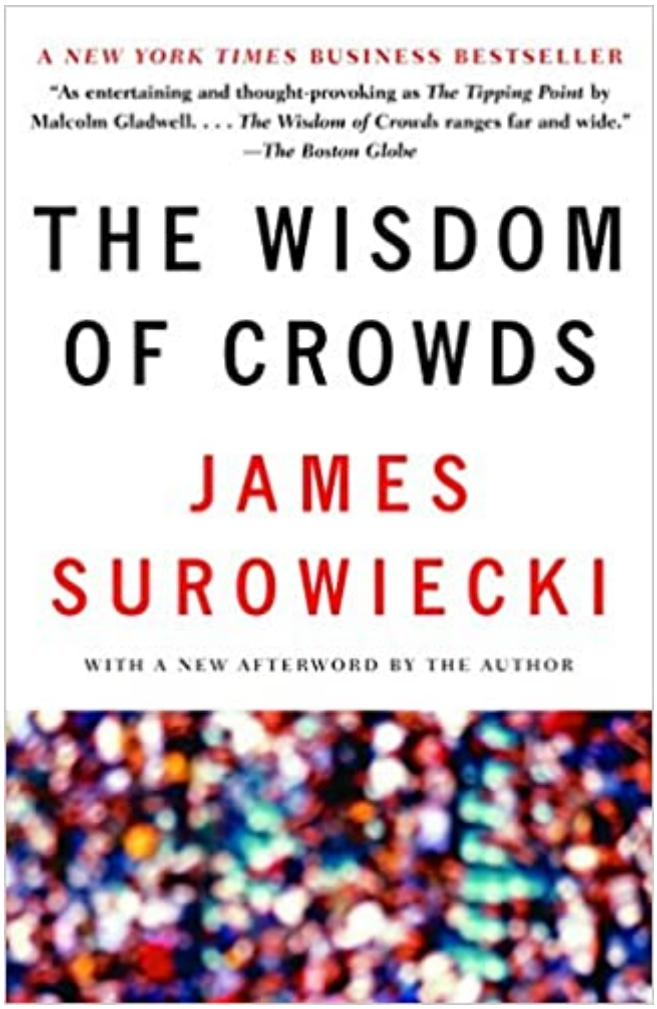
Some existing organizations have gotten rid of helical management. Zappos for example - fired all it’s managers and “teams” just make decisions on how and what they are going to work on.

Lot’s of stuff is “organized” in different ways. This is a new form of organization.

Some of the reason for a DAO is the wisdom of the crowd.

Why/How wisdom of the crowd works....

Most of this is from https://www.amazon.com/Wisdom-Crowds-James-Surowiecki/dp/0385721706/ref=sr_1_1?crid=267JDRKCNTR2J&keywords=Surowiecki&qid=1649072078&sprefix=surowiecki+%2Caps%2C131&sr=8-1



Oinas-Kukkonen in A. Koohang et al. (Eds): Knowledge Management: Theoretical Foundation. Informing Science Press, Santa Rosa, CA, US, pp. 173-189. captures the wisdom of crowds approach with the following eight conjectures:

1. It is possible to describe how people in a group think as a whole.
2. In some cases, groups are remarkably intelligent and are often smarter than the smartest people in them.
3. The three conditions for a group to be intelligent are diversity, independence, and decentralization.
4. The best decisions are a product of disagreement and contest.
5. Too much communication can make the group as a whole less intelligent.
6. Information aggregation functionality is needed.
7. The right information needs to be delivered to the right people in the right place, at the right time, and in the right way.
8. There is no need to chase the expert.

As created in Wyoming the DAO is a “staked” democracy.

SF0038 (the Wyoming DAO law) has 2 flavors - the human voting flavor and the “autonomous” system.

Note that “diversity, independence, and decentralization” are critical factors. This is a general management principal. You make better decisions when the “team” is diversified. You get better code out of “independent” workers. You get more productivity out of decentralized teams.

Prediction markets

The most common application is the prediction market, a speculative or betting market created to make verifiable predictions. Betfair traded $28 billion last year. To create a good perdition market the “members” have to have a financial stake in the outcome. It is not “Who do you think will win the 2024 election” it is “Who will you vote for - with $$ in result.”

Betfair has a better than 80% prediction accuracy on elections.

This is the system of “wisdom of the crowds” that underpins risk forecasts for insurance.

The principle of the prediction market is also used in project management software to let team members predict a project’s “real” deadline and budget for a project.

problems with wisdom of the crowd

Examples:

1. stock market bubbles
2. things like the Challenger Disaster
3. personality cults
4. disinformation

problems with DAO (human managed)

1. What to vote on needs to be delivered, explained and understood. (6, 7 above)
2. Voting Actually Taking Place - Members must participate
3. Voters voting according to the Wyoming DAO law - tracking / vote-taking is a process
4. Votes are recorded - you would think this is easy - it is not.

Technical Requirements

From SF0038: “An algorithmically managed decentralized autonomous organization may only form under this chapter if the underlying smart contracts are able to be updated, modified or otherwise upgraded.”

So… How do you upgrade a smart contract…

Just a regular contract is forever - once it is written with a transaction to the chain this is immutable data.

The answer is a thing called a “Proxy” contract. The Proxy is a contract that is the front end and receives the calls - then uses the address of a destination contract and passes the calls on to this other contract. The owner of the proxy can point it at any contract.

The Voting Contract

```
1: // SPDX-License-Identifier: MIT
2: pragma solidity >=0.6.0 <=0.9.0;
3:
4: import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
5:
6: contract DaoVoterContractUpgradableV1 is ERC721 {
7:
8:     mapping (uint256 => uint256) public tokenIdToPrice;
9:     string private baseURI;
```

```
10:     int public nMinted;        // # of tokens minted.
11:     address payable owner;
12:
13:     event NFTPurchased(uint256 indexed NFT, address seller, address buyer, uint256 price);
14:     event NFTAvailable(uint256 indexed NFT, uint256 price);
15:     event NFTRemoved(uint256 indexed NFT);
16:
17:     event VoterRegistered(uint256 indexed NFT);
18:     event ProposalRegistered(uint256 indexed NFT);
19:
20:     constructor() ERC721("Wyoming NFT", "WYNFT") {
21:         owner = payable(msg.sender);
22:     }
23:
24:     modifier onlyOwner() {
25:         require( msg.sender == owner, "Sender not authorized.");
26:         // Do not forget the "_;"! It will be replaced by the actual function
27:         // body when the modifier is used.
28:         _;
29:     }
30:
31:     function initialize(string memory tokenURI) public onlyOwner() {
32:         baseURI = tokenURI;
33:         nMinted = 0;
34:     }
35:
36:     // Make `_newOwner` the new owner of this contract.
37:     function changeOwner(address payable _newOwner) public onlyOwner() {
38:         owner = _newOwner;
39:     }
40:
41:     function mintNFT(uint256 _tokenNFT) public returns (uint256) {
42:         uint256 newItemId = _tokenNFT;
43:         _safeMint(msg.sender, newItemId);
44:         nMinted += 1;
45:         return newItemId;
46:     }
47:
48:     function registerVoter(uint256 _tokenNFT) public returns (uint256) {
49:         uint256 newItemId = _tokenNFT;
50:         _safeMint(msg.sender, newItemId);
51:         nMinted += 1;
52:         emit VoterRegistered(newItemId);
53:         return newItemId;
54:     }
55:
56:     function createProposal(uint256 _tokenNFT) public returns (uint256) {
57:         uint256 newItemId = _tokenNFT;
58:         _safeMint(msg.sender, newItemId);
59:         nMinted += 1;
60:         ProposalRegistered(newItemId);
61:         return newItemId;
62:     }
63:
64:     // function balanceOf(address owner) public view virtual override returns (uint256) {
65:
66:     function _baseURI() internal view override returns (string memory) {
67:         return baseURI;
68:     }
69:
70:     function allowBuy(uint256 _tokenId, uint256 _price) external {
71:         require(msg.sender == ownerOf(_tokenId), 'Not owner of this token');
72:         require(_price > 0, 'Price must be larger than zero');
73:         tokenIdToPrice[_tokenId] = _price;
74:
75:         emit NFTAvailable(_tokenId, _price);
```

```
    }

    function disallowBuy(uint256 _tokenId) external {
        require(msg.sender == ownerOf(_tokenId), 'Not owner of this token');
        tokenIdToPrice[_tokenId] = 0;

        emit NFTRemoved(_tokenId);
    }

    function buy(uint256 _tokenId) external payable {
        uint256 price = tokenIdToPrice[_tokenId];
        require(price > 0, 'This token is not for sale');
        require(msg.value == price, 'Incorrect value');

        address seller = ownerOf(_tokenId);
        _transfer(seller, msg.sender, _tokenId);
        tokenIdToPrice[_tokenId] = 0; // not for sale anymore
        payable(seller).transfer(msg.value); // send the ETH to the seller

        emit NFTPurchased(_tokenId, seller, msg.sender, msg.value);
    }

    // This function is called for all messages sent to
    // this contract, except plain Ether transfers
    // (there is no other function except the receive function).
    // Any call with non-empty calldata to this contract will execute
    // the fallback function (even if Ether is sent along with the call).
    fallback() external payable {}

    // This function is called for plain Ether transfers, i.e.
    // for every call with empty calldata.
    receive() external payable {}

    function withdraw( uint256 _amount ) public onlyOwner() {
        owner.transfer(_amount);
    }

    // destroy the contract and reclaim the leftover funds.
    function kill() public onlyOwner() {
        //    Calling selfdestruct(address) sends all of the contract's current balance to address.
        selfdestruct(payable(msg.sender));
    }
}
```

The Proxy

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.6.0 <=0.9.0;

import "@openzeppelin/contracts/proxy/Proxy.sol";

/**
 * @title DaoVoterContract
 * @dev Gives the possibility to delegate any call to a foreign implementation.
 * Implements a proxy to the upgradable DaoVoterContractUpgradableV1 contract.
 */
contract DaoVoterContract is Proxy {

    address private owner;
    address private DaoVoterContractImpl;
    uint256 public version;
```

```
16:
17:     modifier onlyOwner() {
18:         require( msg.sender == owner, "Sender not authorized.");
19:         // Do not forget the "_;"! It will be replaced by the actual function
20:         // body when the modifier is used.
21:         _;
22:     }
23:
24:     constructor( address _addr, uint256 _version ) Proxy() {
25:         DaoVoterContractImpl = _addr;
26:         owner = msg.sender;
27:         version = _version;
28:     }
29:
30:     /**
31:     * @dev Make `_newOwner` the new owner of this contract.
32:     */
33:     function changeOwner(address _newOwner) public onlyOwner() {
34:         owner = _newOwner;
35:     }
36:
37:     /**
38:     * @dev Upgrade the underling contract to a new version. `_newAddr` is the address of
39:     * the new contract. `_newVersion` is the new version number.
40:     */
41:     function upgradeContract(address _newAddr, uint256 _newVersion) public onlyOwner() {
42:         DaoVoterContractImpl = _newAddr;
43:         version = _newVersion;
44:     }
45:
46:     /**
47:     * @dev Tells the address of the implementation where every call will be delegated.
48:     * @return address of the implementation to which it will be delegated
49:     */
50:     function _implementation() internal view override  returns (address) {
51:         return ( DaoVoterContractImpl );
52:     }
53:
54: }
```

What is in the “Proxy.sol”

```
 1: // SPDX-License-Identifier: MIT
 2: // OpenZeppelin Contracts (last updated v4.5.0) (proxy/Proxy.sol)
 3:
 4: pragma solidity ^0.8.0;
 5:
 6: /**
 7:  * @dev This abstract contract provides a fallback function that delegates all calls to another contract using
 8:  * instruction `delegatecall`. We refer to the second contract as the _implementation_ behind the proxy, and it
 9:  * be specified by overriding the virtual {_implementation} function.
10:  *
11:  * Additionally, delegation to the implementation can be triggered manually through the {_fallback} function, c
12:  * different contract through the {_delegate} function.
13:  *
14:  * The success and return data of the delegated call will be returned back to the caller of the proxy.
15:  */
16: abstract contract Proxy {
17:     /**
18:      * @dev Delegates the current call to `implementation`.
19:      *
20:      * This function does not return to its internal call site, it will return directly to the external caller.
```

```
     * this function does not return to its internal call site, it will return directly to the external caller.
     */
    function _delegate(address implementation) internal virtual {
        assembly {
            // Copy msg.data. We take full control of memory in this inline assembly
            // block because it will not return to Solidity code. We overwrite the
            // Solidity scratch pad at memory position 0.
            calldatacopy(0, 0, calldatasize())

            // Call the implementation.
            // out and outsize are 0 because we don't know the size yet.
            let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)

            // Copy the returned data.
            returndatacopy(0, 0, returndatasize())

            switch result
            // delegatecall returns 0 on error.
            case 0 {
                revert(0, returndatasize())
            }
            default {
                return(0, returndatasize())
            }
        }
    }

    /**
     * @dev This is a virtual function that should be overriden so it returns the address to which the fallback
     * and {_fallback} should delegate.
     */
    function _implementation() internal view virtual returns (address);

    /**
     * @dev Delegates the current call to the address returned by `_implementation()`.
     *
     * This function does not return to its internall call site, it will return directly to the external caller
     */
    function _fallback() internal virtual {
        _beforeFallback();
        _delegate(_implementation());
    }

    /**
     * @dev Fallback function that delegates calls to the address returned by `_implementation()`. Will run if
     * function in the contract matches the call data.
     */
    fallback() external payable virtual {
        _fallback();
    }

    /**
     * @dev Fallback function that delegates calls to the address returned by `_implementation()`. Will run if
     * is empty.
     */
    receive() external payable virtual {
        _fallback();
    }

    /**
     * @dev Hook that is called before falling back to the implementation. Can happen as part of a manual `_fal
     * call, or as part of the Solidity `fallback` or `receive` functions.
     *
     * If overriden should call `super._beforeFallback()`.
     */
    function _beforeFallback() internal virtual {}
}
```

```
86: }
```