DECENTRALIZED AUTONOMOUS ORGANIZATION TO AUTOMATE GOVERNANCE
FINAL DRAFT - UNDER REVIEW

CHRISTOPH JENTZSCH
FOUNDER & CTO, SLOCK.IT
CHRISTOPH.JENTZSCH@SLOCK.IT

Abstract. This paper describes the first implementation of Decentralized Autonomous Organization (DAO) code to automate organizational governance and decision-making. The code can be used by individuals working together collaboratively outside of a traditional corporate form. It can also be used by a registered corporate entity to automate formal governance rules contained in corporate bylaws or imposed by law. First the DAO concept is described, then minority rights is discussed, and a solution to a "robbing the minority" attack vector is proposed. Finally, a practical implementation of a first generation DAO entity is provided using smart contracts written in Solidity on the Ethereum blockchain.

1. Introduction

Corporate entities of all kinds are governed by rules that describe permitted and proscribed conduct. These rules may exist as private contracts (like bylaws or shareholder agreements) between corporate owners. They may also be imposed by law in addition to or in the absence of a written agreement between participants.

Historically, corporations have only been able to act through people (or through corporate entities that were themselves ultimately controlled by people). This presents two simple and fundamental problems. Whatever a private contract or public law require: (1) people do not always follow the rules and (2) people do not always agree what the rules actually require. Collaboration without a corporate form does not solve these problems, necessarily, and it may introduce others. In the absence of a corporate form, an explicit written agreement is substituted for unclear informal "understandings" and the legal protections provided by a corporate form will not be available.

Rule-breaking within an organization not always obvious, and motives may not matter to stakeholders once their money has been lost. While bad behavior may make a corporation or its management civilly or criminally liable, punishment can come as little comfort to an investor who has already lost their money.

Crowdfunding (?) amplifies the problem. On the one hand, it has made it easier for small contributors to invest in large projects, and it has also made it possible for entrepreneurs to receive financial support that might not have been easily available in the past. On the other hand, small investors remain vulnerable to financial mismanagement or outright fraud, and because they have a small stake in a venture, they may lack power to identify problems, participate in governance decisions, or to easily recover their investment (??). At the same time, corporate leadership and management may be accused of malfeasance or mismanagement when they believe that they have acted in good faith, based on their understanding of their obligations and interpretation of applicable rules.

This paper presents a potential solution using Ethereum, (??) a blockchain technology which integrates a Turing complete programming language with smart contract processing functionality. This paper illustrates a method that for the first time allows the creation of organizations in which (1) participants maintain direct real-time control of contributed funds and (2) governance rules are formalized, automated and enforced using software. Specifically, standard smart contract code (??) has been written that can be used to form a Decentralized Autonomous Organization (DAO) on the Ethereum blockchain. This paper explains how a DAO's code works, focusing on some basic formation and governance features, including structure, creation and voting rights.

First a DAO's Creation Phase and basic functionality are described. Then minority owner rights are discussed and a solution to the "Majority Robbing the Minority Attack" problem is proposed: the "DAO split." The smart contract code is then explored in detail, and conclude with an explanation and detailed specification of the "DAO split."

The code for the smart contracts is located at: `https://github.com/slockit/DAO/`

A word of caution, at the outset: the legal status of DAOs remains the subject of active and vigorous debate and discussion. Not everyone shares the same definition. Some have said that they are autonomous code and can operate independently of legal systems; others have said that they must be owned or operate by humans or human created entities. There will be many uses cases, and the DAO code will develop over time. Ultimately, how a DAO functions and its legal status will depend on many factors, including how DAO code is used, where it is used, and who uses it. This paper does not speculate about the legal status of DAOs worldwide. This paper is not intended to offer legal advice or conclusions. Anyone who uses DAO code will do so at their own risk.

2. Dao Concept

DAO code is written in the "Solidity" programming language. A DAO is activated by deployment on the Ethereum blockchain.

Once deployed, a DAO's code requires "ether" to engage in transactions on Ethereum. Ether is the digital fuel that powers the Ethereum network. Without ether, a

DAO can not do anything so a DAO's first order of business is to receive ether. After a DAO's code is deployed, ether may be sent to the DAO's smart contract address during an initial Creation Phase which is defined in the DAO's code.

In exchange for ether, a DAO's code creates tokens that are assigned to the account of the person who sent the ether. The token grants its holder voting and ownership rights. The number of tokens created is proportional to the amount of ether transferred. Token price varies over time (see section 5). Token ownership is freely transferable on the Ethereum blockchain, when the Creation Phase has ended.

A minimum DAO Creation goal and Creation Phase time-period are set as parameters in a DAO's code at the time of deployment. If the minimum DAO Creation goal is not reached at the close of the Creation Phase, all ether is returned. After the Creation Phase has ended, the total ether raised is denoted by Ξ_{raised} and the total amount of tokens created is denoted by T_{total}.

A DAO stores ether and other Ethereum based tokens and transmits them based on the DAO's code. It does not do much else. It cannot build a product, write code or develop hardware. It requires a "Contractor" to accomplish these and other goals. A DAO selects a Contractor by accepting a Contractor's proposal.

Any DAO Token Holder may become a Contractor by submitting proposals to use a DAO's ether, denoted by Ξ_{transfer}. If a proposal is approved, the DAO transmits ether to a smart contract representing the proposed project. Such smart contracts can be parameterized and enable a DAO to interact with and influence the project it chose to support. An example of such an agreement between a DAO and a project to be funded can be found in the appendix (A.4).

Members of a DAO cast votes weighted by the amount of tokens they control. Tokens are divisible, indistinguishable and can easily be transferred between accounts. Within the contracts, the individual actions of members, cannot be directly determined. There is a set time frame t_p to debate and vote on any given proposal. In our example, this time frame is set by the creator of the proposal, and is required to be at least two weeks for a regular proposal.

After t_p has passed, any token holder can call a function in the DAO contract that will verify that the majority voted in favor of the proposal and that quorum was reached; the function will execute the proposal if this is the case. If this is not the case, the proposal will be closed.

The minimum quorum represents the minimum number of tokens required for a vote to be valid, is denoted by q_{min}, and calculated as follows:

$$q_{min} = \frac{T_{\text{total}}}{d} + \frac{\Xi_{\text{transfer}} \cdot T_{\text{total}}}{3 \cdot (\Xi_{DAO} + R_{\text{DAO}})} \tag{1}$$

Where d is the `minQuorumDivisor`. This parameter's default value is 5, but it will double in the case the quorum has not been met for over a year. Ξ_{DAO} is the amount of ether owned by a DAO and R_{DAO} is the amount of reward tokens owned by this DAO, as explained in section 7 (also see `rewardToken` in A.3). The sum $\Xi_{DAO} + R_{\text{DAO}}$ is equal to the amount of ether used to Create DAO tokens plus the rewards received or said another way, the total amount of ether a DAO has ever received.

This means, initially, a quorum of 20% of all tokens is required for any proposal to pass. In the event Ξ_{transfer} equals the amount of ether a DAO has ever received, then a quorum of 53.33% is required.

In order to prevent "proposal spam," a minimal deposit can be required to be paid when creating a proposal, which gets refunded if quorum is achieved. If quorum is not achieved, the DAO keeps the proposal deposit. The value of the proposal deposit can be changed from the default value by the DAO through another proposal.

3. Notation

Throughout this paper, Ξ always represents an amount of ether in its base unit wei. This is defined as 1 Wei $= 10^{-18}$ Ether (?). Similarly, DAO tokens are denoted with T and always represent the amount of DAO tokens in its base unit, defined as 10^{-16} DAO token.

4. Majority robs minority attack

Minority owner rights can be a problem in any corporate form. Minority rights may be protected or addressed by provisions in corporate governance documents or by statute or judge-made law. But many of these solutions fail because minority owners may lack voting rights or the ability to "vote with their feet" and easily retrieve their capital. This paper presents a solution to this problem in the DAO's code.

A problem every DAO has to mitigate is the ability for the majority to rob the minority by changing governance and ownership rules after DAO formation. For example, an attacker with 51% of the tokens, acquired either during the fueling period or created afterwards, could make a proposal to send all the funds to themselves. Since they would hold the majority of the tokens, they would always be able to pass their proposals.

To prevent this, the minority must always have the ability to retrieve their portion of the funds. Our solution is to allow a DAO to split into two. If an individual, or a group of token holders, disagree with a proposal and want to retrieve their portion of the ether before the proposal gets executed, they can submit and approve a special type of proposal to form a new DAO. The token holders that voted for this proposal can then split the DAO moving their portion of the ether to this new DAO, leaving the rest alone only able to spend their own ether.

This idea originates from a blog post by Vitalik Buterin (?).

A problem this simple fix doesn't address is voter apathy: some token holders might not be actively involved in their DAO and might not follow proposals closely. An attacker could use this to their advantage. Even though the minority has the chance to retrieve their funds and split the DAO, some could be unaware of the situation and fail to act. For a DAO to be considered safe, it is required that inactive token holders must also be protected from losing their ether. Our proposed solution is implemented by limiting each individual DAO to a single Curator. This Curator controls the list of addresses that can receive ether from the DAO, across all proposals. This gives the Curator of a DAO considerable power. To prevent the abuse of this power, it is possible for a DAO to vote for a new Curator, which may result in a split of the DAO as described above.

Any token holder can make a proposal to vote for a new Curator. In effect, even a single token holder is able to both retrieve their remaining portion of ether and maintain their right to any future rewards associated to their previous contribution, as these will be sent to the new DAO automatically. Rewards are defined as any ether received by a DAO generated from products the DAO funded so far and are explained in further detail in section 7.

The process of choosing a new Curator is as follows: Any token holder can submit a proposal for a new Curator. In this case, no proposal deposit is required, because an attacker could vote for an extremely high deposit, preventing any splits. The debating period for this proposal is 7 days. This is 7 days less than the minimum required for regular proposals, allowing anyone to retrieve their funds before a potentially malicious proposal goes through. There is no quorum requirement, so that every token holder has the ability to split into their own DAO. The debating period is used to discuss (on or off-chain) the new Curator and conduct a first vote that's non-binding. After this first vote, token holders can confirm its results or not. In the case a majority opts to keep the original Curator, the minority can either keep the original Curator in order to avoid a split, or inversely they can confirm their vote for a new Curator and move their portion of the ether to a new DAO.

5. Token Price

DAO Token Creation rate decreases over time. This reflects an assumption that early acts of DAO Creation have greater risk, as they may have less information about the potential success of the DAO and do not know whether what contribution will lead to full fueling of the DAO. The DAO described in this paper has the following Creation schedule:

(2)
$$P(t) = \begin{cases} 1 & \text{if} \quad t < t_c - 2 \cdot \mathrm{w} \\ 1 + 0.05 \cdot m(t) & \text{if} \quad t_c - 2 \cdot \mathrm{w} \leqslant t < t_c - 4 \cdot \mathrm{d} \\ 1.5 & \text{otherwise} \end{cases}$$

with the multiplier m defined as:

$$m(t) = (t - (t_c - 2 \cdot \mathrm{w}))/\mathrm{d} \tag{3}$$

Here t is the unix time in seconds, t_c is the closing time of the fueling period (see A.2 `closingTime`), w is a week in seconds and d a day in seconds. Hence the number of tokens (in its base unit) each person Creates is calculated as: $P(t) \cdot \Xi_c$. Here Ξ_c stands for the amount of ether sent to fuel a DAO, denoted in wei. This results in a constant Creation rate in the beginning, until 2 weeks before the end of the DAO Creation Phase. At this time the amount of ether required to Create DAO tokens increases daily by 0.05 Ξ_c per base unit of DAO token. Until 4 days before the closing time when there will be a constant Creation rate of 1.5 Ξ_c per base unit of DAO token.

Creation rate decreases during the Creation Phase could lead to a situation where a single contributor, having Created DAO tokens at the initial Creation rate, could split the DAO immediately after the Creation Phase ends, thereby receiving more ether than they put in due to other contributors having fueled a DAO at a higher Creation rate (?). In order to avoid that possibility, all ether that is used to fuel a DAO above the initial Creation rate, will be sent to an extra account. Denoted as `extraBalance` in A.2. This ether can be sent back to the DAO through a proposal after the DAO has spent at least this amount of ether. This rule is implemented in the internal function `isRecipientAllowed` in section 6.3.

6. Contracts

This section will detail the smart contracts implementing the aforementioned concept. The contracts are written in the programming language Solidity (?). Each contract has member variables and functions which can be externally called by sending a transaction to the Ethereum network with the DAO contract address as the recipient and the method ID (optional with parameters) as data. This section will discuss the meaning of the variables and the functions in detail.

The main contract is called 'DAO'. It defines the inner workings of the DAO and it derives the member variables and functions from 'Token' and 'TokenCreation'. Token defines the inner workings of the DAO Token and TokenCreation defines how the DAO token is created by fueling the DAO with ether. In addition to these three contracts, there is the 'ManagedAccount' contract, which acts as a helper contract to store the rewards which are to be distributed to the token holders and the `extraBalance` (see section 5). The contract 'SampleOffer' (A.4) is an example of what a proposal from a contractor to the DAO could look like.

6.1. **Token.**

```
contract TokenInterface {
    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;
    uint256 public totalSupply;
    function balanceOf(address _owner) constant returns (uint256 balance);
    function transfer(address _to, uint256 _amount) returns (bool success);
    function transferFrom(address _from, address _to, uint256 _amount) returns (bool success);
    function approve(address _spender, uint256 _amount) returns (bool success);
    function allowance(address _owner, address _spender) constant returns (uint256 remaining);

    event Transfer(address indexed _from, address indexed _to, uint256 _amount);
    event Approval(address indexed _owner, address indexed _spender, uint256 _amount);
}
```

Above is the interface of the Token contract. The interfaces of these contracts are used in the text of this document to give a simple overview of the functions and variables used in the contract, the full implementation can be found in the appendix (A.1). This contract represents the standard token as described here: `https://github.com/ethereum/wiki/Standardized_Contract_APIs`, and the contract `https://github.com/ConsenSys/Tokens/blob/master/Token_Contracts/contracts/StandardToken.sol` was used as a starting point for the contracts creation.

The map `balances` stores the number of DAO tokens which are controlled by an address. All contracts which derive from `TokenInterface` can directly modify this map, but only 4 functions do so: `createTokenProxy`, `transfer` , `transferFrom` and `splitDAO`.

The map `allowed` is used to track the previously specified addresses that are allowed to send tokens on someone else's behalf.

The integer `totalSupply` is the total number of DAO tokens in existence. The `public` keyword creates a function with the same name as the variable which returns its value so that it is publically available.

The function `balanceOf` returns the balance of the specified address.

The function `transfer` is used to send token from the sender of the message to another address.

The function `transferFrom` is used to transfer tokens on behalf of someone else who has approved the transfer in advance using the `approve` function.

The function `approve` can be used by the DAO token owner to specify a certain `spender` to transfer a specified `value` from their account using the `transferFrom` function. To check whether a certain address is allowed to spend DAO tokens on behalf of someone else, the `allowance` function can be used, which returns the number of tokens which can be spent by the `spender`. This is similar to writing a check.

The event `Transfer` is used to inform lightweight clients about changes in `balances`.

The event `Approval` is used to inform lightweight clients about changes in `allowed`.

6.2. TokenCreation.

```
contract TokenCreationInterface {
    uint public closingTime;
    uint public minTokensToCreate;
    bool public isFueled;
    address public privateCreation;
    ManagedAccount extraBalance;
    mapping (address => uint256) weiGiven;

    function TokenCreation(uint _minTokensToCreate, uint _closingTime);
    function createTokenProxy(address _tokenHolder) returns (bool success);
    function refund();
    function divisor() returns (uint divisor);

    event FuelingToDate(uint value);
    event CreatedToken(address indexed to, uint amount);
    event Refund(address indexed to, uint value);
}
```

Above is the interface of the `TokenCreation` contract (A.2).

The integer `closingTime` is the (unix) time at which the Creation Phase ends.

The integer `minTokensToCreate` is the number of wei-equivalent tokens which are needed to be created by the DAO in order to be considered fueled.

The boolean `isFueled` is true if DAO has reached its minimum fueling goal, false otherwise.

The address `privateCreation` is used for DAO splits - if it is set to 0, then it is a public Creation, otherwise, only the address stored in `privateCreation` is allowed to create tokens.

The managed account (A.5) `extraBalance` is used to hold the excess ether which is received after the Creation rate is decreased during the Creation Phase. Anything that has been paid above the initial price goes to this account.

The map `weiGiven` stores the amount of wei given by each contributor during the Creation Phase and is only used to refund the contributors if the Creation Phase does not reach its fueling goal.

The constructor `TokenCreation` initiates the Creation Phase with the arguments `minTokensToCreate`, `closingtime` and `privateCreation`, which will be set in the constructor of the DAO contract (A.3) which is only executed once, when the DAO is deployed. In order to interact with the contract the following functions can be used:

createTokenProxy. This function creates one unit of the DAO tokens minimum denomination for every wei sent. The price is calculated as

$$\Xi_c \cdot 20/\text{divisor}$$

Here Ξ_c is the amount of wei given in order to create tokens, and `divisor` is calculated depending on the time: $20/P(t)$, as described in section 5. The parameter `tokenHolder` defines the receiver of the newly minted tokens.

refund. This function can be called by any contributor to receive their wei back if the Creation Phase failed to meet its fueling goal.

divisor. This function is used to calculate the price of the token during the Creation Phase in the function `createTokenProxy`.

The events `FuelingToDate`, `CreatedToken` and `Refund` are used to inform lightweight clients of the status of the Creation Phase.

6.3. DAO.

```
contract DAOInterface {
    Proposal[] public proposals;
    uint minQuorumDivisor;
    uint lastTimeMinQuorumMet;
    address public curator;
    address[] public allowedRecipients;
    mapping (address => uint) public rewardToken;
    uint public totalRewardToken;
    ManagedAccount public rewardAccount;
    ManagedAccount public DAOrewardAccount;
    mapping (address => uint) public paidOut;
    mapping (address => uint) public DAOpaidOut;
    mapping (address => uint) public blocked;
    uint public proposalDeposit;
    uint sumOfProposalDeposits;
    DAO_Creator public daoCreator;

    struct Proposal {
        address recipient;
        uint amount;
        string description;
        uint votingDeadline;
        bool open;
        bool proposalPassed;
        bytes32 proposalHash;
        uint proposalDeposit;
        bool newCurator;
        SplitData[] splitData;
        uint yea;
        uint nay;
        mapping (address => bool) votedYes;
        mapping (address => bool) votedNo;
        address creator;
    }

    struct SplitData {
        uint splitBalance;
        uint totalSupply;
        uint rewardToken;
        DAO newDAO;
    }

    modifier onlyTokenholders {}

    function DAO(
        address _curator,
        DAO_Creator _daoCreator,
        uint _proposalDeposit,
        uint _minTokensToCreate,
        uint _closingTime,
        address _privateCreation
    )
    function () returns (bool success);
    function receiveEther() returns(bool);
    function newProposal(
        address _recipient,
        uint _amount,
        string _description,
        bytes _transactionData,
        uint _debatingPeriod,
        bool __newCurator
    ) onlyTokenholders returns (uint _proposalID);
    function checkProposalCode(
        uint _proposalID,
        address _recipient,
        uint _amount,
        bytes _transactionData
    ) constant returns (bool _codeChecksOut);
    function vote(
        uint _proposalID,
        bool _supportsProposal
    ) onlyTokenholders returns (uint _voteID);
    function executeProposal(
        uint _proposalID,
        bytes _transactionData
    ) returns (bool _success);
    function splitDAO(
        uint _proposalID,
        address _newCurator
    ) returns (bool _success);
    function newContract(address _newContract);
    function changeAllowedRecipients(address _recipient, bool _allowed) external returns (bool _success);
    function changeProposalDeposit(uint _proposalDeposit) external;
    function retrieveDAOReward(bool _toMembers) external returns (bool _success);
    function getMyReward() returns(bool _success);
    function withdrawRewardFor(address _account) returns(bool _success);
    function transferWithoutReward(address _to, uint256 _amount) returns (bool success);
    function transferFromWithoutReward(
        address _from,
        address _to,
        uint256 _amount
    ) returns (bool success);
    function halveMinQuorum() returns (bool _success);
    function numberOfProposals() constant returns (uint _numberOfProposals);
    function getNewDAOAddress(uint _proposalID) constant returns (address _newDAO);
    function isBlocked(address _account) internal returns (bool);
    function unblockMe() returns (bool);

    event ProposalAdded(
        uint indexed proposalID,
        address recipient,
        uint amount,
        bool newCurator,
        string description
    );
    event Voted(uint indexed proposalID, bool position, address indexed voter);
    event ProposalTallied(uint indexed proposalID, bool result, uint quorum);
    event NewCurator(address indexed _newCurator);
    event AllowedRecipientAdded(address indexed _recipient);
}
```

The original contract used as a starting point for the DAO was: `http://chriseth.github.io/browser-solidity/?gist=192371538cf5e43e6dab` as described in `https://blog.ethereum.org/2015/12/04`. The main feature added is the splitting mechanism and all that comes with it. This section will define the member variables and functions from the above smart contract one at a time.

The array `proposals` holds all the proposals ever made.

The integer `minQuorumDivisor` is used to calculate the quorum needed for a proposal to pass. It is set to 5, but will double in the case a quorum has not been reached for over a year.

The integer `lastTimeMinQuorumMet` keeps track of the last time the quorum was reached.

The address `curator` is set at the creation of the DAO and defines the Curator.

The list `allowedRecipients` is commonly referred to as the whitelist. The DAO can only send transactions to itself, the `curator`, `extraBalance` and addresses in the whitelist. Only the `curator` can add/remove addresses to/from the whitelist.

The map `rewardToken` tracks the addresses that are owed rewards generated by the products of the DAO. Those addresses can only be DAOs.

The integer `totalRewardToken` tracks the amount of reward tokens in existence.

The variable `rewardAccount` is of the type `ManagedAccount` , which will be discussed in A.5. It is used to manage the rewards which are to be distributed to the DAO Token Holders.

Similar, `DAOrewardAccount` is also of the type `ManagedAccount`. This account is used to receive all rewards from projects funded by the DAO. It will then redistribute them amongst all splitted DAOs as well as itself using the function `retrieveDAOReward`.

The map `paidOut` is used to keep track how much wei a single token holder has already retrieved from `rewardAccount`.

Similar, the map `DAOpaidOut` is used to keep track how much a single DAO has already received from `DAOrewardAccount`.

The map `blocked` stores the addresses of the DAO Tokens that have voted and therefore are not allowed to be transferred until the vote has concluded. The address points to the proposal ID.

The integer `proposalDeposit` specifies the minimum deposit to be paid in wei for any proposal that does not include a change of the Curator.

The integer `sumOfProposalDeposits` is the sum of all proposal deposits of open proposals.

The contract `daoCreator` is used to create a new DAO with the same code as this DAO, used in the case of a split.

A single proposal has the parameters:

recipient: The address where the `amount` of wei will go to if the proposal is accepted.

amount: The amount of wei to transfer to `recipient` if the proposal is accepted.

description: A plain text description of the proposal.

votingDeadline: A unix timestamp, denoting the end of the voting period.

open: A boolean which is false if the votes have already been counted, true otherwise.

proposalPassed: A boolean which is true if a quorum has been achieved with the majority approving the proposal.

proposalHash: A hash to check validity of a proposal. Equal to `sha3(_recipient, _amount, _transactionData)`.

proposalDeposit: The deposit (in wei) the creator of a proposal has send to submit a proposal. It is taken from the `msg.value` of a `newProposal` call; its purpose is to prevent spam. Its minimal value is set when the contract is deployed as constructor parameter. But the creator of the proposal can send any amount above this for the deposit. The proposals will be sorted by the proposal deposit in the GUI, so in the case that a proposal is considered important, the creator of the proposal can deposit extra ether to advertise their proposal. The creator of the proposal will be refunded the entire deposit if quorum is reached, if it is not reached the deposit remains with the DAO.

newCurator: A boolean which is true if this proposal assigns a new Curator.

splitData: The data used to split the DAO. This data is gathered from proposals when they require a new Curator.

yea: Number of tokens in favor of the proposal.

nay: Number of tokens opposed to the proposal.

votedYes: Simple mapping to check if a token holder has voted for it.

votedNo: Simple mapping to check if a token holder has voted against it.

creator: The address of the token holder that created the proposal.

The split data structure is used to split the DAO. It contains:

splitBalance: The balance of the current DAO minus the proposal deposit at the time of split.

totalSupply: The total amount of DAO tokens in existence at the time of the split.

rewardToken: The amount of reward tokens owned by the original DAO at the time of split.

newDAO: The address of the new DAO contract (0 if not created yet).

Those are all the member variables which are stored in this smart contract on the blockchain. This information can at any time be read from the blockchain using an Ethereum client.

This section will discuss the functions of the DAO contract in detail. Many of the member variables that are used in this contract are defined in one of the other three contracts.

There is a special function which is called the constructor. It has the same name as the contract "DAO." This function is only executed once, when the DAO is created. In the DAO constructor, the following variables are set:

- `curator`
- `daoCreator`
- `proposalDeposit`
- `rewardAccount`
- `DAOrewardAccount`
- `minTokensToCreate`
- `closingTime`
- `privateCreation`
- `lastTimeMinQuorumMet`
- `minQuorumDivisor`
- `allowedRecipients`

In order to interact with the smart contract the following functions are used:

fallback function. The fallback function is a function without a specific name. It is called when the contract receives a transaction without data (a pure value transfer). There are no direct arguments for this function. The fallback function will call `createTokenProxy` passing the address of the sender as an argument during the Creation Phase. This will trigger the immediate creation of tokens. In order to protect users, this function will send the ether received after the end of the Creation Phase back to the sender for a time period of 40 days. After which this function is repurposed to receive ether as simple deposit to the DAO using the function `receiveEther`.

receiveEther. A simple function used to receive ether. It does nothing but return true when the DAO receives ether.

newProposal. This function is used to create a new proposal. The arguments of the function are:

recipient: The address of the recipient of the ether in the proposal (has to be the DAO address itself, the current Curator or an address on the whitelist `allowedRecipients`).

amount: The amount of wei to be sent in the proposed transaction.

description: A string describing the proposal.

transactionData: The data of the proposed transaction.

debatingPeriod: The amount of time to debate the proposal, at least 2 weeks for a normal proposal and at least 1 week for a new Curator proposal.

newCurator: A boolean defining whether this proposal is for a new Curator or not.

After checking the sanity of the proposal (see code), this function creates a proposal which is open for voting for a certain amount of time. The function will return a proposal ID which is used to vote.

checkProposalCode. This function is used to check that a certain proposal ID matches a certain transaction. The arguments of the function are:

proposalID: The proposal ID.

recipient: The address of the recipient of the proposed transaction.

amount: The amount of wei to be sent with the proposed transaction.

transactionData: The data of the proposed transaction.

If the `recipient`, the `amount` and the `transactionData` match the proposal ID, the function will return `true` , otherwise it will return `false`. This will be used to verify that the proposal ID matches what the DAO token holder thinks they are voting on.

vote. This function is used to vote on a proposal. The arguments of the function are:

proposalID: The proposal ID.

supportsProposal: A boolean Yes/No does the DAO token holder support the proposal

The function simply checks whether the sender has yet to vote and whether the proposal is still open for voting. If both requirements are met, it records the vote in the storage of the contract. The tokens used to vote will be blocked, meaning they can not be transferred until the proposal is closed. This is to avoid voting several times with different sender addresses.

executeProposal. This function can be called by anyone. It counts the votes, in order to check whether the quorum is met, and executes the proposal if it passed, unless it is a proposal for a new Curator, than it does nothing. The arguments of the function are:

proposalID: The proposal ID.

transactionData: The data of the proposed transaction

The function checks whether the voting deadline has passed and that the `transactionData` matches the proposal ID. Then it checks whether the quorum has been met (see Eq. 1) and if the proposal had a majority of support. If this is the case, it executes the proposal and refunds the proposal deposit. If the quorum has been achieved, but the proposal was declined by the majority of the voters, the proposal deposit is refunded and the proposal closes.

splitDAO. After a new Curator has been proposed, and the debating period in which the token holders could vote for or against the proposal has passed, this function is called by each of the DAO token holders that want to leave the current DAO and move to a new DAO with the proposed new Curator. This function creates a new DAO and moves a portion of the ether, as well as a portion of the reward tokens to the new DAO. The arguments are:

proposalID: The proposal ID.

newCurator: The address of the new Curator of the new DAO.

After a sanity check (see code), this function will create the new DAO if it hasnt already been created using the contract `daoCreator`, updates the split data stored in the proposal and stores the address of the new DAO in the split data. This function moves the portion of ether that belongs to the caller of this function in the original DAO to the new DAO. This ether amount is denoted by Ξ_{sender}, stated in wei and is calculated as follows:

$$\Xi_{\text{sender}} = \Xi_{\text{DAO}} \cdot T_{\text{sender}} / T_{\text{total}} \tag{4}$$

Here T_{sender} is the amount of tokens of the caller of the function and Ξ_{DAO} is the balance of the DAO at the time of the split. This will be used to effectively create tokens in the newly created DAO and fuel the new DAO just as the original DAO was fueled. In addition to the ether which is moved to the new DAO, the reward tokens R_{sender} are also transferred. They are calculated as follows:

$$R_{\text{sender}} = R_{\text{DAO}} \cdot T_{\text{sender}} / T_{\text{total}} \tag{5}$$

Where R_{DAO} is the amount of reward tokens owned by the original DAO at the time of the split. These tokens allow the new DAO to retrieve their portion of the reward using the `retrieveDAOReward` function of the original DAO. At the end of this process all original DAO tokens of the sender account are destroyed. It is important to notice that in all integer division descirbed above, there may be remainders which stay with the DAO.

newContract. This function can only be called by the DAO itself (through a proposal and a vote) and is used to move all remaining ether, as well as all rewardTokens to a new address. This is used to update the contract. The new address needs to be approved by the Curator.

transfer and transferFrom. These functions overload the functions defined in the `Token` contract. They do call `transfer / transferFrom` function in the `Token` contract, but they additionally transfer information about the already paid out rewards attached to the tokens being transferred using the `transferPaidOut` function.

transferPaidOut. This function is called when making any transfer of DAO tokens using `transfer` or `transferFrom` and it updates the array `paidOut` to track the amount of rewards which has been paid out already, P, and is calculated as follows:

$$P = P_{\text{from}} \cdot T_{\text{amount}} / T_{\text{from}} \tag{6}$$

Here P_{from} is the total amount of ether which has been paid out to the `from` address (the sender), T_{amount} is the amount of tokens to be transferred and T_{from} is the amount of tokens owned by the `from` address.

transferWithoutReward and transferFromWithoutReward. The same as `transfer` and `transferFrom`, but it calls `getMyReward` prior to that.

getMyReward. Calls `withdrawRewardFor` with the sender as the parameter. This is used to withdraw the portion of the rewards which belong to the sender from the `rewardAccount`.

withdrawRewardFor. This function is used to retrieve the portion of the rewards in the `rewardAccount` which belong to the address given as a parameter. The amount of ether Ξ_{reward} which is then sent to the DAO token holder that calls this function is:

$$\Xi_{\text{reward}} = T_{\text{sender}} \cdot \Xi_{\text{rewardAccount}} / T_{\text{total}} - \Xi_{\text{paidOut[sender]}} \tag{7}$$

Here $\Xi_{\text{rewardAccount}}$ is the total rewards ever received by the `rewardAccount` and $\Xi_{\text{paidOut[sender]}}$ is the total amount of wei which has already been paid out to the DAO token holder address, which is given as a parameter. The reward tokens are further elaborated in section 8.

retrieveDAOReward. This function, when called by a DAO, sends the rewards which belong to this DAO from `DAOrewardAccount` to either the DAO itself, or to the `rewardAccount` of the respective DAO in order to be distributed among its token holders, depending on the parameter `_toMembers`.

changeAllowedRecipients. This function can add/remove an address to/from the whitelist, `allowedRecipients`. It can only be executed by the Curator.

halveMinQuorum. When called, halves the minimum quorum in the case it has not been met for over 52 weeks, by doubling `minQuorumDivisor`. Also the curator can call this function without the 52 weeks limit, but not more than once every other week.

numberOfProposals. Returns the total number of proposals ever created.

getNewDAOAddress. This is just a helper function to read the address of a newly created ‘split DAO‘. It gets the proposal ID which was used for the split as input parameter and returns the address of the new DAO.

isBlocked. This function returns true when the address given as parameter is currently blocked to transfer tokens due to an ongoing vote it has participated in, otherwise it returns false. It also unblocks the tokens in the case the voting deadline of the proposal is over.

unblockMe. Calling `isBlocked` with the address of the sender.

changeProposalDeposit. This function changes the parameter `proposalDeposit`. It can only be called by the DAO through a transaction which was proposed and voted for by a majority of the token holders.

6.4. Managed Account.

```
 contract ManagedAccountInterface {
    address public owner;
    bool public payOwnerOnly;
    uint public accumulatedInput;

    function payOut(address _recipient, uint _amount) returns (bool);

    event PayOut(address _recipient, uint _amount);
}
```

This contract is used to manage the rewards and the `extraBalance` (as explained in section 5). It has two member variables:

The address `owner`, is the only address with permission to withdraw from that account (in our case the DAO) and send ether to another address using the `payOut` function.

The bool `payOwnerOnly` specifies whether the owner is the only address which can receive ether from this account.

The integer, `accumulatedInput`, represents the total sum of ether (in wei) which has been sent to this contract so far.

The fallback function is called when the contract receives a transaction without data (a pure value transfer). There are no direct arguments for this function. When it is called it counts the amount of ether it receives and stores it in `accumulatedInput`.

The function `payOut` can only be executed by the `owner` (in our case the DAO). It has two arguments: `recipient` and `amount`. It is used to send `amount` wei to a `recipient` and is called by `getMyReward` in the DAO contract.

7. Reward Tokens

This section gives a description of how reward tokens are implemented in this contract. Much of the information has already been explained, but it is restated here for clarity.

Reward tokens are used to divide the ether sent to DAOrewardAccount amongst the various DAOs that own reward tokens. Reward tokens are only transferred in the event of a DAO split or an update of the contract, they can never be owned by anything other than the original DAO or a fork of the original DAO that generated the reward tokens.

Reward tokens are generated when the DAO makes any transaction spending ether. When the DAOs products send ether back to the DAO, the ether is held within DAOrewardAccount. The DAO can use these rewards to fund new proposals or to fairly distribute the rewards to the reward token holders (using a proposal which gets voted on by the DAO token holders).

Then the token holders of the DAOs will be able to claim the ether they have earned for their contribution to the original DAO that issued the reward token. To do this the DAO retrieve its rewards by calling the retrieveDAOReward function, with the paramter _toMembers set to true, which send the rewards to the rewardAccount (a ManagedAccount contract) and keeps track of the payouts in DAOpaidOut. Then and only then will the token holders of the DAOs be able to call the getMyReward function and receive their ether.

These payouts are tracked by the map paidOut which keeps track of which token holders have claimed their fair portion of the rewards. This process guarantees that any DAO token holder whose ether was spent building a product will receive the rewards promised to them from that product even if they decide to split from the DAO.

8. Split

This section formally describes a few important parameters and their behavior during a split.

The total amount of DAO tokens totalSupply is defined as follows:

$$T_{\text{total}} = \sum_{i=0}^{2^{256}-1} T_i \tag{8}$$

Where T_i is the amount of DAO tokens owned by an address i (balances[i]). Note that 2^{256} is the total number of possible addresses in Ethereum. Similarly, the amount of reward tokens R_{total} is defined as follows:

$$R_{\text{total}} = \sum_{i=0}^{2^{256}-1} R_i = \sum_{p=0;p.proposalPassed=true}^{numProposals} p.amount \tag{9}$$

For every passed proposal that sends ether out of the DAO, an amount of reward tokens equal to the amount being spent (in wei) is created.

Lets assume that during the split, a fraction of DAO tokens, X, changes the Curator and leaves the DAO. The new DAO created receives $X \cdot \Xi_{\text{DAO pre}}$, a portion of the remaining ether from the original DAO.

$$\Xi_{\text{DAO post}} = (1 - X) \cdot \Xi_{\text{DAO pre}} \tag{10}$$

Here $\Xi_{\text{DAO pre}}$ is the ether balance of the original DAO before the split and $\Xi_{\text{DAO post}}$ is the ether balance of the original DAO after the split.

A portion of the reward tokens is transferred to the new DAO in a very similar manner:

$$R_{\text{DAO post}} = (1 - X) \cdot R_{\text{DAO pre}} \tag{11}$$

Here R_{DAO} is the amount of reward tokens owned by the DAO (prior to the first split 100% of all rewards tokens ever created are owned by the DAO).

$$R_{\text{newDAO}} = (X) \cdot R_{\text{DAO pre}} \tag{12}$$

The number of reward tokens owned by the new DAO are denoted by R_{newDAO}. The total amount of reward tokens R_{total} stays constant during the split, no reward tokens are ever destroyed.

The original DAO tokens of the accounts that confirmed the new Curator are destroyed. Hence:

$$T_{\text{total post}} = (1 - X) \cdot T_{\text{total pre}} \tag{13}$$

This process allows DAO token holders to retrieve their ether from the DAO at any time without losing out on any of the future rewards.They are entitled to receive even if they choose to leave the DAO.

9. Updates

Although the code of the contract specified at a certain address in the Ethereum blockchain can not be changed, there might still be a need for a single member or the DAO as a whole to change the contracts. Every single member can always split the DAO as described above and move their funds to a new DAO. From there they can move their funds to another new DAO with a new smart contract. But in order to use a new code for the complete DAO one can simply create a new DAO contract with all the needed features and deploy it on the blockchain, and make a proposal to call the newContract function with the address of the new contract as parameter. If accepted, the complete DAO moves to the new contract, meaning, all ether and reward tokens are transferred to the new contract. In order to use the same underlying DAO tokens there, one can use the approve function and give the new DAO the right to move the tokens. In the new contract this right should only be usable in restricted functions which are only callable by the owner of the tokens. Another option is to create new tokens in the new contract based on the token distribution in the old contract. This can also be achieved by a proof that the old tokens are destroyed (sending to the 0 address). This process allows for the DAO to maintain static immutable code on the Ethereum blockchain, while still being able to be updated if the need arises.

10. Acknowledgements

I want to thank Stephan Tual and Simon Jentzsch for fruitful discussions and corrections, as well as Gavin Wood and Christian Reitwiessner for a review of the contracts and the development of Solidity, the programing language used to write the contracts.

Special thanks goes to Yoichi Hirai and Lefteris Karapetsas for reviewing the smart contracts and making significant improvements.

I also want to thank Griff Green for reviewing and editing the paper.

Last but not least I want to thank our community which has given feedback, corrections and encouragement.

Appendix A. Contracts

A.1. Token.

```
contract TokenInterface {
    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;

    /// @return Total amount of tokens
    uint256 public totalSupply;

    /// @param _owner The address from which the balance will be retrieved
    /// @return The balance
    function balanceOf(address _owner) constant returns (uint256 balance);

    /// @notice Send `_amount` tokens to `_to` from `msg.sender`
    /// @param _to The address of the recipient
    /// @param _amount The amount of tokens to be transferred
    /// @return Whether the transfer was successful or not
    function transfer(address _to, uint256 _amount) returns (bool success);

    /// @notice Send `_amount` tokens to `_to` from `_from` on the condition it
    /// is approved by `_from`
    /// @param _from The address of the sender
    /// @param _to The address of the recipient
    /// @param _amount The amount of tokens to be transferred
    /// @return Whether the transfer was successful or not
    function transferFrom(address _from, address _to, uint256 _amount)
        returns (bool success);

    /// @notice `msg.sender` approves `_spender` to spend `_amount` tokens on
    /// its behalf
    /// @param _spender The address of the account able to transfer the tokens
    /// @param _amount The amount of tokens to be approved for transfer
    /// @return Whether the approval was successful or not
    function approve(address _spender, uint256 _amount) returns (bool success);

    /// @param _owner The address of the account owning tokens
    /// @param _spender The address of the account able to transfer the tokens
    /// @return Amount of remaining tokens of _owner that _spender is allowed
    /// to spend
    function allowance(address _owner, address _spender)
        constant
        returns (uint256 remaining);

    event Transfer(address indexed _from, address indexed _to, uint256 _amount);
    event Approval(
        address indexed _owner,
        address indexed _spender,
        uint256 _amount
    );
}


contract Token is TokenInterface {
    // Protects users by preventing the execution of method calls that
    // inadvertently also transferred ether
    modifier noEther() {if (msg.value > 0) throw; _}

    function balanceOf(address _owner) constant returns (uint256 balance) {
        return balances[_owner];
    }

    function transfer(address _to, uint256 _amount)
```

```
        noEther
        returns (bool success)
    {
        if (balances[msg.sender] >= _amount && _amount > 0) {
            balances[msg.sender] -= _amount;
            balances[_to] += _amount;
            Transfer(msg.sender, _to, _amount);
            return true;
        }
        else
           return false;
    }

    function transferFrom(address _from, address _to, uint256 _amount)
        noEther
        returns (bool success)
    {
        if (balances[_from] >= _amount
            && allowed[_from][msg.sender] >= _amount
            && _amount > 0
        ) {
            balances[_to] += _amount;
            balances[_from] -= _amount;
            allowed[_from][msg.sender] -= _amount;
            Transfer(_from, _to, _amount);
            return true;
        }
        else
            return false;
    }

    function approve(address _spender, uint256 _amount) returns (bool success) {
        allowed[msg.sender][_spender] = _amount;
        Approval(msg.sender, _spender, _amount);
        return true;
    }

    function allowance(address _owner, address _spender)
        constant
        returns (uint256 remaining)
    {
        return allowed[_owner][_spender];
    }
}
```

A.2. TokenCreation.

```
contract TokenCreationInterface {

    // End of token creation, in Unix time
    uint public closingTime;
    // Minimum fueling goal of the token creation, denominated in tokens to
    // be created
    uint public minTokensToCreate;
    // True if the DAO reached its minimum fueling goal, false otherwise
    bool public isFueled;
    // For DAO splits - if privateCreation is 0, then it is a public token
    // creation, otherwise only the address stored in privateCreation is
    // allowed to create tokens
    address public privateCreation;
    // hold extra ether which has been sent after the DAO token
    // creation rate has increased
    ManagedAccount public extraBalance;
    // tracks the amount of wei given from each contributor (used for refund)
    mapping (address => uint256) weiGiven;

    /// @dev Constructor setting the minimum fueling goal and the
    /// end of the Token Creation
    /// @param _minTokensToCreate Minimum required wei-equivalent tokens
    ///        to be created for a successful DAO Token Creation
    /// @param _closingTime Date (in Unix time) of the end of the Token Creation
    /// @param _privateCreation Zero means that the creation is public.  A
    /// non-zero address represents the only address that can create Tokens
    /// (the address can also create Tokens on behalf of other accounts)
    // This is the constructor: it can not be overloaded so it is commented out
    //  function TokenCreation(
        //  uint _minTokensTocreate,
        //  uint _closingTime,
        //  address _privateCreation
    //  );

    /// @notice Create Token with ‘_tokenHolder‘ as the initial owner of the Token
    /// @param _tokenHolder The address of the Tokens’s recipient
    /// @return Whether the token creation was successful
    function createTokenProxy(address _tokenHolder) returns (bool success);

    /// @notice Refund ‘msg.sender‘ in the case the Token Creation did
    /// not reach its minimum fueling goal
    function refund();

    /// @return The divisor used to calculate the token creation rate during
    /// the creation phase
    function divisor() returns (uint divisor);

    event FuelingToDate(uint value);
    event CreatedToken(address indexed to, uint amount);
    event Refund(address indexed to, uint value);
}


contract TokenCreation is TokenCreationInterface, Token {
    function TokenCreation(
        uint _minTokensToCreate,
        uint _closingTime,
        address _privateCreation) {

        closingTime = _closingTime;
        minTokensToCreate = _minTokensToCreate;
        privateCreation = _privateCreation;
        extraBalance = new ManagedAccount(address(this), true);
    }

    function createTokenProxy(address _tokenHolder) returns (bool success) {
        if (now < closingTime && msg.value > 0
            && (privateCreation == 0 || privateCreation == msg.sender)) {

            uint token = (msg.value * 20) / divisor();
            extraBalance.call.value(msg.value - token)();
            balances[_tokenHolder] += token;
            totalSupply += token;
            weiGiven[_tokenHolder] += msg.value;
            CreatedToken(_tokenHolder, token);
            if (totalSupply >= minTokensToCreate && !isFueled) {
                isFueled = true;
                FuelingToDate(totalSupply);
            }
            return true;
        }
        throw;
    }

    function refund() noEther {
        if (now > closingTime && !isFueled) {
            // Get extraBalance - will only succeed when called for the first time
            extraBalance.payOut(address(this), extraBalance.accumulatedInput());

            // Execute refund
            if (msg.sender.call.value(weiGiven[msg.sender])()) {
                Refund(msg.sender, weiGiven[msg.sender]);
                totalSupply -= balances[msg.sender];
                balances[msg.sender] = 0;
                weiGiven[msg.sender] = 0;
            }
        }
    }

    function divisor() returns (uint divisor) {
        // The number of (base unit) tokens per wei is calculated
        // as ‘msg.value‘ * 20 / ‘divisor‘
        // The fueling period starts with a 1:1 ratio
        if (closingTime - 2 weeks > now) {
            return 20;
        // Followed by 10 days with a daily creation rate increase of 5%
        } else if (closingTime - 4 days > now) {
            return (20 + (now - (closingTime - 2 weeks)) / (1 days));
        // The last 4 days there is a constant creation rate ratio of 1:1.5
        } else {
            return 30;
        }
    }
}
```

A.3. DAO.

```
contract DAOInterface {

    // The amount of days for which people who try to participate in the
    // creation by calling the fallback function will still get their ether back
    uint constant creationGracePeriod = 40 days;
    // The minimum debate period that a generic proposal can have
    uint constant minProposalDebatePeriod = 2 weeks;
    // The minimum debate period that a split proposal can have
    uint constant minSplitDebatePeriod = 1 weeks;
    // Period of days inside which it’s possible to execute a DAO split
    uint constant splitExecutionPeriod = 27 days;
    // Period of time after which the minimum Quorum is halved
    uint constant quorumHalvingPeriod = 25 weeks;
    // Period after which a proposal is closed
    // (used in the case ‘executeProposal‘ fails because it throws)
    uint constant executeProposalPeriod = 10 days;
    // Denotes the maximum proposal deposit that can be given. It is given as
    // a fraction of total Ether spent plus balance of the DAO
    uint constant maxDepositDivisor = 100;

    // Proposals to spend the DAO’s ether or to choose a new Curator
    Proposal[] public proposals;
    // The quorum needed for each proposal is partially calculated by
    // totalSupply / minQuorumDivisor
    uint public minQuorumDivisor;
    // The unix time of the last time quorum was reached on a proposal
    uint  public lastTimeMinQuorumMet;
```

```
// Address of the curator
address public curator;
// The whitelist: List of addresses the DAO is allowed to send ether to
mapping (address => bool) public allowedRecipients;

// Tracks the addresses that own Reward Tokens. Those addresses can only be
// DAOs that have split from the original DAO. Conceptually, Reward Tokens
// represent the proportion of the rewards that the DAO has the right to
// receive. These Reward Tokens are generated when the DAO spends ether.
mapping (address => uint) public rewardToken;
// Total supply of rewardToken
uint public totalRewardToken;

// The account used to manage the rewards which are to be distributed to the
// DAO Token Holders of this DAO
ManagedAccount public rewardAccount;

// The account used to manage the rewards which are to be distributed to
// any DAO that holds Reward Tokens
ManagedAccount public DAOrewardAccount;

// Amount of rewards (in wei) already paid out to a certain DAO
mapping (address => uint) public DAOpaidOut;

// Amount of rewards (in wei) already paid out to a certain address
mapping (address => uint) public paidOut;
// Map of addresses blocked during a vote (not allowed to transfer DAO
// tokens). The address points to the proposal ID.
mapping (address => uint) public blocked;

// The minimum deposit (in wei) required to submit any proposal that is not
// requesting a new Curator (no deposit is required for splits)
uint public proposalDeposit;

// the accumulated sum of all current proposal deposits
uint sumOfProposalDeposits;

// Contract that is able to create a new DAO (with the same code as
// this one), used for splits
DAO_Creator public daoCreator;

// A proposal with ‘newCurator == false‘ represents a transaction
// to be issued by this DAO
// A proposal with ‘newCurator == true‘ represents a DAO split
struct Proposal {
    // The address where the ‘amount‘ will go to if the proposal is accepted
    // or if ‘newCurator‘ is true, the proposed Curator of
    // the new DAO).
    address recipient;
    // The amount to transfer to ‘recipient‘ if the proposal is accepted.
    uint amount;
    // A plain text description of the proposal
    string description;
    // A unix timestamp, denoting the end of the voting period
    uint votingDeadline;
    // True if the proposal’s votes have yet to be counted, otherwise False
    bool open;
    // True if quorum has been reached, the votes have been counted, and
    // the majority said yes
    bool proposalPassed;
    // A hash to check validity of a proposal
    bytes32 proposalHash;
    // Deposit in wei the creator added when submitting their proposal. It
    // is taken from the msg.value of a newProposal call.
    uint proposalDeposit;
    // True if this proposal is to assign a new Curator
    bool newCurator;
    // Data needed for splitting the DAO
    SplitData[] splitData;
    // Number of Tokens in favor of the proposal
    uint yea;
    // Number of Tokens opposed to the proposal
    uint nay;
    // Simple mapping to check if a shareholder has voted for it
    mapping (address => bool) votedYes;
    // Simple mapping to check if a shareholder has voted against it
    mapping (address => bool) votedNo;
    // Address of the shareholder who created the proposal
    address creator;
}

// Used only in the case of a newCurator proposal.
struct SplitData {
    // The balance of the current DAO minus the deposit at the time of split
    uint splitBalance;
    // The total amount of DAO Tokens in existence at the time of split.
    uint totalSupply;
    // Amount of Reward Tokens owned by the DAO at the time of split.
    uint rewardToken;
    // The new DAO contract created at the time of split.
    DAO newDAO;
}

// Used to restrict access to certain functions to only DAO Token Holders
modifier onlyTokenholders {}

/// @dev Constructor setting the Curator and the address
/// for the contract able to create another DAO as well as the parameters
/// for the DAO Token Creation
/// @param _curator The Curator
/// @param _daoCreator The contract able to (re)create this DAO
/// @param _proposalDeposit The deposit to be paid for a regular proposal
/// @param _minTokensToCreate Minimum required wei-equivalent tokens
///        to be created for a successful DAO Token Creation
/// @param _closingTime Date (in Unix time) of the end of the DAO Token Creation
/// @param _privateCreation If zero the DAO Token Creation is open to public, a
/// non-zero address means that the DAO Token Creation is only for the address
// This is the constructor: it can not be overloaded so it is commented out
//  function DAO(
    //  address _curator,
    //  DAO_Creator _daoCreator,
    //  uint _proposalDeposit,
    //  uint _minTokensToCreate,
    //  uint _closingTime,
    //  address _privateCreation
//  );

/// @notice Create Token with ‘msg.sender‘ as the beneficiary
/// @return Whether the token creation was successful
function () returns (bool success);


/// @dev This function is used to send ether back
/// to the DAO, it can also be used to receive payments that should not be
/// counted as rewards (donations, grants, etc.)
/// @return Whether the DAO received the ether successfully
function receiveEther() returns(bool);

/// @notice ‘msg.sender‘ creates a proposal to send ‘_amount‘ Wei to
/// ‘_recipient‘ with the transaction data ‘_transactionData‘. If
/// ‘_newCurator‘ is true, then this is a proposal that splits the
/// DAO and sets ‘_recipient‘ as the new DAO’s Curator.
/// @param _recipient Address of the recipient of the proposed transaction
/// @param _amount Amount of wei to be sent with the proposed transaction
/// @param _description String describing the proposal
/// @param _transactionData Data of the proposed transaction
/// @param _debatingPeriod Time used for debating a proposal, at least 2
/// weeks for a regular proposal, 10 days for new Curator proposal
/// @param _newCurator Bool defining whether this proposal is about
/// a new Curator or not
/// @return The proposal ID. Needed for voting on the proposal
function newProposal(
    address _recipient,
    uint _amount,
    string _description,
    bytes _transactionData,
    uint _debatingPeriod,
    bool _newCurator
) onlyTokenholders returns (uint _proposalID);

/// @notice Check that the proposal with the ID ‘_proposalID‘ matches the
/// transaction which sends ‘_amount‘ with data ‘_transactionData‘
/// to ‘_recipient‘
/// @param _proposalID The proposal ID
/// @param _recipient The recipient of the proposed transaction
/// @param _amount The amount of wei to be sent in the proposed transaction
/// @param _transactionData The data of the proposed transaction
/// @return Whether the proposal ID matches the transaction data or not
function checkProposalCode(
    uint _proposalID,
    address _recipient,
    uint _amount,
    bytes _transactionData
) constant returns (bool _codeChecksOut);

/// @notice Vote on proposal ‘_proposalID‘ with ‘_supportsProposal‘
/// @param _proposalID The proposal ID
/// @param _supportsProposal Yes/No - support of the proposal
/// @return The vote ID.
function vote(
    uint _proposalID,
    bool _supportsProposal
) onlyTokenholders returns (uint _voteID);

/// @notice Checks whether proposal ‘_proposalID‘ with transaction data
/// ‘_transactionData‘ has been voted for or rejected, and executes the
/// transaction in the case it has been voted for.
/// @param _proposalID The proposal ID
/// @param _transactionData The data of the proposed transaction
/// @return Whether the proposed transaction has been executed or not
function executeProposal(
    uint _proposalID,
    bytes _transactionData
) returns (bool _success);

/// @notice ATTENTION! I confirm to move my remaining ether to a new DAO
/// with ‘_newCurator‘ as the new Curator, as has been
/// proposed in proposal ‘_proposalID‘. This will burn my tokens. This can
/// not be undone and will split the DAO into two DAO’s, with two
/// different underlying tokens.
/// @param _proposalID The proposal ID
/// @param _newCurator The new Curator of the new DAO
/// @dev This function, when called for the first time for this proposal,
/// will create a new DAO and send the sender’s portion of the remaining
/// ether and Reward Tokens to the new DAO. It will also burn the DAO Tokens
/// of the sender.
function splitDAO(
    uint _proposalID,
    address _newCurator
) returns (bool _success);

/// @dev can only be called by the DAO itself through a proposal
/// updates the contract of the DAO by sending all ether and rewardTokens
/// to the new DAO. The new DAO needs to be approved by the Curator
/// @param _newContract the address of the new contract
function newContract(address _newContract);


/// @notice Add a new possible recipient ‘_recipient‘ to the whitelist so
/// that the DAO can send transactions to them (using proposals)
/// @param _recipient New recipient address
/// @dev Can only be called by the current Curator
/// @return Whether successful or not
function changeAllowedRecipients(address _recipient, bool _allowed) external returns (bool _success);


/// @notice Change the minimum deposit required to submit a proposal
/// @param _proposalDeposit The new proposal deposit
/// @dev Can only be called by this DAO (through proposals with the
/// recipient being this DAO itself)
function changeProposalDeposit(uint _proposalDeposit) external;

/// @notice Move rewards from the DAORewards managed account
/// @param _toMembers If true rewards are moved to the actual reward account
///                   for the DAO. If not then it’s moved to the DAO itself
/// @return Whether the call was successful
function retrieveDAOReward(bool _toMembers) external returns (bool _success);

/// @notice Get my portion of the reward that was sent to ‘rewardAccount‘
/// @return Whether the call was successful
function getMyReward() returns(bool _success);

/// @notice Withdraw ‘_account‘’s portion of the reward from ‘rewardAccount‘
/// to ‘_account‘’s balance
/// @return Whether the call was successful
function withdrawRewardFor(address _account) internal returns (bool _success);

/// @notice Send ‘_amount‘ tokens to ‘_to‘ from ‘msg.sender‘. Prior to this
/// getMyReward() is called.
/// @param _to The address of the recipient
/// @param _amount The amount of tokens to be transfered
/// @return Whether the transfer was successful or not
function transferWithoutReward(address _to, uint256 _amount) returns (bool success);

/// @notice Send ‘_amount‘ tokens to ‘_to‘ from ‘_from‘ on the condition it
/// is approved by ‘_from‘. Prior to this getMyReward() is called.
/// @param _from The address of the sender
/// @param _to The address of the recipient
/// @param _amount The amount of tokens to be transfered
/// @return Whether the transfer was successful or not
function transferFromWithoutReward(
        address _from,
        address _to,
        uint256 _amount
    ) returns (bool success);

    /// @notice Doubles the 'minQuorumDivisor' in the case quorum has not been
    /// achieved in 52 weeks
    /// @return Whether the change was successful or not
    function halveMinQuorum() returns (bool _success);

    /// @return total number of proposals ever created
    function numberOfProposals() constant returns (uint _numberOfProposals);

    /// @param _proposalID Id of the new curator proposal
    /// @return Address of the new DAO
    function getNewDAOAddress(uint _proposalID) constant returns (address _newDAO);

    /// @param _account The address of the account which is checked.
    /// @return Whether the account is blocked (not allowed to transfer tokens) or not.
    function isBlocked(address _account) internal returns (bool);

    /// @notice If the caller is blocked by a proposal whose voting deadline
    /// has exprired then unblock him.
    /// @return Whether the account is blocked (not allowed to transfer tokens) or not.
    function unblockMe() returns (bool);

    event ProposalAdded(
        uint indexed proposalID,
        address recipient,
        uint amount,
        bool newCurator,
        string description
    );
    event Voted(uint indexed proposalID, bool position, address indexed voter);
    event ProposalTallied(uint indexed proposalID, bool result, uint quorum);
    event NewCurator(address indexed _newCurator);
    event AllowedRecipientChanged(address indexed _recipient, bool _allowed);
}

// The DAO contract itself
contract DAO is DAOInterface, Token, TokenCreation {

    // Modifier that allows only shareholders to vote and create new proposals
    modifier onlyTokenholders {
        if (balanceOf(msg.sender) == 0) throw;
            _
    }

    function DAO(
        address _curator,
        DAO_Creator _daoCreator,
        uint _proposalDeposit,
        uint _minTokensToCreate,
        uint _closingTime,
        address _privateCreation
    ) TokenCreation(_minTokensToCreate, _closingTime, _privateCreation) {

        curator = _curator;
        daoCreator = _daoCreator;
        proposalDeposit = _proposalDeposit;
        rewardAccount = new ManagedAccount(address(this), false);
        DAOrewardAccount = new ManagedAccount(address(this), false);
        if (address(rewardAccount) == 0)
            throw;
        if (address(DAOrewardAccount) == 0)
            throw;
        lastTimeMinQuorumMet = now;
        minQuorumDivisor = 5; // sets the minimal quorum to 20%
        proposals.length = 1; // avoids a proposal with ID 0 because it is used

        allowedRecipients[address(this)] = true;
        allowedRecipients[curator] = true;
    }

    function () returns (bool success) {
        if (now < closingTime + creationGracePeriod && msg.sender != address(extraBalance))
            return createTokenProxy(msg.sender);
        else
            return receiveEther();
    }


    function receiveEther() returns (bool) {
        return true;
    }


    function newProposal(
        address _recipient,
        uint _amount,
        string _description,
        bytes _transactionData,
        uint _debatingPeriod,
        bool _newCurator
    ) onlyTokenholders returns (uint _proposalID) {

        // Sanity check
        if (_newCurator && (
            _amount != 0
            || _transactionData.length != 0
            || _recipient == curator
            || msg.value > 0
            || _debatingPeriod < minSplitDebatePeriod)) {
            throw;
        } else if (
            !_newCurator
            && (!isRecipientAllowed(_recipient) || (_debatingPeriod <  minProposalDebatePeriod))
        ) {
            throw;
        }

        if (_debatingPeriod > 8 weeks)
            throw;

        if (!isFueled
            || now < closingTime
            || (msg.value < proposalDeposit && !_newCurator)) {

            throw;
        }

        if (now + _debatingPeriod < now) // prevents overflow
            throw;

        // to prevent a 51% attacker to convert the ether into deposit
        if (msg.sender == address(this))
```

```
        throw;

    _proposalID = proposals.length++;
    Proposal p = proposals[_proposalID];
    p.recipient = _recipient;
    p.amount = _amount;
    p.description = _description;
    p.proposalHash = sha3(_recipient, _amount, _transactionData);
    p.votingDeadline = now + _debatingPeriod;
    p.open = true;
    //p.proposalPassed = False; // that's default
    p.newCurator = _newCurator;
    if (_newCurator)
        p.splitData.length++;
    p.creator = msg.sender;
    p.proposalDeposit = msg.value;

    sumOfProposalDeposits += msg.value;

    ProposalAdded(
        _proposalID,
        _recipient,
        _amount,
        _newCurator,
        _description
    );
}


function checkProposalCode(
    uint _proposalID,
    address _recipient,
    uint _amount,
    bytes _transactionData
) noEther constant returns (bool _codeChecksOut) {
    Proposal p = proposals[_proposalID];
    return p.proposalHash == sha3(_recipient, _amount, _transactionData);
}


function vote(
    uint _proposalID,
    bool _supportsProposal
) onlyTokenholders noEther returns (uint _voteID) {

    Proposal p = proposals[_proposalID];
    if (p.votedYes[msg.sender]
        || p.votedNo[msg.sender]
        || now >= p.votingDeadline) {

        throw;
    }

    if (_supportsProposal) {
        p.yea += balances[msg.sender];
        p.votedYes[msg.sender] = true;
    } else {
        p.nay += balances[msg.sender];
        p.votedNo[msg.sender] = true;
    }

    if (blocked[msg.sender] == 0) {
        blocked[msg.sender] = _proposalID;
    } else if (p.votingDeadline > proposals[blocked[msg.sender]].votingDeadline) {
        // this proposal's voting deadline is further into the future than
        // the proposal that blocks the sender so make it the blocker
        blocked[msg.sender] = _proposalID;
    }

    Voted(_proposalID, _supportsProposal, msg.sender);
}


function executeProposal(
    uint _proposalID,
    bytes _transactionData
) noEther returns (bool _success) {

    Proposal p = proposals[_proposalID];

    uint waitPeriod = p.newCurator
        ? splitExecutionPeriod
        : executeProposalPeriod;
    // If we are over deadline and waiting period, assert proposal is closed
    if (p.open && now > p.votingDeadline + waitPeriod) {
        closeProposal(_proposalID);
        return;
    }

    // Check if the proposal can be executed
    if (now < p.votingDeadline  // has the voting deadline arrived?
        // Have the votes been counted?
        || !p.open
        // Does the transaction code match the proposal?
        || p.proposalHash != sha3(p.recipient, p.amount, _transactionData)) {

        throw;
    }

    // If the curator removed the recipient from the whitelist, close the proposal
    // in order to free the deposit and allow unblocking of voters
    if (!isRecipientAllowed(p.recipient)) {
        closeProposal(_proposalID);
        p.creator.send(p.proposalDeposit);
        return;
    }

    bool proposalCheck = true;

    if (p.amount > actualBalance())
        proposalCheck = false;

    uint quorum = p.yea + p.nay;

    // require 53% for calling newContract()
    if (_transactionData.length >= 4 && _transactionData[0] == 0x68
        && _transactionData[1] == 0x37 && _transactionData[2] == 0xff
        && _transactionData[3] == 0x1e
        && quorum < minQuorum(actualBalance() + rewardToken[address(this)])) {

            proposalCheck = false;
    }

    if (quorum >= minQuorum(p.amount)) {
        if (!p.creator.send(p.proposalDeposit))
            throw;
        lastTimeMinQuorumMet = now;
        // set the minQuorum to 20% again, in the case it has been reached
        if (quorum > totalSupply / 5)
            minQuorumDivisor = 5;
    }

    // Execute result
    if (quorum >= minQuorum(p.amount) && p.yea > p.nay && proposalCheck) {
        if (!p.recipient.call.value(p.amount)(_transactionData))
            throw;

        p.proposalPassed = true;
        _success = true;

        // only create reward tokens when ether is not sent to the DAO itself and
        // related addresses. Proxy addresses should be forbidden by the curator.
        if (p.recipient != address(this) && p.recipient != address(rewardAccount)
            && p.recipient != address(DAOrewardAccount)
            && p.recipient != address(extraBalance)
            && p.recipient != address(curator)) {

            rewardToken[address(this)] += p.amount;
            totalRewardToken += p.amount;
        }
    }

    closeProposal(_proposalID);

    // Initiate event
    ProposalTallied(_proposalID, _success, quorum);
}


function closeProposal(uint _proposalID) internal {
    Proposal p = proposals[_proposalID];
    if (p.open)
        sumOfProposalDeposits -= p.proposalDeposit;
    p.open = false;
}

function splitDAO(
    uint _proposalID,
    address _newCurator
) noEther onlyTokenholders returns (bool _success) {

    Proposal p = proposals[_proposalID];

    // Sanity check

    if (now < p.votingDeadline  // has the voting deadline arrived?
        //The request for a split expires XX days after the voting deadline
        || now > p.votingDeadline + splitExecutionPeriod
        // Does the new Curator address match?
        || p.recipient != _newCurator
        // Is it a new curator proposal?
        || !p.newCurator
        // Have you voted for this split?
        || !p.votedYes[msg.sender]
        // Did you already vote on another proposal?
        || (blocked[msg.sender] != _proposalID && blocked[msg.sender] != 0) )  {

        throw;
    }

    // If the new DAO doesn't exist yet, create the new DAO and store the
    // current split data
    if (address(p.splitData[0].newDAO) == 0) {
        p.splitData[0].newDAO = createNewDAO(_newCurator);
        // Call depth limit reached, etc.
        if (address(p.splitData[0].newDAO) == 0)
            throw;
        // should never happen
        if (this.balance < sumOfProposalDeposits)
            throw;
        p.splitData[0].splitBalance = actualBalance();
        p.splitData[0].rewardToken = rewardToken[address(this)];
        p.splitData[0].totalSupply = totalSupply;
        p.proposalPassed = true;
    }

    // Move ether and assign new Tokens
    uint fundsToBeMoved =
        (balances[msg.sender] * p.splitData[0].splitBalance) /
        p.splitData[0].totalSupply;
    if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == false)
        throw;


    // Assign reward rights to new DAO
    uint rewardTokenToBeMoved =
        (balances[msg.sender] * p.splitData[0].rewardToken) /
        p.splitData[0].totalSupply;

    uint paidOutToBeMoved = DAOpaidOut[address(this)] * rewardTokenToBeMoved /
        rewardToken[address(this)];

    rewardToken[address(p.splitData[0].newDAO)] += rewardTokenToBeMoved;
    if (rewardToken[address(this)] < rewardTokenToBeMoved)
        throw;
    rewardToken[address(this)] -= rewardTokenToBeMoved;

    DAOpaidOut[address(p.splitData[0].newDAO)] += paidOutToBeMoved;
    if (DAOpaidOut[address(this)] < paidOutToBeMoved)
        throw;
    DAOpaidOut[address(this)] -= paidOutToBeMoved;

    // Burn DAO Tokens
    Transfer(msg.sender, 0, balances[msg.sender]);
    withdrawRewardFor(msg.sender); // be nice, and get his rewards
    totalSupply -= balances[msg.sender];
    balances[msg.sender] = 0;
    paidOut[msg.sender] = 0;
    return true;
}

function newContract(address _newContract){
    if (msg.sender != address(this) || !allowedRecipients[_newContract]) return;
    // move all ether
    if (!_newContract.call.value(address(this).balance)()) {
        throw;
    }

    //move all reward tokens
    rewardToken[_newContract] += rewardToken[address(this)];
    rewardToken[address(this)] = 0;
    DAOpaidOut[_newContract] += DAOpaidOut[address(this)];
    DAOpaidOut[address(this)] = 0;
}


function retrieveDAOReward(bool _toMembers) external noEther returns (bool _success) {
    DAO dao = DAO(msg.sender);

    if ((rewardToken[msg.sender] * DAOrewardAccount.accumulatedInput()) /
        totalRewardToken < DAOpaidOut[msg.sender])
        throw;

    uint reward =
        (rewardToken[msg.sender] * DAOrewardAccount.accumulatedInput()) /
        totalRewardToken - DAOpaidOut[msg.sender];
    if(_toMembers) {
        if (!DAOrewardAccount.payOut(dao.rewardAccount(), reward))
            throw;
        }
    else {
        if (!DAOrewardAccount.payOut(dao, reward))
            throw;
    }
    DAOpaidOut[msg.sender] += reward;
    return true;
}

function getMyReward() noEther returns (bool _success) {
    return withdrawRewardFor(msg.sender);
}


function withdrawRewardFor(address _account) noEther internal returns (bool _success) {
    if ((balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply < paidOut[_account])
        throw;

    uint reward =
        (balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply - paidOut[_account];
    if (!rewardAccount.payOut(_account, reward))
        throw;
    paidOut[_account] += reward;
    return true;
}


function transfer(address _to, uint256 _value) returns (bool success) {
    if (isFueled
        && now > closingTime
        && !isBlocked(msg.sender)
        && transferPaidOut(msg.sender, _to, _value)
        && super.transfer(_to, _value)) {

        return true;
    } else {
        throw;
    }
}


function transferWithoutReward(address _to, uint256 _value) returns (bool success) {
    if (!getMyReward())
        throw;
    return transfer(_to, _value);
```

```
}


function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {
    if (isFueled
        && now > closingTime
        && !isBlocked(_from)
        && transferPaidOut(_from, _to, _value)
        && super.transferFrom(_from, _to, _value)) {

        return true;
    } else {
        throw;
    }
}


function transferFromWithoutReward(
    address _from,
    address _to,
    uint256 _value
) returns (bool success) {

    if (!withdrawRewardFor(_from))
        throw;
    return transferFrom(_from, _to, _value);
}


function transferPaidOut(
    address _from,
    address _to,
    uint256 _value
) internal returns (bool success) {

    uint transferPaidOut = paidOut[_from] * _value / balanceOf(_from);
    if (transferPaidOut > paidOut[_from])
        throw;
    paidOut[_from] -= transferPaidOut;
    paidOut[_to] += transferPaidOut;
    return true;
}


function changeProposalDeposit(uint _proposalDeposit) noEther external {
    if (msg.sender != address(this) || _proposalDeposit > (actualBalance() + rewardToken[address(this)])
        / maxDepositDivisor) {

        throw;
    }
    proposalDeposit = _proposalDeposit;
}


function changeAllowedRecipients(address _recipient, bool _allowed) noEther external returns (bool _success) {
    if (msg.sender != curator)
        throw;
    allowedRecipients[_recipient] = _allowed;
    AllowedRecipientChanged(_recipient, _allowed);
    return true;
}
```

```
function isRecipientAllowed(address _recipient) internal returns (bool _isAllowed) {
    if (allowedRecipients[_recipient]
        || (_recipient == address(extraBalance)
            // only allowed when at least the amount held in the
            // extraBalance account has been spent from the DAO
            && totalRewardToken > extraBalance.accumulatedInput()))
        return true;
    else
        return false;
}

function actualBalance() constant returns (uint _actualBalance) {
    return this.balance - sumOfProposalDeposits;
}


function minQuorum(uint _value) internal constant returns (uint _minQuorum) {
    // minimum of 20% and maximum of 53.33%
    return totalSupply / minQuorumDivisor +
        (_value * totalSupply) / (3 * (actualBalance() + rewardToken[address(this)]));
}


function halveMinQuorum() returns (bool _success) {
    // this can only be called after ‘quorumHalvingPeriod‘ has passed or at anytime
    // by the curator with a delay of at least ‘minProposalDebatePeriod‘ between the calls
    if ((lastTimeMinQuorumMet < (now - quorumHalvingPeriod) || msg.sender == curator)
        && lastTimeMinQuorumMet < (now - minProposalDebatePeriod)) {
        lastTimeMinQuorumMet = now;
        minQuorumDivisor *= 2;
        return true;
    } else {
        return false;
    }
}

function createNewDAO(address _newCurator) internal returns (DAO _newDAO) {
    NewCurator(_newCurator);
    return daoCreator.createDAO(_newCurator, 0, 0, now + splitExecutionPeriod);
}

function numberOfProposals() constant returns (uint _numberOfProposals) {
    // Don’t count index 0. It’s used by isBlocked() and exists from start
    return proposals.length - 1;
}

function getNewDAOAddress(uint _proposalID) constant returns (address _newDAO) {
    return proposals[_proposalID].splitData[0].newDAO;
}

function isBlocked(address _account) internal returns (bool) {
    if (blocked[_account] == 0)
        return false;
    Proposal p = proposals[blocked[_account]];
    if (now > p.votingDeadline) {
        blocked[_account] = 0;
        return false;
    } else {
        return true;
    }
}

function unblockMe() returns (bool) {
        return isBlocked(msg.sender);
    }
}

contract DAO_Creator {
    function createDAO(
        address _curator,
        uint _proposalDeposit,
        uint _minTokensToCreate,
        uint _closingTime
    ) returns (DAO _newDAO) {

        return new DAO(
            _curator,
            DAO_Creator(this),
            _proposalDeposit,
            _minTokensToCreate,
            _closingTime,
            msg.sender
        );
    }
}
```

A.4. **Sample Offer.**

```
contract SampleOffer {

    uint totalCosts;
    uint oneTimeCosts;
    uint dailyCosts;


    address contractor;
    bytes32 hashOfTheTerms;
    uint minDailyCosts;
    uint paidOut;

    uint dateOfSignature;
    DAO client; // address of DAO

    bool public promiseValid;
    uint public rewardDivisor;
    uint public deploymentReward;

    modifier callingRestriction {
        if (promiseValid) {
            if (msg.sender != address(client))
                throw;
        } else if (msg.sender != contractor) {
                throw;
        }
        -
    }

    modifier onlyClient {
        if (msg.sender != address(client))
            throw;
        -
    }

    function SampleOffer(
        address _contractor,
        bytes32 _hashOfTheTerms,
        uint _totalCosts,
        uint _oneTimeCosts,
        uint _minDailyCosts,
        uint _rewardDivisor,
        uint _deploymentReward
    ) {
        contractor = _contractor;
        hashOfTheTerms = _hashOfTheTerms;
        totalCosts = _totalCosts;
        oneTimeCosts = _oneTimeCosts;
        minDailyCosts = _minDailyCosts;
        dailyCosts = _minDailyCosts;
        rewardDivisor = _rewardDivisor;
        deploymentReward = _deploymentReward;
    }

    function sign() {
        if (msg.value < totalCosts || dateOfSignature != 0)
            throw;
        if (!contractor.send(oneTimeCosts))
            throw;
        client = DAO(msg.sender);
        dateOfSignature = now;
        promiseValid = true;
    }

    function setDailyCosts(uint _dailyCosts) onlyClient {
        dailyCosts = _dailyCosts;
        if (dailyCosts < minDailyCosts)
            promiseValid = false;
    }

    function returnRemainingMoney() onlyClient {
        if (client.receiveEther.value(this.balance)())
            promiseValid = false;
    }

    function getDailyPayment() {
        if (msg.sender != contractor)
            throw;
        uint amount = (now - dateOfSignature) / (1 days) * dailyCosts - paidOut;
        if (contractor.send(amount))
            paidOut += amount;
    }

    function setRewardDivisor(uint _rewardDivisor) callingRestriction {
        if (_rewardDivisor < 50)
            throw; // 2% is the default max reward
        rewardDivisor = _rewardDivisor;
    }

    function setDeploymentFee(uint _deploymentReward) callingRestriction {
        if (deploymentReward > 10 ether)
            throw; // TODO, set a max defined by Curator, or ideally oracle (set in euro)
        deploymentReward = _deploymentReward;
    }

    function updateClientAddress(DAO _newClient) callingRestriction {
        client = _newClient;
    }

    // interface for Ethereum Computer
    function payOneTimeReward() returns(bool) {
        if (msg.value < deploymentReward)
```

```
            throw;
        if (promiseValid) {
            if (client.DAOrewardAccount().call.value(msg.value)()) {
                return true;
            } else {
                throw;
            }
        } else {
            if (contractor.send(msg.value)) {
                return true;
            } else {
                throw;
            }
        }
    }

    // pay reward
    function payReward() returns(bool) {
        if (promiseValid) {
            if (client.DAOrewardAccount().call.value(msg.value)()) {
                return true;
            } else {
                throw;
            }
        } else {
            if (contractor.send(msg.value)) {
                return true;
            } else {
                throw;
            }
        }
    }
}
```

A.5. Managed Account.

```
contract ManagedAccountInterface {
    // The only address with permission to withdraw from this account
    address public owner;
    // If true, only the owner of the account can receive ether from it
    bool public payOwnerOnly;
    // The sum of ether (in wei) which has been sent to this contract
    uint public accumulatedInput;

    /// @notice Sends ‘_amount‘ of wei to _recipient
    /// @param _amount The amount of wei to send to ‘_recipient‘
    /// @param _recipient The address to receive ‘_amount‘ of wei
    /// @return True if the send completed
    function payOut(address _recipient, uint _amount) returns (bool);

    event PayOut(address indexed _recipient, uint _amount);
}


contract ManagedAccount is ManagedAccountInterface{

    // The constructor sets the owner of the account
    function ManagedAccount(address _owner, bool _payOwnerOnly) {
        owner = _owner;
        payOwnerOnly = _payOwnerOnly;
    }

    // When the contract receives a transaction without data this is called.
    // It counts the amount of ether it receives and stores it in
    // accumulatedInput.
    function() {
        accumulatedInput += msg.value;
    }

    function payOut(address _recipient, uint _amount) returns (bool) {
        if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient != owner))
            throw;
        if (_recipient.call.value(_amount)()) {
            PayOut(_recipient, _amount);
            return true;
        } else {
            return false;
        }
    }
}
```