Enabling a decentralized organization through smart contracts and tokens on the Ethereum blockchain

Hung Huy Tran

Degree Thesis
Information Technology
2018

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	6604
Författare:	Hung Huy Tran
Arbetets namn:	Att möjliggöra en decentraliserad organisation genom smarta kontrakt och tokens på blockkedjan Ethereum
Handledare (Arcada):	M.Sc. Magnus Westerlund
Uppdragsgivare:	KSF Media / Media organisation

Sammandrag:
I början av 2014 föreslog och beskrev Vitalik Buterin i sin vitbok en blockkedja som kan utföra arbiträra komplexa beräkningar. Detta ledde till skapandet av Ethereum, som kan implementera helt tillförlitligt smarta kontrakt. Ethereum är ett distribuerat register upprätthållet av noder i ett nätverk. I nätverket kör noderna Ethereum Virtuella Maskin (EVM), som är en exekveringsmiljö för smarta kontrakt. Syftet med detta examensarbete är att integrera ett system som möjliggör en decentraliserad organisation (DAO), som är självberoende och drivs av autonoma tjänster. En så kallad token skapas också som en valuta inom systemet, för att stöda värdeöverföring inom organisationen. Dessutom skapas också ett auktoritetssystem, som bestämmer användarnas befogenhet i organisationen. Målet uppnås genom att man skapar ett koncepttest med smarta kontrakt på blockkedjan Ethereum. Den består av två huvudkomponenter: ett token- och ett organisationskontrakt. Det förstnämnda fungerar som organisationens tillgång, medan det senare erbjuder tjänster till användaren. Funktioner (tjänster) som implementeras i den praktiska delen av arbetet är begränsade till krav från mediaföretaget KSF. Det innebär att inga ytterligare funktioner skapas om det inte är nödvändigt. Dessutom testas också varje funktion manuellt i stället för automatiskt. Slutligen fokuserar arbetet endast på back-end-delen av utvecklingen. Skapandet av de smarta kontrakten sker med hjälp av programmeringsspråket Solidity, Mist och andra verktyg. Mist är en webbläsare som används för att distribuera, testa och exekvera smarta kontrakt. Dessutom används också dokumentationer, böcker, artiklar och forum för att lära om Ethereum och smarta kontrakt. Arbetet är indelat i sex kapitel. Introduktionskapitlet beskriver syfte, avgränsning och metod. I andra kapitlet förklaras konceptet och kraven från mediaföretaget. I tredje kapitlet förklaras centrala begrepp gällande Ethereum och smarta kontrakt. Fjärde kapitlet berör utvecklingsverktyg som användes under den praktiska delen av arbetet. Utförandet av den praktiska delen beskrivs i femte kapitlet. Där berörs skapande och testning av de smarta kontrakten. I sista kapitlet dras en slutsats för hela arbetet. Resultatet av arbetet är en redogörelse för smarta kontrakts fördelar och nackdelar, samt kraven för att implementera smarta kontrakt.

Nyckelord:	Ethereum, blockkedja, smarta kontrakt, token, DAO
Sidantal:	61
Språk:	Engelska
Datum för godkännande:	

DEGREE THESIS	
Arcada	
Degree Programme:	Information Technology
Identification number:	6604
Author:	Hung Huy Tran
Title:	Enabling a decentralized organization through smart contracts and token on the Ethereum blockchain
Supervisor (Arcada):	M.Sc. Magnus Westerlund
Commissioned by:	KSF Media / Media organization

Abstract:

The thesis focuses on the Ethereum blockchain, which is a distributed register maintained by nodes in a network. The thesis examines Ethereum’s smart contract, which allows automatic execution of arbitrary calculations. In the practical part, smart contracts are used to integrate a system that enables a decentralized organization (DAO), which is self-reliant and driven by autonomous services. A so-called token is also created as a currency within the system. Features implemented in the practical work are limited to the requirements of the media company. The creation of the smart contracts takes place through the programming language Solidity and other development tools required to complete the thesis. Materials are taken from literature and media such as documentations, forums, video clips, and books about Ethereum and smart contracts.

This thesis is divided into six chapters. The introduction chapter describes the purpose, delimitation and method. The second chapter explains the concept and the requirements on the smart contracts. The third chapter explains key concepts regarding Ethereum and smart contracts. The fourth chapter concerns development tools used during the practical part of the thesis. The practical part is described in the fifth chapter. It involves the creation and testing of smart contracts. The final chapter draws a conclusion for the entire thesis.

The result of the thesis is a statement of the advantages and disadvantages of smart contracts, as well as the requirements for implementing smart contracts.

Keywords:	Ethereum, blockchain, smart contract, token, DAO
Number of pages:	61
Language:	English
Date of acceptance:	

CONTENTS

Figures

Tables

ABBREVIATIONS

DAO	Decentralized organization
Dapps	Decentralized applications
EOA	Externally owned account
EVM	Ethereum virtual environment
GUI	Graphic user interface
LLL	Lisp Like Language
IDE	Integrated development environment
P2P	Peer-to-peer
POW	Proof of work
POS	Proof of stake
Testnet	Testing network

ACKNOWLEDGEMENT

In this section, I would like to thank Petri Honkanen, who is a blockchain research specialist, and M. Sc. Magnus Westerlund, who works at Arcada, for giving me this assignment, as well as, supervising and guiding me during the thesis process.

Lastly, I would also like to thank my family for supporting me during my studies at Arcada. Without them, I would have not considered applying to Arcada a few years ago.

1 INTRODUCTION

1.1 Background

Blockchain is a distributed computing network, which is run by networked processing nodes. The task of a node is to execute and record transactions that are collected into blocks. These blocks are appended to a public ledger in sequence. In addition, blocks are also immutable due to strong cryptography. As a result, the network remains secure. (Ethereum community 2016)

The initial blockchain implementation is first described in Satoshi Nakamoto's white paper about Bitcoin, which utilizes blockchain technology to transfer value (Nakamoto 2008). Since the paper, the blockchain development has increased exponentially. As a result, this lead to the creation of altcoins, which are upgraded versions of Bitcoin. (Ethereum community 2016)

In the early 2014, Vitalik Buterin proposed and described in his white paper (Buterin 2014) about a blockchain, which can perform any arbitrary complex computation. In his paper, he presented the technical designs and rationale for a blockchain protocol and smart contract architecture. As a result, this lead to the creation of Ethereum, which is a blockchain that can implement a fully trustless smart contract. (Ethereum community 2016)

1.2 Objective

The objective of this thesis is to create a token that enables the functioning of a decentralized organization (DAO) for a media organization that also provide the requirements for the work. The goal is accomplished by building proof of concept with smart contracts on the Ethereum blockchain. In addition, the work is about learning and understanding challenges in Ethereum and smart contract development.

In the thesis, the following questions will be answered:

- What are the advantages and disadvantages with smart contract?
- How are smart contracts tested?
- What are the requirements for building smart contracts?

1.3 Delimitation

Since this thesis focuses mainly on Ethereum and smart contract development, no other blockchains will be covered. In this thesis, DAO and the token services (functions) are limited to the requirements of the commissioning organization, which will be listed in chapter two. This means that no additional functions will be created if it is not necessary. Moreover, each function will be tested manually instead of automatic. While the latter has better advantage over former in terms of speed, functions are limited enough to be tested manually in this thesis. Lastly, this thesis focuses solely on the back-end development of the decentralized application (Dapp) and does not include a graphic user interface (GUI).

1.4 Method

In this thesis, documentations, books, articles and forums are used for learning about Ethereum and smart contract development. The smart contracts are built with the help of tutorials, documentation and external tools, as well as, frequent testing of the smart contracts with a browser. The smart contracts will be written in integrated development environments (IDE) that supports Solidity.

2 PLANNING PROCESS

A plan was constructed for the architecture of a system supporting a DAO. The scheme was based on the requirements by KSF and worked as a guideline for building the smart contract in this thesis.

2.1 Concept

The concept is to enable a decentralized media organization that is driven by smart contracts. Through smart contracts, the organization can automate a service that would otherwise require administration. Also, the idea is to have the transactions handled through the blockchain. As a result, this allows users to directly interact with the services in connection to the organization.

A token is also created. The purpose of a token is to serve as the currency within the organization. The token exists as a payment method for executing services, as well as, being a reward for contributing to the organization.

The designed system should have a hierarchy for determining the authority level. Without any authority, all the smart contracts would be accessible by anyone. This would cause vulnerabilities within the structure of the organization. For this reason, different types of membership are created. These are “Normal”, “CEO”, “Member”.

2.2 Requirements

This section presents a list of requirements by KSF.

The requirements are:

- Create a token for the organization.
- Create a proof of stake function for the token.
- Create membership system for the organization.
- Allow purchase and sell of token.
- Allow members to create content by paying token.
- Allow members to review content, which gives token reward.
- Allow members to view content.
- Allow CEO to change the organization settings.
- All transactions should be secure.

3 ETHEREUM

Ethereum is a Turing complete blockchain for decentralized applications. The blockchain includes a peer-to-peer network protocol that connects nodes into a network. In the network, nodes run the Ethereum Virtual Machine (EVM), which is a runtime environment for smart contracts. (Ethereum community 2016)

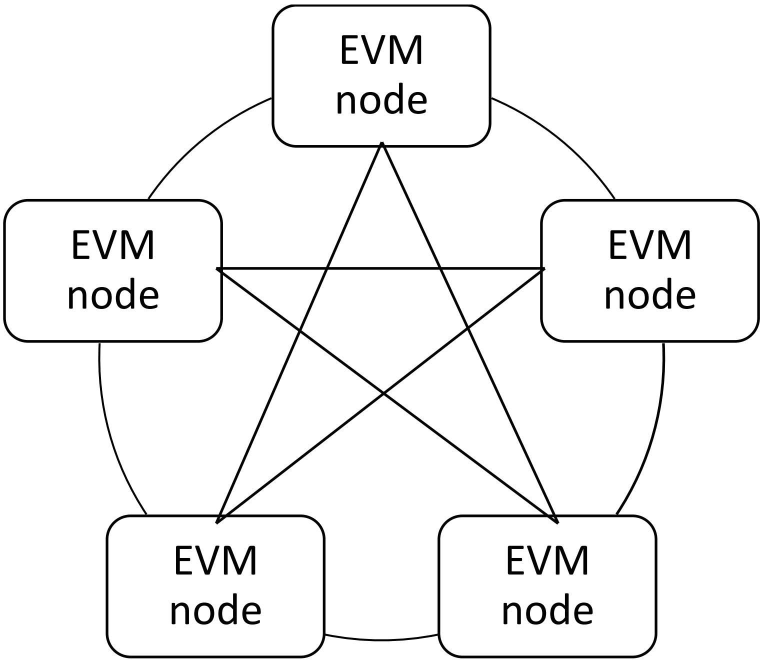

Figure 1. The architecture of Ethereum blockchain, where nodes runs the EVM.

While Ethereum is slower than a traditional computer in terms of computation, it has several advantages over the latter. Ethereum has the benefit of an extreme level of fault tolerance, zero downtime, and an immutable data store. Additionally, all the states and transaction history are public. (Ethereum community 2016)

3.1 Accounts

In Ethereum, accounts are necessary for executing transactions and communicating with the blockchain. Accounts are state objects that have a 20-byte address, which represent an external agent's identification that can either be externally owned accounts (EOAs) or contract accounts. These two account types are similar but are controlled in different ways; while EOAs are controlled by private keys, contract accounts are commanded by their internal code, which can only be executed by EOAs. Furthermore, all accounts also have public keys that are used to sign transactions. (Ethereum community 2016 & Buterin 2014 p.13)

Regardless of account type, every account has a balance, which can hold Ether (explained in section 3.2). In addition, contract accounts also have contract storage. These states are being updated with every new block on the blockchain. (Ethereum community 2016)

3.2 Ether

Ethereum operates with a currency known as Ether. It is used for making transactions and pay for EVM computation in gas, which is indirectly obtained through Ether. Ether can be held by an EOA or contract account. It can be sent to another account through the account's public key (address). However, the transactions require gas. Similarly, tokens, which are sub-currencies of Ether, also require gas to be transferred from one account to another.

Ether can either be obtained by mining, purchasing from other users, or trading for other cryptocurrencies. Further, ether can be split into smaller denominations as displayed on table 1. (Dannen 2017 p.38 & Ethereum community 2016)

Table 1. Ether denomination in Wei.

Unit	Wei
Wei	1
Kwei	1,000
Mwei	1,000,000
Gwei	1,000,000,000
Microether	1,000,000,000,000
Milliether	1,000,000,000,000,000
Ether	1,000,000,000,000,000,000

3.2.1 Gas

Gas are fees for transactions and the computation for consumed blockchain resources. It has an important part in increasing the stability and long-term demand for ether. (Ethereum community 2016)

While ether's value fluctuates, gas is supposed to remain stable. In Ethereum, gas is dynamically adjusted according to the price of ether due to the currency's price volatility. This is calculated based on the volume and complexity of the request, multiplied with the current gas price. (Ethereum community 2016)

The gas value is associated with numerous terms displayed in table 2.

Table 2. Explanation of gas associated definitions.

Gas cost	A static value that should always remain stable. It is used for calculating the computation cost.
Gas price	A floating value that is adjusted with the price of ether. The price depends on the user's willingness to spend and the node's acceptance on the blockchain.
Gas limit	The maximum amount of gas that can be used per block on the blockchain.
Gas fees	The fee paid to the miners that are required for making transactions or executing smart contracts.

3.3 Mining

Mining is a definition for increasing the total volume of ether on the blockchain. A term for nodes securing the network by mining is called miners. The miners job is to create, verify, publish, and propagate blocks on the blockchain. To produce new blocks, the miners have to compete with each other by solving a difficult algorithm in order to receive ether as reward for their contribution. In addition, the reward will only be distributed to the block finder if the block is valid and contains proof of work (POW) of a given difficulty. (Ethereum community 2016)

A block is created on 15 seconds average on the Ethereum network. This equals the punctuation between the nodes on the blockchain, which guarantees that no malicious attacks can occur, such as rewriting history and double spending. (Ethereum community 2016)

3.3.1 Proof of work

Ethereum uses a POW algorithm called Ethash, which involves finding a dataset, a header, and a nonce. To find these three, the headers of each block from the blockchain is hashed together to create a seed, which is used to generate a pseudorandom cache. This cache produces a dataset using a non-cryptographic hash function. The dataset, a header and a nonce is repeatedly hashed until the results satisfies the difficulty target. (Tikhomirov 2017)

3.3.2 Proof of stake

Proof of stake (POS) is an alternative method for reaching consensus on a blockchain. Consensus is reached through validators that are given the task to suggest transactions, in forms of block, to the blockchain. In order to suggest, validators have to solve a POS algorithm. The idea with the algorithm is to control the overflow of suggestions on the blockchain. In addition, the validators must include the solution together with the suggested block for validation check. If the block is regarded as valid, then it is appended to the network. (Li et al. 2017 p.298-299)

Solving the POS algorithm requires stakes, which are virtual resources. The amount of stakes decides the speed of discovering a solution and generating a new block on the blockchain. (Li et al. 2017 p.300).

3.4 Smart contract

Smart contracts and traditional contracts handles the agreement procedures in different ways. The latter relies on trusted institutions to follow the procedures, while the former's process is handled by the Ethereum blockchain. In comparison, the former has an advantage over the latter in terms of handling the procedures. The former allows the possibility of entering an agreement that would otherwise be considered too risky with traditional contracts. (Ethereum community 2016)

Smart contracts are a set of functions and states that allows Turing complete computation. In Ethereum, a deployed smart contract is located at a specific address where they exist in an Ethereum-specific binary format. All contracts executed on the EVM, which allows them to communicate with each other. (Ethereum community 2016)

Smart contracts are usually developed with a high-level programming language like Solidity, which is explained in section 4.1. Other programming languages that can also be used for developing smart contracts are Serpent and LLL. (Ethereum community 2016)

3.4.1 Tokens

Tokens are digital assets whose value is determined by the market demand and supply in addition to the value proposition of the asset. In Ethereum, tokens are created with smart contracts. A token contract can be created with the standard ERC20Token contract, which contains variables and methods that enable the contract to function as an asset. The ERC20Token contract has for example, a balance sheet that records the ownership of the tokens, a transfer function for sending tokens, and a total supply variable, which represents the total amount of tokens. The interface of a ERC20Token contract can be seen in figure 6 in section 5.1. (Dannen 2017 p.97-104, Vogelstellar & Buterin 2015)

In Ethereum, tokens are not necessary because they are sub-currencies of Ether. Hence, creation of token depends entirely on the developer’s choice. (Dannen 2017 p.98)

3.4.2 DAO

Decentralized autonomous organization (DAO) is a collection of contracts (or a type of contract) on the blockchain. Its purpose is to be driven automatically by smart contracts, which is coded to automate services of organizations. Services offered by DAO can be for example, fundraising, operations, spending, governance, or expansion. (Ethereum community 2016)

3.4.3 Dapps

Decentralized applications (Dapps) are applications that automates business logic through smart contract. Dapps are usually built with a mixture of HTML, CSS and JavaScript for the front-end, and with Solidity and Ethereum as the back-end (see figure 2). As a result, this allows web applications to be partly decentralized. Developing Dapps requires some form of centralization for now but may further progress into full decentralization in the future. (Ethereum community 2016)

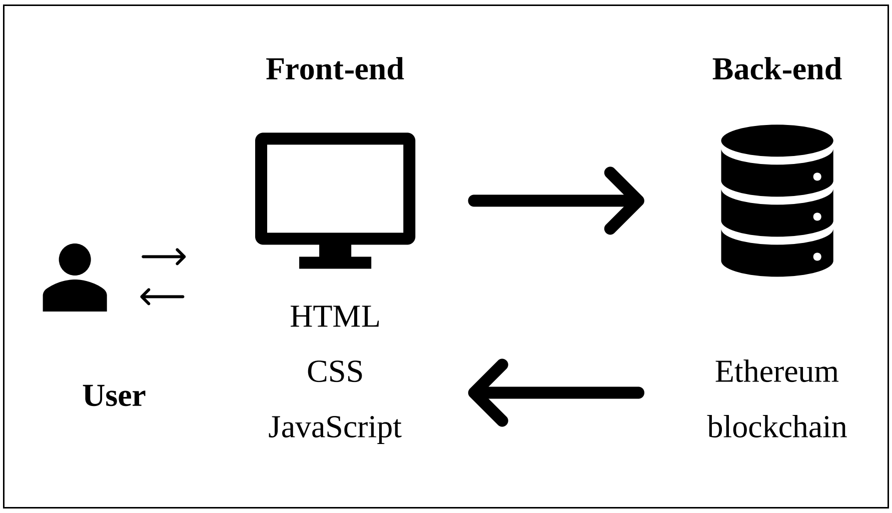

Figure 2. An illustration of the components of a decentralized application.

3.4.4 Vulnerabilities

Human error is a major cause for bugs occurring in smart contracts. The lack of documentation on the internet makes it difficult for developers to find reliable sources on how to create a secure smart contract. (Atzei, Bartoletti & Cimoli 2017 p.164-165 & Ethereum community 2016)

In Ethereum, smart contracts contain various vulnerabilities. For this reason, the awareness of vulnerabilities should be considered when developing smart contracts. Consequently, this reduces the possibility of malicious exploitation of the smart contract. In table 3, the vulnerabilities in smart contract are presented. (Atzei, Bartoletti & Cimoli 2017 p.169)

Table 3. List of smart contract vulnerabilities. (Atzei, Bartoletti & Cimoli 2017 p.169 - 175)

Vulnerabilities	Explanation
Call of the unknown	If a function, that does not exist on a contract, is called then the fallback function of that contract is called. The fallback function is a special function with no names or arguments, which can be arbitrarily programmed.
Gasless send	Out-of-gas exception occur when there is no signature or enough gas to execute.
Exception disorders	Exceptions is not handled consistently, instead they depend on the communication between contracts.
Type casts	Detecting type errors is limited which could cause undetected bugs.
Re-entrancy	A non-recursive function is invoked multiple times before its termination by using the fallback function of smart contracts.

Keeping secrets	Private fields in contracts does not ensure secrecy because the field values are visible in mined transactions.
Immutable bugs	Published contracts are immutable, therefore bugs cannot be directly patched.
Ether lost in transfer	Sending ether to an orphan address, which is not associated with any user or contract, results in funds being lost forever.
Stack size limit	Calling a contract from another contract adds one instance to the associated transaction. If the stack limit reaches 1024 instances, then an exception is thrown.
Unpredictable state	A transaction is not guaranteed to run in the same state the contract was at the time of execution, which could cause states to be unpredictable.
Generating randomness	Generating randomness can accomplished by using the seeds (hash or timestamp) from any block on the blockchain.
Time constraints	Restrict actions based on timestamp.

4 DEVELOPMENT TOOLS

4.1 Solidity

Solidity, a contract-oriented and high-level language, is used for writing the smart contracts in this thesis. The language was influenced by C++, Python, and JavaScript. It is a statically typed programming language that supports features, such as inheritance, libraries, and complex user-defined types. With Solidity, the developer can create smart contracts that allows e.g., voting, crowdfunding, blind auctions, multi-signature wallets and more. (Ethereum 2017)

4.2 Remix

Remix, an integrated development environment (IDE), is used for developing smart contracts in Solidity. Developing in Remix allows developers to find bugs easier. In the practical part of this thesis, Remix is used together with Mist (explained in section 4.4) to debug and test the smart contracts. (Yann300 2017)

4.3 IntelliJ IDEA Community Edition

IntelliJ IDEA Community Edition, which is developed by JetBrains, is a premier IDE that is free for download. It includes refactoring, code inspections, navigation, version control system integration, and more. It also supports various programming languages, e.g., Java, Groovy, Scala, and Clojure. In addition, alternative plugins can be downloaded and installed.

Although, the IDE is not required for completing the practical part of this thesis, it offers features, such as refactoring, project management, and syntax highlights that aids the development process significantly. In this thesis, the smart contracts are written in the IDE.

4.3.1 IntelliJ IDEA Community Edition installation

IntelliJ IDEA Community Edition is installed by executing a setup file, provided by JetBrains (https://www.jetbrains.com/idea). IntelliJ comes with limited features; however, additional features can be added through third-party plugins. IntelliJ-Solidity is a plugin that enables support for Solidity language (serce & ll345374 2018). The plugin is installed by following these steps:

1. Go to IntelliJ IDEA > Preferences > Plugins > Browse repositories.
2. Use the search field to type in “IntelliJ-Solidity”, which displays the plugin.
3. Press “Install”, which will install the plugin.
4. Lastly, IntelliJ needs to be restarted in order to active the plugin.

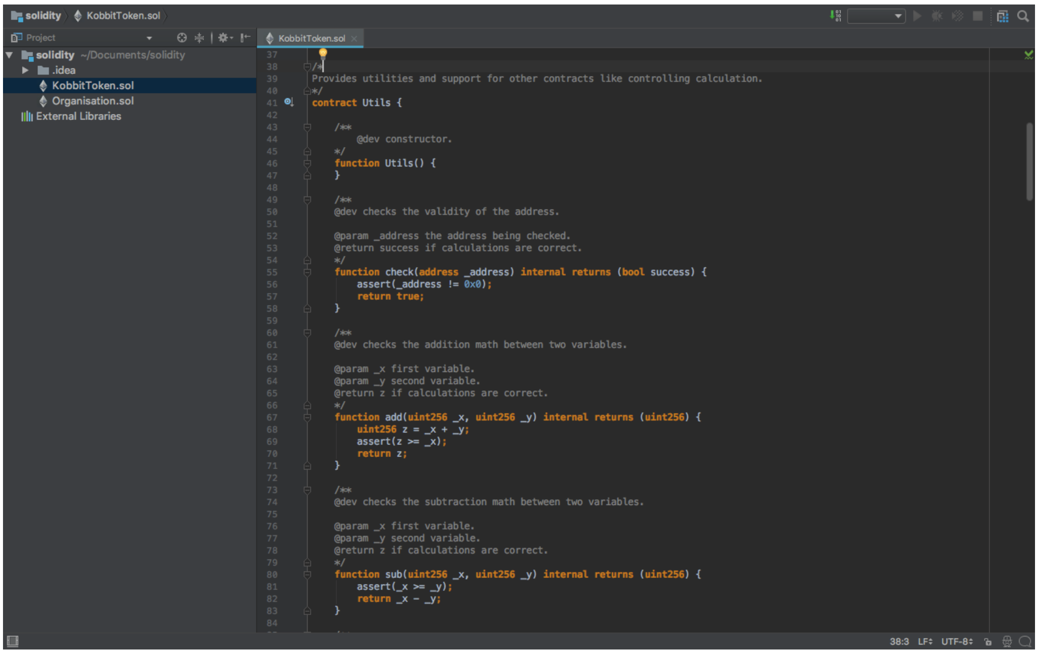

Figure 3. Interface of IntelliJ IDEA Community Edition.

4.4 Mist browser

Mist browser is a user-friendly wallet that connects to the Ethereum network. It can run a few functions of a full node e.g., execute ether transactions and smart contracts. In addition, the browser can also hold accounts. In the practical part, the browser is used for deploying, testing, and accessing distributed smart contracts. (Dannen 2017 p.21-22)

4.4.1 Mist browser installation

Mist browser is installed by executing a setup file, which can be downloaded from Ethereum official website (https://www.ethereum.org). The installation is simple because it only requires the user to follow the instructions during the process. However, to utilize the functionalities offered by the Mist browser, the user has to select a blockchain to synchronize with. This is done by navigating to the browser's menu bar > Develop > Network and selecting a network, which will start the synchronizing process.

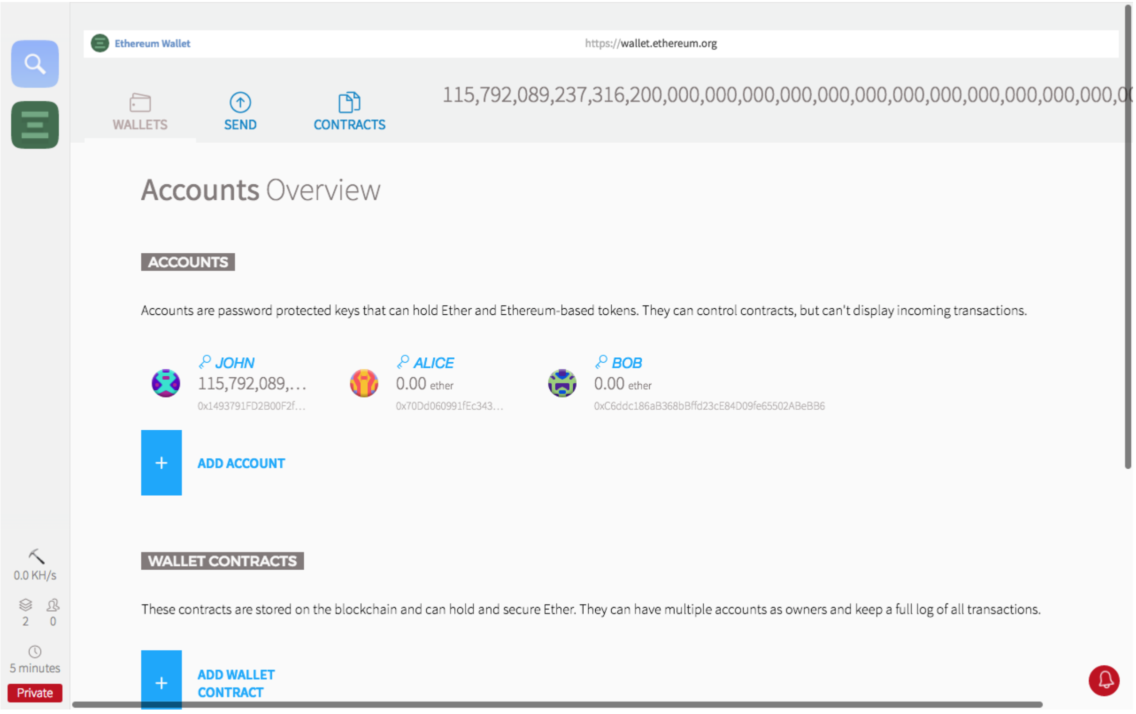

Figure 4. Home interface of Mist Browser.

4.5 Geth

The go-ethereum (Geth) is a command line interface tool for running a node on Ethereum. It is implemented by Go language to interact with the blockchain. Through Geth, the users have the ability to mine ether, transfer funds, create contracts, explore block history, and more. Geth is pre-installed with Mist. (Ethereum community 2016)

4.6 Testnet

Test network (Testnet) is an alternative Ethereum blockchain, which is used for tests. Developing in testnet allows the user to use play ether, which holds no value. (Dannen 2017 p.79)

5 IMPLEMENTATION PROCESS

The smart contracts consist of two key components: a token and organization contract. The former functions as the asset of an organization, while the latter offers services for the users.

5.1 Token contract

The token contract (figure 5) utilizes the functionalities provided by an ERC20Token (figure 6), which is a standard interface for token (Vogelstellar & Buterin 2015). The ERC20Token contains variables like name, symbols, decimals, and totalSupply, which determines the characteristics of the contract. Further, it also contains a transfer function for token, which can be seen on figure 7.

```
contract Kobbit is Owned, Token {
    uint public stakeRateInterest; // The interest rate of staking.
    uint public tokenPriceInWei; // The price per token.
    uint public stakeIntervalInMinutes; // The time interval between stakes.
    mapping (address => uint) public timeUntilNextStake; // The time until next available stake.

    event Staked(address stakingAddress, uint stakeAmount); // Broadcast if token is being staked.
    event TokenBought(address buyer, uint tokenAmount, uint etherAmount); // Broadcast if token is being bought.
    event TokenSold(address seller, uint tokenAmount); // Broadcast if token is being sold.
    event PaymentConfirmed(address from, uint tokenAmount); // Broadcast if payment is confirmed.
```

Figure 5. The token contract, which is named Kobbit.

```
contract Token is Utils{
    string public name; // The name of the token.
    string public symbol; // The symbol of the token.
    uint8 public decimals; // Amount of decimals in token number.
    uint256 public totalSupply; // The total supply of the token.

    mapping (address => uint256) public balanceOf; // Checks the balance of an address.
    mapping (address => mapping (address => uint256)) public allowance; // Checks the allowance that another sender have

    event Transfer(address indexed from, address indexed to, uint256 value); // Transfer broadcast message.
    event Approval(address indexed owner, address indexed _spender, uint256 _value); // Approval broadcast message.

    /*
    @dev the constructor that initiate initialSupply, tokenName, decimalUnits, tokenSymbol. Sets variable values.

    @param initialSupply the starting supply amount.
    @param tokenName the name of the token.
    @param decimalUnits decimals unit for token supply.
    @param tokenSymbol the symbol for the token.
    */
    function Token(
        uint256 initialSupply,
        string tokenName,
        uint8 decimalUnits,
        string tokenSymbol
        ) {
        balanceOf[msg.sender] = initialSupply; // Give the owner all of the token supply.
        totalSupply = initialSupply; // The initialSupply equals totalSupply.
        name = tokenName; // Give the token a name.
        symbol = tokenSymbol; // Symbol of the token.
        decimals = decimalUnits; // Given decimal units.
    }
```

Figure 6. The ERC20 Token contract.

```
    function transferFrom(address _from, address _to, uint256 _value)
        public
        returns (bool success)
    {
        check(_from); // Checks validity of address _from.
        check(_to); // Checks validity of address _to.
        allowance[_from][msg.sender] = sub(allowance[_from][msg.sender], _value); // Substract from allowance.
        balanceOf[_from] = sub(balanceOf[_from], _value); // Substract from address _from.
        balanceOf[_to] = add(balanceOf[_to], _value); // Add to address _to.
        Transfer(_from, _to, _value); // Broadcast transfer listenener.
        return true;
    }
}
```

Figure 7. ERC20 Token transfer function.

The token contract also inherits from two additional contracts that acts as supports. These are: “Owned” and “Utils” from the ERC20Token. The former determines the owner of contract, while the latter performs mathematical calculations for the transactions that are handled by the contract. In addition, the contract has functions for buying and selling tokens, a POS function, and a payment function, which can be seen from figure 8 to 11.

```
    function stake(address _to)
        public
        allowStake(_to)
        ownerOnly
        returns (bool success)
    {
        uint temp = mul(balanceOf[_to], stakeRateInterest); // Multiplies the balanceOf address with stakeRateInterest.
        uint stakeValue = div(temp, 100); // Divide the temporary value with 100, and we get stakeValue. Float numbers d
        balanceOf[_to] = add(balanceOf[_to], stakeValue); // Add the stakeValue to balanceOf address.
        totalSupply = add(totalSupply, stakeValue); // Add tokens to totalSupply.
        timeUntilNextStake[_to] = now + stakeIntervalInMinutes * 1 minutes; // Set the timer for next staking possibilit

        Staked(_to, stakeValue); // Broadcast a message.
        return true;
    }
```

Figure 8. POS function, which increase the amount of token based on the user’s stake.

```
    function buy(address from, uint amount)
        public
        ownerOnly
        returns (bool success)
    {
        balanceOf[from] = add(balanceOf[from], amount); // Add the amount to given address.
        balanceOf[msg.sender] = sub(balanceOf[msg.sender], amount); // Substract from the msg.sender.

        TokenBought(from, amount, amount / 1 ether); // Broadcast how many tokens were bought.
        return true;
    }
```

Figure 9. Buy function for purchasing token with ether.

```
function sell(uint256 amount, address from)
    public
    ownerOnly
    returns (bool success)
{
    balanceOf[msg.sender] = add(balanceOf[msg.sender], amount); // Add the amount to msg.sender.
    balanceOf[from] = sub(balanceOf[from], amount); // Substract from the address from.

    TokenSold(from, amount); // Broadcast the amount of token sold.
    return true;
}
```

Figure 10. Sell function for selling token for ether.

```
function pay(address _from, uint256 _amount)
    public
    ownerOnly
    returns (bool success)
{
    balanceOf[_from] = sub(balanceOf[_from], _amount); // Substract from the payer.
    balanceOf[msg.sender] = add(balanceOf[msg.sender], _amount); // Add to the msg.sender.

    PaymentConfirmed(_from, _amount); // Broadcast a message, payment confirmed.
    return true;
}
```

Figure 11. Payment function for transferring ether into an account.

5.2 Organization contract

The organization contract utilizes the token contract as an asset to run its ecosystem. It is also built with “Owned” and “Utils” contract that are mentioned earlier. In addition, the contract contains variables that determines the settings of the organization. The variables are set during the deployment process but can also be changed later. These variables are displayed on figure 12.

```
function Organisation(
    Token token,
    uint pricePerContent,
    uint minutesForReview,
    uint minimumReviewsForContents,
    uint marginOfReviewsForMajority,
    uint tokenRewardForReview)
```

Figure 12. The settings for organization contract.

The contract’s functions are limited to members only. This means that no outsider can execute the functions. However, functions are also restricted based on the authority level of its users. Hence, this allows the organization to delegate power according to membership types, which are Normal, CEO, and Member (displayed in figure 13).

In addition to the membership system, the functions also have requirements. Executing a function can for example, require a certain amount of Ether or token.

```
enum Account {Normal, CEO, Member}
```

Figure 13. Available memberships in the organization.

The contract's functions are necessary for demonstrating a proof of concept of a decentralized organization. Other than that, they also fulfill the requirements that is given in section 2.2. The testing of these functions, which can be seen from figure 14 to 21, are described in section 5.4.

```
function addMember( // Need a second function that uses this function when ownership is transferred to this contract
    address targetMember,
    string memberName,
    Account accountType)
    ownerOnly
{
    require(member[targetMember].memberAddress != targetMember); // Checks if the targetMember has an memberAddress.
    member[targetMember] = Member(targetMember, memberName, now, accountType); //Sets the targetMember to member map
}
```

Figure 14. Function for adding member to the organization, which is only executable by CEO.

```
function newContent(string contentTitle, string contentDescription)
    memberOnly
{
    tokenAddress.pay(msg.sender, contentPrice); // Uses the token address function to pay to this contract.
    uint contentID = contents.length++; // Set ID based on contents.length++.
    Content storage c = contents[contentID]; // Creates content struct that holds the content information.
    c.publisherAddress = msg.sender; // Set the publisher address.
    c.author = member[msg.sender].name; // Set the author of the content.
    c.title = contentTitle; // Set the content title.
    c.description = contentDescription; // Set the description of the content.
    c.reviewDeadline = now + reviewPeriodInMinutes * 1 minutes; // Deadline for reviewing content.
    c.numberOfReviews = 0; // Amount of reviews.
    c.executed = false; // Checks if the content is executed.
    c.contentPassed = false; // Checks if content passed the criterias.

    ContentAdded(contentID, c.author, c.title, c.description); // Event listener activated.
    numberOfContents = contentID+1; // Add the total number of content in organisation with 1.
}
```

Figure 15. Function for creating new content, which requires tokens.

```
    function reviewContent(
        uint contentNumber,
        bool supportsContent,
        string justificationText
    )
        memberOnly
    {
        Content storage c = contents[contentNumber]; // Checks a content based on number.
        require(now < c.reviewDeadline); // Checks if time is within deadline.
        require(c.publisherAddress != msg.sender); // Checks that the reviwer is not the publisher.
        require(c.reviewed[msg.sender] == false); // Checks if the reviewer reviewed.
        c.reviewed[msg.sender] = true; // Sets review to true
        c.numberOfReviews++; // Increase reviews by one.
        if (supportsContent) {
            c.currentResult++; // If the content is supported, increase by one.
        } else {
            if(c.currentResult > 0) {
            c.currentResult--; // If the reviewer does not support, decrease by one.
            }
        }

        tokenAddress.transfer(msg.sender, reviewReward); // Transfer tokens from this contract to the function executor.
        Reviewed(contentNumber, supportsContent, msg.sender, justificationText); // Event listener activated based on th
    }
```

Figure 16. Review function, which returns token as contribution reward.

```
    function executeContent(uint contentNumber)
        memberOnly
    {
        Content storage c = contents[contentNumber]; // Sets a content based on number.
        require(now > c.reviewDeadline // Checks if the deadline has passed.
          && !c.executed // Checks if the content has been executed or not.
          && c.numberOfReviews > minimumReviews); // Checks if content passed minimum review criteria.
        if (c.currentResult >= majorityMarginForContent) { // Checks if the content passed a certain margin.
            c.executed = true; // Execute.
            c.contentPassed = true; // Passed.
        } else {
            c.executed = true; // Execute.
            c.contentPassed = false; // Failed.
        }

        ContentTallied(contentNumber, c.currentResult, c.numberOfReviews, c.contentPassed); // Content information bein
    }
```

Figure 17. Function for executing a specific content that will either be approved or disapproved.

```
    function stake(address to)
        memberOnly
    {
        require(member[to].memberAddress == to); // Checks if the address of member equals the one to.
        tokenAddress.stake(to); // Stake based on the function on token address.
    }
```

Figure 18. Staking function, which increases the amount of tokens that the user has.

```
    function purchase() // Does not give the exact amount because float doesn't exist.
        memberOnly
        payable
    {
        uint amount = div(msg.value, tokenAddress.tokenPriceInWei()); // Calculates the amount of token that will be bou
        tokenAddress.buy(msg.sender, amount); // Buy function in token address.
    }
```

Figure 19. Buy tokens function, which exchanges Ether for tokens.

```
function sell(uint256 amount)
    memberOnly
{
    tokenAddress.sell(amount, msg.sender); // Sell the token to this contract.
    msg.sender.transfer(amount * 1 ether); // This contract send ether to the msg.sender.
}
```

Figure 20. Sell tokens function, which exchanges tokens for Ether.

```
function appointNewCEO(address to)
    ceoOnly
{
    require(member[to].memberAddress == to); // This function is broken. Checks if the address to is registered.
    check(to); // Checks if the address is valid.
    require(to != msg.sender); // Checks if the msg.sender is the address to.
    member[msg.sender].privilege = Account.Member; // Sets current CEO to Member.
    member[to].privilege = Account.CEO; // Address to gets appointed as CEO.
}
```

Figure 21. Appoint a new CEO function, which can only be executed by CEO.

5.3 Setup

This section focuses on the setup part, which is critical for testing the smart contracts that are mentioned above. The setups are done entirely in Mist browser.

5.3.1 Synchronizing with the blockchain

Solo network is a private blockchain that runs locally, which is used as the blockchain for testing smart contracts in this thesis. To be able to synchronize with the blockchain, the user has to navigate to browser’s menu bar > Develop > Network and then select solo network. After selecting and synchronizing with the blockchain, the user receives an infinite amount of play ether for testing.

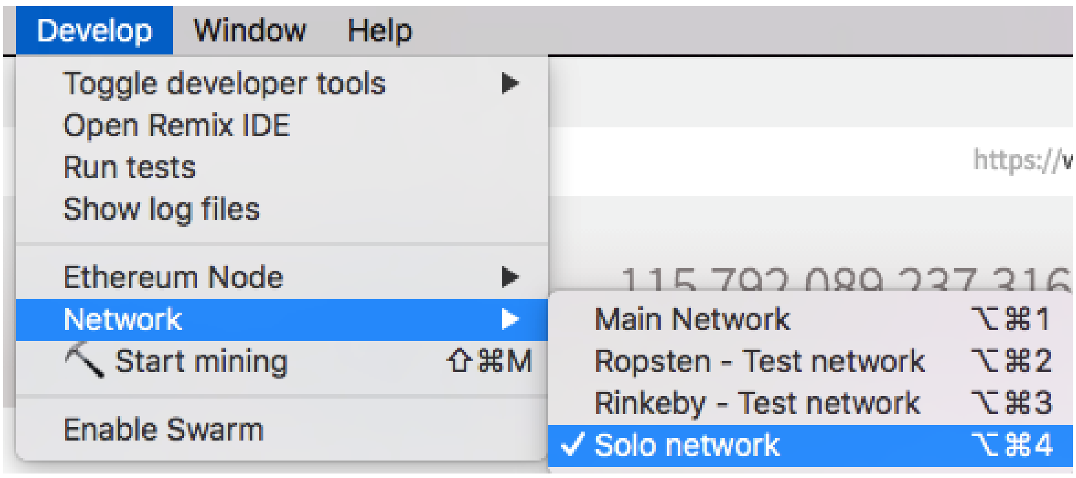

Figure 22. Selecting the network to synchronize with.

5.3.2 Creating additional accounts

Additional accounts are created to test the membership restrictions of the smart contracts. Additionally, each account also holds a certain amount of play ether, which is used for testing the functions. Accounts are created through “Add account”, which can be seen on figure 4 in section 4.4.1.

5.3.3 Deployment of the smart contracts

The contracts are deployed in Mist’s home interface > Contracts > Deploy new contract. Clicking on “Deploy new contract” will open a window that allows the user to insert a Solidity smart contract source code and select a contract to deploy (figure 23).

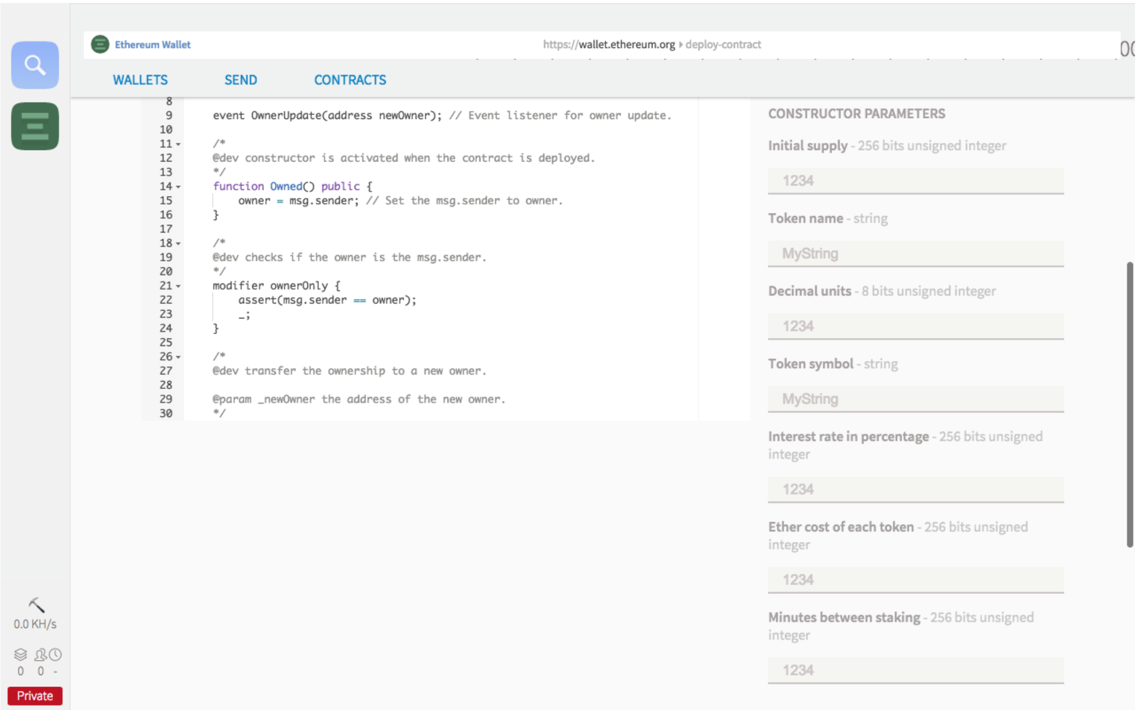

Figure 23. Deploy contract section, which displays Solidity contract source code and constructor parameter.

In this thesis, the token contract is first deployed in order to retrieve an address, which will be used to link the contract with the organization contract. Before the contract can be deployed, all the necessary constructor parameter has to be filled, which can also be seen at figure 23. After the deployment, the initial token supplies are distributed to the contract creator.

Deploying the organization contract works the same way. The only differences are the constructor parameter fields, which requires other information. These fields are:

- Token, which is the address of the token.
- Price per content, which decides the price of a content in the organization.
- Minutes for review, the amount of time for reviewing a content.
- Minimum reviews for contents, the minimum requirement for a content to be approved.
- Margin of reviews for majority, the amount of review required for majority.
- Token reward for review, the reward for reviewing a content.

5.3.4 Transferring token contract ownership and funds

After the deployment of the smart contracts, the user has to send all the token supplies, as well as, transfer the ownership of the token contract to the organization contract's address. The reason is to allow the latter to call the former's functions and store enough token balance to sell to users.

In Mist, this can be done by sending all the token funds to the organization address through the "Send" feature, which can be seen on figure 24.

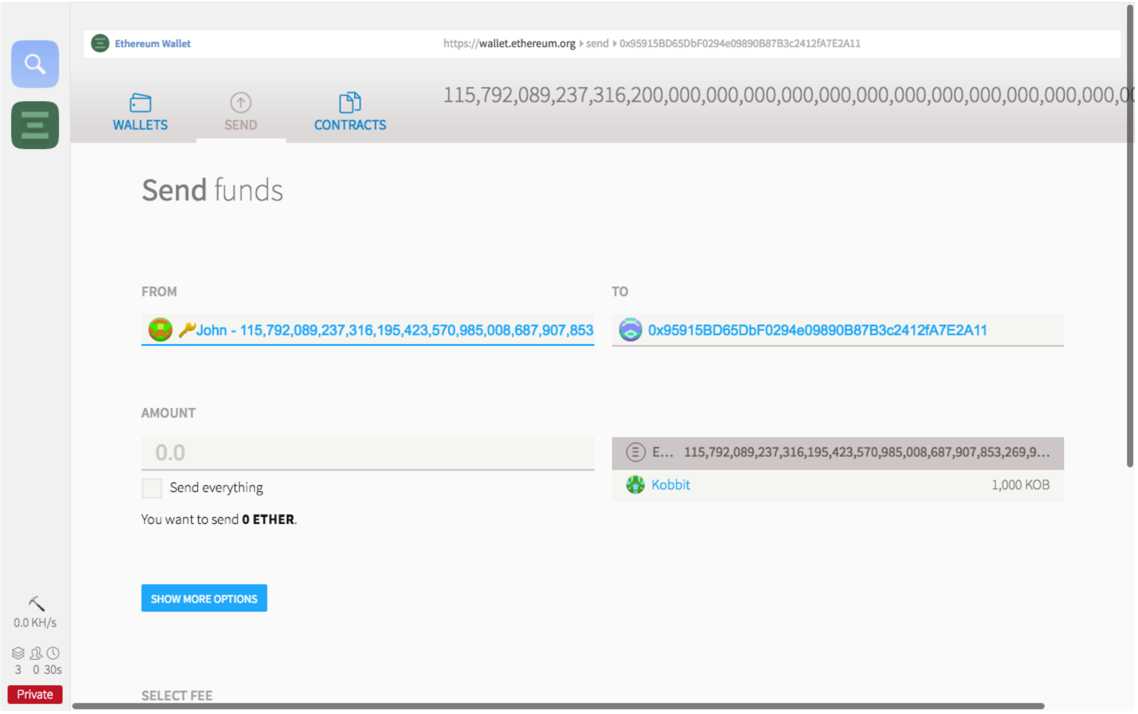

Figure 24. Send funds tab in Mist browser.

In order to transfer the ownership of the token contract, the user must go to the "Contracts" tab in Mist, which displays all the available contracts. By opening the token contract, the user will be able to use a function called "Transfer Ownership" to change the owner to the organization address, which can be seen on figure 25.

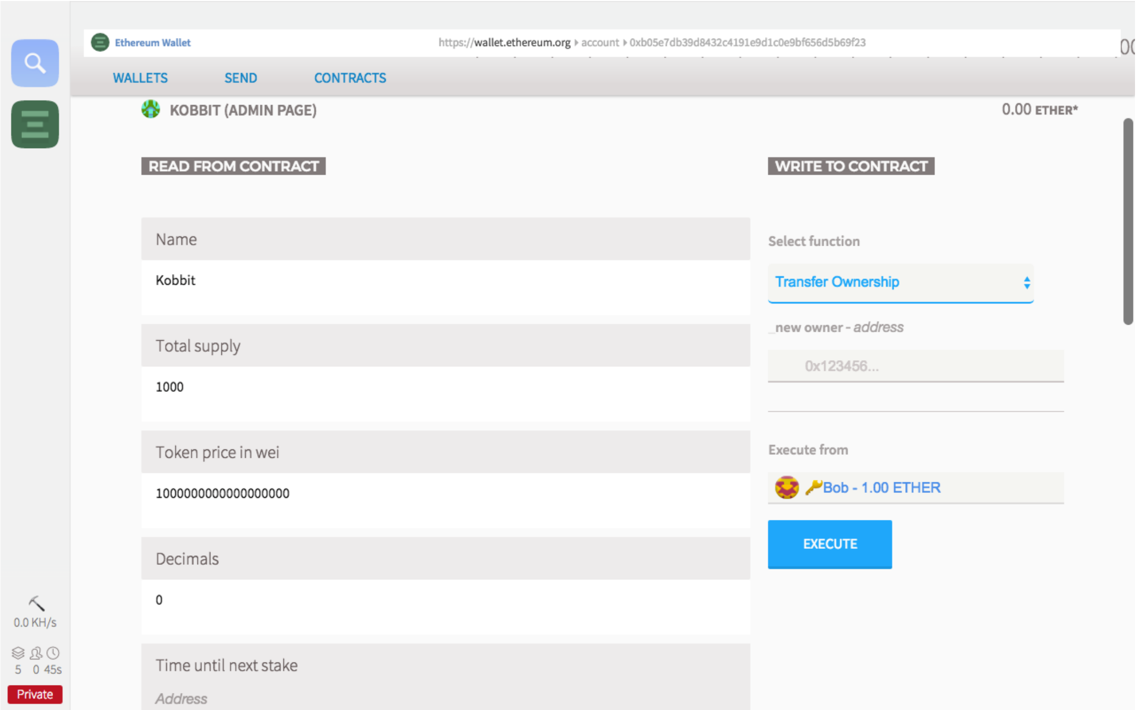

Figure 25. The interface of Kobbit contract in Mist.

5.4 The smart contract tests

In this thesis, the tests are done separately and manually in Mist's contract interface, which is displayed on figure 26. The functions, which are being tested, correlate to the functions in the organization contract, which is mentioned in section 5.2, as well as, the services of the DAO. In the sections below, the testing of each function is explained.

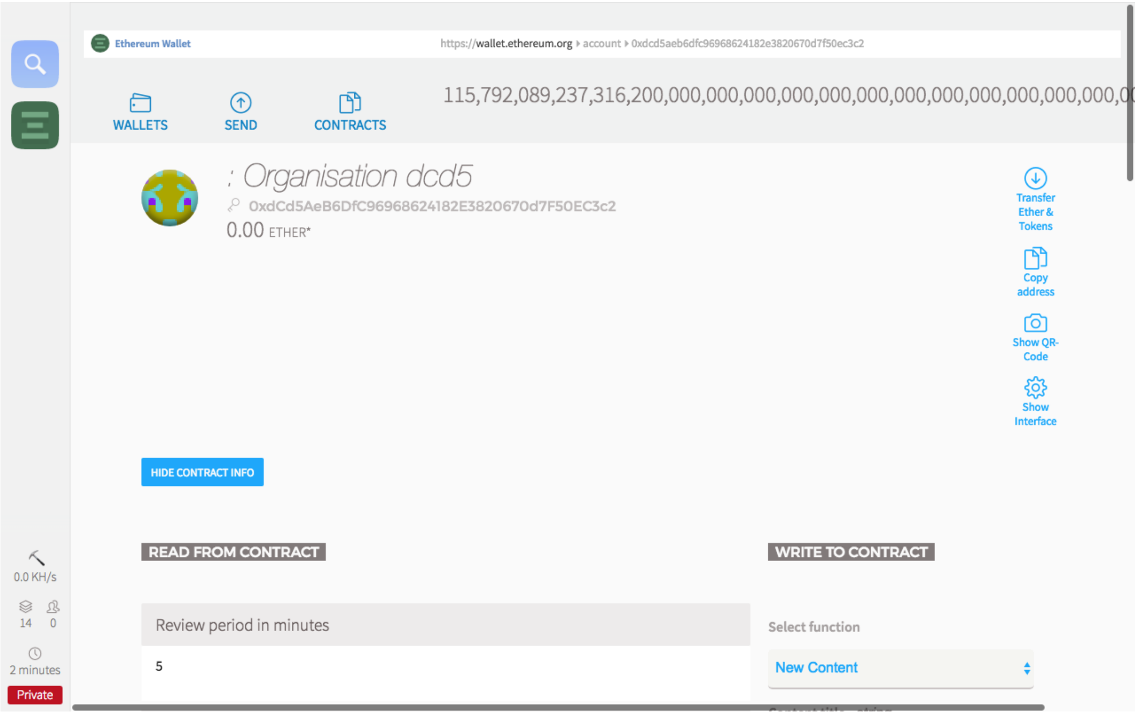

Figure 26. The interface of organization contract where the user can read from the contract, select and execute functions.

5.4.1 Stake

This function allows the user to create tokens based on the stake percentage of the deployed token contract and an account's total amount of token. To execute the function, the user has to input an account address, which in turn will create additional tokens to the given address. To test the function's reliability, it was executed twice within a time frame, in order to test the time interval of staking. Further, the newly created tokens were also checked if it corresponds to the stake percentage.

5.4.2 Change organization settings

This function changes the organization settings. To change the settings, the user has to fill in the address of token contract, content price, minutes for review, minimum reviews for content, margin of reviews for majority, and token reward for review. Afterwards, the function is executed, which in turn will change the settings on the organization contract interface. This was tested by changing the organization settings.

5.4.3 Purchase

This function allows the user to buy tokens from the organization. To execute the function, an Ether amount, in form of a whole number, needs to be input. Thereafter, an amount of token, equivalent to the Ether amount, is sent to the function caller from the organization contract. This was tested by executing the function from an account with Ether and checking the token balance afterwards.

5.4.4 Review content

This function allows a member to review a content and receive token as contribution reward. To test this function, the user has to input a content number of a published content. Afterwards, the user also has to approve or disapprove the content by ticking or unticking a checkbox. Also, a justification text has to be written. As a final result, this increments the number of reviews on the published content on the organization contract and transfers tokens from the organization to the user. This was tested on published content on the organization contract.

5.4.5 Appoint new CEO

This function appoints a new CEO on the organization contract. It requires an input of a new account address, which will be the new CEO. The expected output of execution is that the current CEO membership status should be changed to normal, while the new address becomes the CEO. This was tested by transferring the CEO status to another account.

5.4.6 Execute content

This function executes a content that will either be approved or disapproved, which depends on the reviews given. To run it, the user has to input a published content number that is past the review deadline. After executing, the content’s variable “content passed” is changed to either true or false, which represents approval and disapproval of that content. This was tested by executing and checking the content variable “content passed” on the organization contract interface.

5.4.7 New content

This function allows the users to post content to the organization contract. It requires the user to fill in the content title and description. In turn, it will create a content, which has variables such as id, publisher address, author, title, description, review deadline, number of reviews, executed and content passed. Afterward, the user can see the content information by checking the content id (content number) on the organization interface. This was tested by executing the function and checking the content in the organization contract interface.

5.4.8 Sell

This function allows the user to sell tokens for Ether. It requires the user to fill in the total amount of tokens that the user wants to sell. Afterward, executing the function will send Ether, equivalent to the total amount of tokens inserted, from the organization contract to the caller of the function. This was tested with an account with tokens.

5.4.9 Transfer ownership

This function allows ownership of the organization contract to be transferred to another account. This was tested by inserting an account address, which represents the new owner, and executing it. After executing, the owner field was changed on the organization contract. However, this can only be executed by the current owner of the contract.

5.4.10 Add Member

This function adds a member to the organization. It requires an input of an account address, a member name, and an account type. As an output, the function is expected to create a new member to the organization, which can be checked on the organization contract by inserting the member’s account address. This was tested by creating an account, inserting the address of that account, executing, and checking the address on the organization contract.

5.5 General Findings

To develop smart contracts, the user must be connected to a Ethereum blockchain to test the contracts, which in addition requires gas to be executed. As a result, this causes some disadvantages. Due to the fact that only one block is appended to the blockchain at a time, a throughput test is not possible. Also, because of the computation made on the blockchain is based on gas, this makes it difficult for the smart contract to resolve gas fee if the code is dynamic.

However, solo network does not require any gas fees, as it is run locally. As a result, this has both negative and positive effect on the smart contract development. The positive effect is instant transactions due to the local state of the blockchain, whereas, the opposite is that new transaction is only triggered if another transaction is made afterwards. Additionally, solo network does not simulate a real blockchain network, which often have multiple nodes. Instead, it is run with one node. For this reason, it would be difficult to test and find vulnerabilities that may occur in public testnet.

Also, the immutable state of contracts makes it more difficult to patch bugs that are found after deployment. With smart contract, the only way to secure the contracts is to cover all the loopholes in the internal code before being deployed. This requires the developer to be precise while writing smart contracts and be aware of the vulnerabilities that exist in smart contracts. In addition to safety measurements, the developer should also use the latest Solidity version, as it is more stable than previous versions. By enabling a kill switch (safety plug), this could allow the developers to self-destruct the contract, which could function as a final call for preventing extended attacks on afflicted smart contracts, however, this is not an efficient solution. Instead, developers should enable modification of important components (variables), which could fix the bugs in the smart contract after deployment, without jeopardizing the organization’s structure or idea.

Manual testing increases the development time significantly. This is not ideal for developing large smart contracts. By using manual testing, each of the organizations function has to be tested separately, which is time consuming. As a result, tracking bugs becomes unmanageable due to the size of the contracts. Ideally, automating the process would be better. This could be done with truffle framework (http://truffleframework.com), which allows unit testing for smart contracts.

6 CONCLUSION

Smart contracts have some advantages over traditional contracts. These advantages allow automation of business logics, but also allows two parties to enter an agreement that would otherwise be considered too risky, in terms of risk management, with traditional contract. As a result, this enables a trustless environment, where the procedures are handled by the Ethereum blockchain and smart contracts. Due to the fact that blockchain is a P2P network, this gives the benefit of an extreme level of fault tolerance, zero downtime, and an immutable data store. For these reasons, smart contracts can be used to build a DAO, Dapp or token, which could potentially help the advancement towards removing the reliance on trusted third parties in today's society.

On the other hand, there are also negative aspects with smart contracts. For example, executing smart contracts requires gas fee, which is not practical in live environment. Gas fee prevents the adoption of smart contracts, as it is difficult to resolve the fee if the smart contracts' code is dynamic, which is not ideal when making transactions. In addition, smart contracts also force the users to pay everything, the service fee, transaction fee, etc. Smart contracts are also immutable, which makes bugs impossible to patch. Code errors could be found after the deployment of the contracts, which could compromise the whole organization. As a result, the organization would become obsolete. Also, there are no method to securing the contracts from all the existing vulnerabilities, due to the fact that they are not well documented or discovered yet. Another issue with smart contracts are also that they are publicly displayed on the ledger, which also prevents information to be stored in secrecy. Hence, this forbids the organization to keep sensitive and secret information from the public. In addition, smart contracts can only be tested while being connected to a blockchain, which makes it difficult to do a performance test. On top of that, manual tests are not ideal for large contacts, as it consumes a lot of time. Consequently, this forces the developers to build unit tests, in order to complete a full smart contract test.

While all the requirements have been fulfilled, the design may not be secure. These functions may have hidden bugs. Discovering these bugs takes time and requires multiple test. Hence, the developer has to test all the possible way of executing the functions in order to secure the contracts, preferably even on the byte code level on the EVM. Also, since this thesis only focuses on implementing the requirements in the form of a functioning proof of concept, the organization is not designed to be operated in live environment. As a conclusion, smart contracts are still in an early development stage, but may improve further in the future.

REFERENCES

Atzei, N., Bartoletti, M. & Cimoli, T. 2017, *A Survey of Attacks on Ethereum Smart Contracts (SoK), Principles of Security and Trust,* eds. M. Maffei & M. Ryan, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 164.

Buterin, V., 2014. A next-generation smart contract and decentralized application platform. white paper.

Dannen C., 2017, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*, DOI 10.1007/978-1-4842-2535-6, Springer Science & Business Media New York, New York, USA.

Ethereum, 2017, *Solidity* [www], Available: https://solidity.readthedocs.io/en/v0.4.21 Retrieved 31.3.18.

Ethereum community, 2016, *Ethereum Homestead Documentation* [www], Available: http://www.ethdocs.org Retrieved 31.3.18.

JetBrains.org/IntelliJ, *What is IntelliJ IDEA Community Edition* [www], Available: http://www.jetbrains.org/pages/viewpage.action?pageId=983211 Retrieved 31.3.18.

Li, W., Andreina, S., Bohli, J. & Karame, G. 2017, *Securing Proof-of-Stake Blockchain Protocols, Data Privacy Management*, eds. J.Garcia-Alfaro, G. Navarro-Arribas, H. Hartenstein & J. Herrera-Joancomarti, Springer International Publishing, Cham, Cryptocurrencies and Blockchain Technology, pp. 297.

Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system.

serce, ll345374, 2018, *Intellij-Solidity* [www], Available: https://plugins.jetbrains.com/plugin/9475-intellij-solidity Retrieved 31.3.18.

Tikhomirov, S., 2017. Ethereum: state of knowledge and research perspectives.

Vogelstellar F., Buterin V., 2015, *ERC-20 Token Standard* [www], Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md Retrieved: 31.3.18

Yann300, 2017, *Remix – Solidity IDE* [www], Available: http://remix.readthedocs.io/en/latest/ Retrieved 31.3.18.

APPENDIX 1 SUMMARY IN SWEDISH

INTRODUKTION

Blockkedja är ett distribuerat datanätverk som är drivs av nätverksbehandlade noder. En nods uppgift är att utföra och registrera transaktioner som samlas in i block. Dessa block läggs till en publik huvudbok i följd. Dessutom är blocken också oföränderliga på grund av stark kryptografi, som säkrar nätverket. (Ethereum samfund 2016)

I början av 2014 föreslog och beskrev Vitalik Buterin i sin vitbok (Buterin 2014) en blockkedja som kan utföra alla arbiträra komplexa beräkningar. Som ett resultat ledde detta till skapandet av Ethereum, som kan implementera helt tillförlitligt smarta kontrakt. (Ethereum samfund 2016)

Syftet

Syftet med detta examensarbete är att skapa en token som möjliggör för en decentraliserad organisation (DAO) att fungera för en mediaorganisation (KSF) som också ställer krav på arbetet. Målet uppnås genom att bygga smarta kontrakt på Ethereum blockkedjan. Dessutom handlar det om att lära och förstå utmaningar i Ethereum och smart kontraktsutveckling.

I examensarbetet kommer följande frågor att besvaras:

- Vilka är fördelarna och nackdelarna med smart kontrakt?
- Hur testas smarta kontrakt?
- Vilka är kraven för att bygga de smarta kontrakten?

Avgränsning

Eftersom arbetet huvudsakligen fokuserar på Ethereum och smart kontraktsutveckling, kommer inga andra blockkedjor att täckas. I detta arbete är DAO och token tjänsterna (funktioner) begränsade till uppdragsorganisationens krav. Det innebär att inga ytterligare funktioner skapas om det inte är nödvändigt. I detta arbete testas smarta kontrakt manuellt.

Metoder

I detta arbetet används dokumentationer, böcker, artiklar och forum för att lära om Ethereum och smart kontraktsutveckling. De smarta kontrakten är byggda med hjälp av handledning, dokumentation och externa verktyg, samt frekvent testning av de smarta kontrakten med en webbläsare. De smarta kontrakten kommer att skrivas i integrerade utvecklingsmiljöer (IDE) som stöder Solidity.

PLANERINGSPROCESS

En plan konstruerades för arkitekturen av ett system som stöder en DAO. Den grundades på KSFs krav och fungerade som riktlinje för att bygga smarta kontraktet i detta examensarbete.

Koncept

Konceptet är att möjliggöra en decentraliserad mediaorganisation som drivs av smarta kontrakt. Genom smarta kontrakt kan organisationen automatisera en tjänst som annars skulle kräva administration. Tanken är också att transaktionerna ska hanteras genom blockkedjan. Som ett resultat gör det här möjligt för användarna att direkt interagera med tjänsterna i samband med organisationen.

En token är också skapad. Syftet med ett token är att fungera som valuta inom organisationen. Token finns som en betalningsmetod för att utföra tjänster, såväl som en belöning för att bidra till organisationen.

I det utformade systemet finns det också medlemskap för att bestämma myndighetsnivå i smarta kontrakt.

Krav

Detta avsnitt presenterar en lista över krav från KSF.
Dessa krav är:

- Skapa en token för organisationen.
- Skapa ett bevis på insats (POS) funktion för token.
- Skapa medlemskapssystem för organisationen.
- Tillåt köp och försäljning av token.

- Tillåt medlemmar att skapa innehåll genom att betala token.
- Tillåt medlemmar att granska innehåll. Vilket ger token belöning.
- Tillåt medlemmar att visa innehåll.
- Låt VD ändra organisationens inställningar.
- Alla transaktioner ska vara säkra
-

ETHEREUM

Ethereum är en Turing komplett blockkedja för decentraliserade applikationer. Blockkedjan innehåller ett protokoll för peer-to-peer (P2P) nätverk som kopplar noder till ett nätverk. I nätverket kör noderna Ethereum Virtuella Maskin (EVM), vilket är en exekveringsmiljö för smarta kontrakt. Dessutom har Ethereum fördelen av extrem nivå av fel tolerans, noll nedtid och ett oföränderligt datalager. (Ethereum samfund 2016)

Konton

I Ethereum är konton nödvändiga för att genomföra transaktioner och kommunicera med blockkedjan. Det finns två kontotyper: externt ägda konton och kontraktskonton. Dessa två kontotyper är likartade, men kontrolleras på olika sätt; medan externt ägda konto kontrolleras av privata nycklar, är kontraktskonton beordrade av sin interna kod, som endast kan utföras av externt ägda konton. Dessutom har alla konton också publika nycklar som används för att teckna transaktioner. (Ethereum samfund 2016)

Oavsett kontotyp har varje konto en balans, som kan hålla Ether. Dessutom har kontraktskonton också kontraktsförvaring. Dessa tillstånd uppdateras med varje nytt block på blockkedjan. (Ethereum samfund 2016)

Ether

Ethereum fungerar med en valuta som kallas Ether. Den används för att göra transaktioner och betala för EVM-beräkning i gas, vilket indirekt erhålls genom ether. Ether kan hållas av en externt ägda konton eller kontraktskonton. Den kan skickas till ett annat konto via kontons publika nyckel (adress). Transaktionerna kräver dock gas. Detta gäller också tokens som är delvalutor av Ether. (Ethereum samfund 2016)

Grävning

Grävning (Mining på engelska) är en definition för att öka den totala volymen Ether på blockkedjan. Metoden tillåter noder att lägga till nya block till blockkedjan genom att lösa en algoritm som kallas bevisa på arbete (proof of work på engelska) på Ethereum. Å andra sidan finns det också en alternativ metod för att nå konsensus i en blockkedja som kallas bevis på insats (proof of stake på engelska). (Ethereum samfund 2016 & Li et al. 2017)

Smarta kontrakt

Med hjälp av smarta kontrakt kan alla procedurer av traditionella kontrakt hanteras genom blockkedjan. Detta tillåter möjligheten att ingå i ett kontrakt, som annars skulle anses vara för riskabelt med traditionella kontrakt. (Ethereum samfund 2016)

Smarta kontrakt är en uppsättning av funktioner och stater som tillåter Turing fullständiga beräkningar. I Ethereum finns ett implementerat smart kontrakt på en specifik adress, där de finns i ett Ethereum-specifikt binärt format. Alla kontrakt genomförs på EVM, vilket gör det möjligt för dem att kommunicera med varandra. Smarta kontrakt kan användas för att skapa token, decentraliserade applikationer och organisationer. (Ethereum samfund 2016)

Smarta kontrakt är dock inte problemfria. Det finns sårbarheter som kan användas för att utnyttja kontrakt för illvilliga motiv. (Ethereum samfund 2016)

UTVECKLINGS VERKTYG

Solidity

I detta examensarbete används Solidity för att skriva de smarta kontrakten. Språket är ett kontrakt-orienterat programmeringsspråk som är influerades av C++, Python och JavaScript. Den stöder funktioner såsom arv, bibliotek och komplexa användardefinierade typer. (Ethereum 2017)

Remix

Remix är en integrerad utvecklingsmiljö som används för att utveckla smarta kontrakt. I detta examensarbetet används Remix tillsammans med Mist för att felsöka och testa smarta kontrakt. (Yann300 2017)

IntelliJ IDEA Community Edition

IntelliJ IDEA Community Edition är en integrerad utvecklingsmiljö som stöder funktioner som refaktorering, kodinspektioner, navigering och versionshanterings system. Den används i examensarbetet för att skriva smarta kontrakt, refaktorera koder, upprätthålla projekts struktur och visa syntaxhöjdpunkter genom alternativa plugin. (JetBrains.org/IntelliJ)

IntelliJ IDEA Community Edition installation

IntelliJ IDEA Community Edition installeras genom att exekvera en installationsfil som tillhandahålls av JetBrains (https://www.jetbrains.com/idea). IntelliJ kommer med begränsade; men ytterligare funktioner kan läggas till genom alternativa plugin. För att ta i bruk stöd för Solidity, måste användaren installera IntelliJ-Solidity plugin (serce & 11345374 2018) i IntelliJ plugininställningar.

Mist webbläsare

Mist webbläsare är en användarvänlig plånbok, som ansluter till Ethereum-nätverket. Det kan köra några funktioner av en fullständig nod t.ex., genomföra Ether transaktioner och smarta kontrakt. Dessutom kan webbläsaren också hålla konton. I examensarbetet används webbläsaren för att distribuera, testa och komma åt distribuerade smarta kontrakt. (Dannen 2017 s.21-22)

Mist webbläsare installation

Mist webbläsaren installeras genom att exekvera en installationsfil, som kan laddas ned från Ethereums officiella webbplats (https://www.ethereum.org). För att utnyttja funktionaliteten som erbjuds av webbläsaren måste användare välja en blockkedja att synkronisera med.

Geth

Geth är ett kommando gränssnittsverktyg för att köra en nod på Ethereum. Det implementeras med Go-språk för att interagera med blockkedjan. Genom Geth har användarna förmåga att bryta ner ether, överföra medel, skapa kontrakt och utforska blockhistorik. Geth är förinstallerad med Mist. (Ethereum samfund 2016)

Testnätet

Testnätet är ett alternativt Ethereum blockkedja som används för test. Utveckling i testnätet tillåter användaren att använda falska ether, som inte håller värde. (Dannen 2017 s.79)

GENOMFÖRINGS PROCESS

Smarta kontraktens arkitektur

De smarta kontrakten består av två huvudkomponenter: ett token- och organisationskontrakt. Den förstnämnda fungerar som organisationens tillgångar, medan den senare erbjuder tjänster till användarna.

Token kontraktet

Token kontraktet utnyttjar de funktioner som tillhandahålls av en ERC20Token, som är ett standardgränssnitt för token. ERC20Token innehåller variabler som bestämmer kontraktets egenskaper. Vidare innehåller den också en överföringsfunktion för token.

Kontraktet ärver också ytterligare två kontrakt från två ytterligare kontrakt som fungerar som stöd. Dessa är ”Owned” och ”Utils”. Den förstmända bestämmer ägaren av kontrakten, medan den senare utför matematiska beräkningar för de transaktioner som hanteras av kontraktet.

I kontraktet finns också extra funktioner som tillåter användare att köpa och sälja token, samt ett POS funktion.

Organisationskontraktet

Organisationskontraktet utnyttjar token kontraktet som en tillgång för att driva sitt ekosystem. Det är också byggt med ”Owned” och ”Utils” kontrakt som nämns tidigare. Dessutom innehåller kontraktet variabler som bestämmer organisationens inställningar.

Kontraktets funktioner är begränsade till endast medlemmar. Det betyder att ingen utomstående kan exekvera funktionerna. Funktionerna begränsas dock även utifrån användarnas auktoritetsnivå. Därmed tillåter detta organisationen att delegera makt enligt medlemskapstyper, som är ”Normal”, ”VD” och ”Medlem”.

De funktioner som implementeras i organisationskontraktet är: lägg till medlem, skapa nytt innehåll, granska innehåll, exekvera innehåll, POS, köp och sälj funktion för token, och funktion för att utse en ny VD. Dessa kan dock endast utföras om alla deras krav är uppfyllda.

Synkronisering med blockkedja

Solo-nätverket är en privat blockkedja som körs lokalt, som används som blockkedjan för att testa smarta kontrakt i detta examensarbetet. För att kunna synkronisera med blockkedjan måste användaren välja solo nätverk i nätverksfliken i Mist. Efter att ha valt och synkroniserats med blockkedjan får användaren en oändlig mängd falska ether för testning.

Ytterligare konton

Ytterligare konton skapas för att testa medlemskapsrestriktionerna för de smarta kontrakten. Detta görs genom Mists inbyggda funktion.

Distribuering av de smarta kontrakten

Distribueringsprocessen utförs genom Mists implementations funktion för smarta kontrakt. Innan token kontraktet och organisationskontraktet kan distribueras måste alla nödvändiga konstruktörparametrar fyllas. Efter distributionen av de smarta kontrakten måste all token leverans och äganderätten av token kontraktet överföras till organisationskontraktets adress. Detta är kritiska inställningar för att aktivera funktionerna i organisationen.

De smarta kontraktstesterna

I detta examensarbetet utförs tester separat och manuellt i Mists kontraktsgränssnitt. Funktioner som testas är: insats, ändra organisationsinställningar, inköp, granska innehållet, utnämna ny VD, utför innehåll, nytt innehåll, sälj, överför ägande, lägg till medlem. Dessa funktioner är viktiga för att bevisa ett koncepttest av en decentraliserad organisation.

Allmänna fynd

För att utveckla smarta kontrakt måste användaren vara ansluten till en Ethereum blockchain för att testa kontrakten, vilket också kräver gas för att utföras. Som ett resultat orsakar detta vissa nackdelar. På grund av att endast ett block kan adderas till blockkedjan åt gången, är ett genomgångstest inte möjligt. På grund av beräkningen på blockkedjan baseras på gas, gör det också svårt för smarta kontraktet att kalkylera gasavgift om koden är dynamisk.

Solo-nätverk kräver emellertid inga gasavgifter, eftersom det körs lokalt. Som ett resultat har detta både negativ och positiv effekt på den smarta kontraktsutvecklingen. Den positiva effekten är omedelbara transaktioner på grund av blockkedjans lokala tillstånd, medan motsatsen är att den nya transaktionen endast utlöses om en annan transaktion görs efteråt. Dessutom simulerar inte solo-nätverket en verklig blockkedja, som ofta har flera noder. Istället körs det med en nod. Av detta skäl skulle det vara svårt att testa och hitta sårbarheter som kan uppstå i det offentliga testnätet.

Också det oföränderliga tillståndet för kontrakt gör det svårare att fixa buggar som hittas efter utplacering. Med smarta kontrakt är det enda sättet att säkra kontrakten att täcka alla kryphål i den interna koden innan de distribueras. Detta kräver att utvecklaren är precis när han skriver smarta kontrakt och är medveten om de sårbarheter som finns i smarta kontrakt. Utöver säkerhetsmätningar bör utvecklaren också använda den senaste Solidity-versionen, eftersom den är stabilare än föregående versioner. Genom att möjliggöra en avstängningsknapp kan detta göra det möjligt för utvecklarna att förstöra kontraktet, vilket kan fungera som ett slutligt anrop för att förhindra förlängda attacker mot drabbade smarta kontrakt. Men detta är dock inte en effektiv lösning. I stället bör utvecklarna möjliggöra modifiering av viktiga komponenter (variabler), vilket skulle kunna fixa felen i de smarta kontrakten efter implementering utan att äventyra organisationens struktur eller idé.

Manuell testning ökar utvecklingstiden avsevärt. Detta är inte idealiskt för att utveckla stora smarta kontrakt. Genom att använda manuell testning måste varje organisationsfunktion testas separat, vilket är tidskrävande. Som ett resultat blir det svårt att hitta buggar på grund av kontraktens storlek. Helst skulle automatiseringen av processen vara bättre. Detta kan göras med truffel ramverk (http://truffleframework.com), vilket möjliggör enhetstestning för smarta kontrakt.

SLUTSATS

Smarta avtal har några fördelar jämfört med traditionella kontrakt. Dessa fördelar gör det möjligt att automatisera affärslogik, men tillåter även två parter att ingå ett avtal som annars skulle anses vara för riskabelt med traditionellt kontrakt. Som ett resultat möjliggör detta en decentraliserad miljö, där procedurerna hanteras av Ethereum blockkedjan och smarta kontrakt. På grund av blockkedjan är ett P2P-nätverk, ger detta fördel av en extrem nivå av feltolerans, noll nedtid och ett oföränderligt datalager. Av dessa skäl kan smarta kontrakt användas för att bygga en decentraliserad organisation, decentraliserad applikation eller token, som potentiellt kan bidra till att avlägsna tillit från betrodda tredje parter i dagens samhälle.

Å andra sidan finns det också negativa aspekter med smarta kontrakt. Till exempel kräver genomförandet av smarta kontrakt gasavgift, vilket inte är praktisk i levande miljö. Gasavgift förhindrar adoptionen av smarta avtal, eftersom det är svårt att kalkylera gasavgiften om de smarta kontraktens kod är dynamiska, vilket inte är idealiskt när man gör transaktioner. Dessutom tvingar smarta kontrakt också användarna att betala allt, serviceavgifterna, transaktionsavgifterna o.s.v. Smarta avtal är också oföränderliga, vilket gör att det inte går att fixa buggar. Buggar kan till exempel hittas efter distribueringen av smarta kontrakt, vilket skulle kunna kompromissa hela organisationen. Som ett resultat skulle organisationen bli obsolet. Det finns också ingen metod för att säkra kontrakten från alla befintliga sårbarheter, på grund av att de är inte väl dokumenterade eller upptäckta ännu. Ett annat problem med smarta kontrakt är också att de är publika, vilket också förhindrar information lagras i sekretess. Detta förbjuder organisationen att hålla känslig och hemlig information från allmänheten. Dessutom kan smarta kontrakt endast testas medan de är ansluten till en blockkedja, vilket gör det svårt att göra ett prestandatest. Utöver detta är manuell test inte idealisk för stora kontakter, eftersom det förbrukar mycket tid. Följaktligen tvingar det utvecklarna att bygga enhetsprov, för att slutföra ett helt smart kontraktstest.

Medan alla krav har uppfyllts kan det hända att konstruktionen inte är säker. Dessa funktioner kan ha gömda fel. Upptäcka dessa buggar tar tid och kräver flera test. Därför måste utvecklaren testa alla möjliga sätt att utföra funktionerna för att säkra kontrakten, helst även på byte-kodnivån på EVM. Eftersom detta examensarbete endast fokuserar på att genomföra kraven i form av ett fungerande koncepttest, är organisationen inte avsedd att drivas i levande miljö. Som en slutsats är smarta avtal fortfarande i ett tidigt utvecklingsstadium, men kan förbättras ytterligare i framtiden.

KÄLLOR

Atzei, N., Bartoletti, M. & Cimoli, T. 2017, *A Survey of Attacks on Ethereum Smart Contracts (SoK), Principles of Security and Trust,* eds. M. Maffei & M. Ryan, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 164.

Buterin, V., 2014. A next-generation smart contract and decentralized application platform. vitbok.

Dannen C., 2017, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*, DOI 10.1007/978-1-4842-2535-6, Springer Science & Business Media New York, New York, USA.

Ethereum, 2017, *Solidity* [www], Tillgänglig: https://solidity.readthedocs.io/en/v0.4.21 Hämtad 31.3.18.

Ethereum samfund, 2016, *Ethereum Homestead Documentation* [www], Tillgänglig: http://www.ethdocs.org Hämtad 31.3.18.

JetBrains.org/IntelliJ, *What is IntelliJ IDEA Community Edition* [www], Tillgänglig: http://www.jetbrains.org/pages/viewpage.action?pageId=983211 Hämtad 31.3.18.

Li, W., Andreina, S., Bohli, J. & Karame, G. 2017, *Securing Proof-of-Stake Blockchain Protocols*, *Data Privacy Management*, eds. J.Garcia-Alfaro, G. Navarro-Arribas, H. Hartenstein & J. Herrera-Joancomarti, Springer International Publishing, Cham, Cryptocurrencies and Blockchain Technology, pp. 297.

serce, ll345374, 2018, *Intellij-Solidity* [www], Tillgänglig: https://plugins.jetbrains.com/plugin/9475-intellij-solidity Hämtad 31.3.18.

Vogelstellar F., Buterin V., 2015, *ERC-20 Token Standard* [www], Tillgänglig: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md Retrieved: 31.3.18

Yann300, 2017, *Remix – Solidity IDE* [www], Tillgänglig: http://remix.readthedocs.io/en/latest/ Hämtad 31.3.18.

APPENDIX 2 KOBBITTOKEN.SOL


```
pragma solidity ^0.4.15;

/*
Owner contract that determines the owner. Includes a transfer function of ownership.
*/
contract Owned {
    address public owner; // The owner address.

    event OwnerUpdate(address newOwner); // Event listener for owner update.

    /*
    @dev constructor is activated when the contract is deployed.
    */
    function Owned() public {
        owner = msg.sender; // Set the msg.sender to owner.
    }

    /*
    @dev checks if the owner is the msg.sender.
    */
    modifier ownerOnly {
        assert(msg.sender == owner);
        _;
    }

    /*
    @dev transfer the ownership to a new owner.

    @param _newOwner the address of the new owner.
    */
    function transferOwnership(address _newOwner) public ownerOnly {
        require(owner != _newOwner); // Checks if the owner equals _newOwner.
        owner = _newOwner; // Set the owner to new owner.
        emit OwnerUpdate(owner); // Send to event listener.
    }
}

/*
Provides utilities and support for other contracts like controlling calculation.
*/
contract Utils {

    /**
        @dev constructor.
    */
    function Utils() public {
    }

    /**
    @dev checks the validity of the address.

    @param _address the address being checked.
    @return success if calculations are correct.
    */
    function check(address _address) public pure returns (bool success) {
        assert(_address != 0x0);
        return true;
    }

    /**
    @dev checks the addition math between two variables.

    @param _x first variable.
    @param _y second variable.
    @return z if calculations are correct.
    */
    function add(uint256 _x, uint256 _y) public pure returns (uint256) {
        uint256 z = _x + _y;
        assert(z >= _x);
        return z;
    }

    /**
    @dev checks the subtraction math between two variables.

    @param _x first variable.
    @param _y second variable.
    @return z if calculations are correct.
    */
    function sub(uint256 _x, uint256 _y) public pure returns (uint256) {
        assert(_x >= _y);
        return _x - _y;
    }

    /**
    @dev checks the multiplication math between two variables.

    @param _x first variable.
    @param _y second variable.
    @return z if calculations are correct.
    */
    function mul(uint256 _x, uint256 _y) public pure returns (uint256) {
        uint256 z = _x * _y;
        assert(_x == 0 || z / _x == _y);
        return z;
    }

    /**
    @dev checks the division math between two variables.

    @param _x first variable.
```

```
    @param _y second variable.
    @return z if calculations are correct.
    */
    function div(uint256 _x, uint256 _y) public pure returns (uint256) {
        uint256 z = _x / _y;
        return z;
    }
}

/*
Basic token functions based on ERC20Token. Base for KobbitToken.
*/
contract Token is Utils{
    string public name; // The name of the token.
    string public symbol; // The symbol of the token.
    uint8 public decimals; // Amount of decimals in token number.
    uint256 public totalSupply; // The total supply of the token.

    mapping (address => uint256) public balanceOf; // Checks the balance of an address.
    mapping (address => mapping (address => uint256)) public allowance; // Checks the allowance that another sender
have.

    event Transfer(address indexed from, address indexed to, uint256 value); // Transfer broadcast message.
    event Approval(address indexed owner, address indexed _spender, uint256 _value); // Approval broadcast message.

    /*
    @dev the constructor that initiate initialSupply, tokenName, decimalUnits, tokenSymbol. Sets variable values.

    @param initialSupply the starting supply amount.
    @param tokenName the name of the token.
    @param decimalUnits decimals unit for token supply.
    @param tokenSymbol the symbol for the token.
    */
    function Token(
        uint256 initialSupply,
        string tokenName,
        uint8 decimalUnits,
        string tokenSymbol
        ) public {
        balanceOf[msg.sender] = initialSupply; // Give the owner all of the token supply.
        totalSupply = initialSupply; // The initialSupply equals totalSupply.
        name = tokenName; // Give the token a name.
        symbol = tokenSymbol; // Symbol of the token.
        decimals = decimalUnits; // Given decimal units.
    }

    /*
    @dev transfer token from msg.sender to an address.

    @param _to the address that token will be sent to.
    @param _value token value sent.
    @return true if successful.
    */
    function transfer(address _to, uint256 _value)
        public
        returns (bool success)
    {
        check(_to); // Checks if the address is valid.
        balanceOf[msg.sender] = sub(balanceOf[msg.sender], _value); // Substract from msg.sender.
        balanceOf[_to] += add(balanceOf[_to], _value); // Add to address given in param.
        emit Transfer(msg.sender, _to, _value); // Send broadcast message.
        return true;
    }

    /*
    @dev approves a address to spend on your behalf.

    @param _spender address of the spender.
    @param _value allowance token amount.
    @return true if successful.
    */
    function approve(address _spender, uint256 _value)
        public
        returns (bool success)
    {
        check(_spender); // Checks if _spender is a valid address.
        allowance[msg.sender][_spender] = _value; // Give allowance to _spender.
        emit Approval(msg.sender, _spender, _value); // Broadcast a message about the approval.
        return true;
    }

    /*
    @dev transfer token from an address to an address.

    @param _from the token address being sent from.
    @param _to the token address being sent to.
    @param _value token value being sent.
    @return returns true if successful.
    */
    function transferFrom(address _from, address _to, uint256 _value)
        public
        returns (bool success)
    {
        check(_from); // Checks validity of address _from.
        check(_to); // Checks validity of address _to.
        allowance[_from][msg.sender] = sub(allowance[_from][msg.sender], _value); // Substract from allowance.
        balanceOf[_from] = sub(balanceOf[_from], _value); // Substract from address _from.
        balanceOf[_to] = add(balanceOf[_to], _value); // Add to address _to.
        emit Transfer(_from, _to, _value); // Broadcast transfer listenener.
        return true;
    }
```

```
}

/*
The smart token contract that has addition functions on-top of Token contract.
Functionalities, that handles the smart token, are implemented in this contract.
*/
contract Kobbit is Owned, Token {
    uint public stakeRateInterest; // The interest rate of staking.
    uint public tokenPriceInWei; // The price per token.
    uint public stakeIntervalInMinutes; // The time interval between stakes.
    mapping (address => uint) public timeUntilNextStake; // The time until next available stake.

    event Staked(address stakingAddress, uint stakeAmount); // Broadcast if token is being staked.
    event TokenBought(address buyer, uint tokenAmount, uint etherAmount); // Broadcast if token is being bought.
    event TokenSold(address seller, uint tokenAmount); // Broadcast if token is being sold.
    event PaymentConfirmed(address from, uint tokenAmount); // Broadcast if payment is confirmed.

    /*
    @dev checks if the address is allowed to stake.

    @param check the address being checked.
    */
    modifier allowStake(address check) {
        assert(now >= timeUntilNextStake[check]);
        _;
    }

    /*
    @dev constructor that uses the basic token address to set variables.
    Setup the kobbit contract first time.

    @param initialSupply the initial supply of token.
    @param tokenName name of the token.
    @param decimalUnits assign decimal units for token.
    @param tokenSymbol assign the token symbol.
    @param interestRateInPercentage the interest rate of staking.
    @param etherCostOfEachToken the cost of token per ether.
    @param minutesBetweenStaking the time interval between stake.
    */
    function Kobbit(
        uint256 initialSupply,
        string tokenName,
        uint8 decimalUnits,
        string tokenSymbol,
        uint interestRateInPercentage,
        uint etherCostOfEachToken,
        uint minutesBetweenStaking
    ) public Token (initialSupply, tokenName, decimalUnits, tokenSymbol) {
        stakeRateInterest = interestRateInPercentage; // Set stakeRateInterest.
        tokenPriceInWei = etherCostOfEachToken * 1 ether; // Set the price of token per ether.
        stakeIntervalInMinutes = minutesBetweenStaking; // Set the time interval between stakes.
    }

    /*
    @dev staking function that stakes to an address.

    @param _to address being staked.
    @return true if successful.
    */
    function stake(address _to)
        public
        allowStake(_to)
        ownerOnly
        returns (bool success)
    {
        uint temp = mul(balanceOf[_to], stakeRateInterest); // Multiplies the balanceOf address with stakeRateInterest.
        uint stakeValue = div(temp, 100); // Divide the temporary value with 100, and we get stakeValue. Float numbers doesn't exist in EVM.
        balanceOf[_to] = add(balanceOf[_to], stakeValue); // Add the stakeValue to balanceOf address.
        totalSupply = add(totalSupply, stakeValue); // Add tokens to totalSupply.
        timeUntilNextStake[_to] = now + stakeIntervalInMinutes * 1 minutes; // Set the timer for next staking possibility.

        emit Staked(_to, stakeValue); // Broadcast a message.
        return true;
    }

    /*
    @dev buy function that can be executed by owner or contract.

    @param from the address buying the token.
    @param amount amount of token being bought.
    @param return true if successful.
    */
    function buy(address from, uint amount)
        public
        ownerOnly
        returns (bool success)
    {
        balanceOf[from] = add(balanceOf[from], amount); // Add the amount to given address.
        balanceOf[msg.sender] = sub(balanceOf[msg.sender], amount); // Substract from the msg.sender.

        emit TokenBought(from, amount, amount / 1 ether); // Broadcast how many tokens were bought.
        return true;
    }

    /*
    @dev sell function that can be executed by owner or contract.

    @param from the address selling the token.
    @param amount amount of token being sold
```

```
    @param return true if successful.
    */
    function sell(uint256 amount, address from)
        public
        ownerOnly
        returns (bool success)
    {
        balanceOf[msg.sender] = add(balanceOf[msg.sender], amount); // Add the amount to msg.sender.
        balanceOf[from] = sub(balanceOf[from], amount); // Substract from the address from.

        emit TokenSold(from, amount); // Broadcast the amount of token sold.
        return true;
    }

    /*
    @dev pay function that transfers token.

    @param _from the address that's paying.
    @param _amount amount being paid.
    @return true if successful.
    */
    function pay(address _from, uint256 _amount)
        public
        ownerOnly
        returns (bool success)
    {
        balanceOf[_from] = sub(balanceOf[_from], _amount); // Substract from the payer.
        balanceOf[msg.sender] = add(balanceOf[msg.sender], _amount); // Add to the msg.sender.

        emit PaymentConfirmed(_from, _amount); // Broadcast a message, payment confirmed.
        return true;
    }

    // Kill of the contract and return ether to owner.
    function kill() public { if (msg.sender == owner) selfdestruct(owner); }
}
```

APPENDIX 3 ORGANISATION.SOL

File - /home/brian/Downloads/folder/Organisation.sol

```
pragma solidity ^0.4.15;

/*
Owner contract that determines the owner. Includes a transfer function of ownership.
*/
contract Owned {
    address public owner; // The owner address.

    event OwnerUpdate(address newOwner); // Event listener for owner update.

    /*
    @dev constructor is activated when the contract is deployed.
    */
    function Owned() public {
        owner = msg.sender; // Set the msg.sender to owner.
    }

    /*
    @dev checks if the owner is the msg.sender.
    */
    modifier ownerOnly {
        assert(msg.sender == owner);
        _;
    }

    /*
    @dev transfer the ownership to a new owner.

    @param _newOwner the address of the new owner.
    */
    function transferOwnership(address _newOwner) public ownerOnly {
        require(owner != _newOwner); // Checks if the owner equals _newOwner.
        owner = _newOwner; // Set the owner to new owner.
        emit OwnerUpdate(owner); // Send to event listener.
    }
}

/*
Token interface with functions and variable returns.
*/
contract Token {
    //Get the token price in wei.
    function tokenPriceInWei() public constant returns (uint);
    //Transfer to an address from
    function transfer(address _to, uint256 _value) public returns (bool success);
    //Require permission to use this function. Transfer token from a given address.
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success);
    //Approve a certain address to spend on their behalf.
    function approve(address _spender, uint256 _value) public returns (bool success);
    //Stake function based on percentage and time limit.
    function stake(address _to) public returns (bool success);
    //Buy tokens with this function.
    function buy(address from, uint amount) public returns (bool success);
    //Sell tokens and receive ether from the contract.
    function sell(uint256 amount, address from) public returns (bool success);
    //Pay token by transfering from an address to the contract.
    function pay(address _from, uint256 _amount) public returns (bool success);
}
/*
Provides utilities and support for other contracts like controlling calculation.
*/
contract Utils {

    /**
        @dev constructor.
    */
    function Utils() public {
    }

    /**
    @dev checks the validity of the address.

    @param _address the address being checked.
    @return success if calculations are correct.
    */
    function check(address _address) public pure returns (bool success) {
        assert(_address != 0x0);
        return true;
    }

    /**
    @dev checks the addition math between two variables.

    @param _x first variable.
    @param _y second variable.
    @return z if calculations are correct.
    */
    function add(uint256 _x, uint256 _y) public pure returns (uint256) {
        uint256 z = _x + _y;
        assert(z >= _x);
        return z;
    }

    /**
    @dev checks the subtraction math between two variables.

    @param _x first variable.
    @param _y second variable.
    @return z if calculations are correct.
    */
    function sub(uint256 _x, uint256 _y) public pure returns (uint256) {
```

```
        assert(_x >= _y);
        return _x - _y;
    }

    /**
    @dev checks the multiplication math between two variables.

    @param _x first variable.
    @param _y second variable.
    @return z if calculations are correct.
    */
    function mul(uint256 _x, uint256 _y) public pure returns (uint256) {
        uint256 z = _x * _y;
        assert(_x == 0 || z / _x == _y);
        return z;
    }

    /**
    @dev checks the division math between two variables.

    @param _x first variable.
    @param _y second variable.
    @return z if calculations are correct.
    */
    function div(uint256 _x, uint256 _y) public pure returns (uint256) {
        uint256 z = _x / _y;
        return z;
    }
}

/*
The Organisation that controls functions that the token has.
*/
contract Organisation is Utils, Owned{
    uint public minimumReviews; // Minimum amount of reviews required for approval.
    uint public majorityMarginForContent; // Majority margin for content.
    uint public reviewReward; // Review reward for reviewing a content.
    Token public tokenAddress;  // The token address being used by the organisation.
    uint public contentPrice; // Price for publishing a content.
    uint public reviewPeriodInMinutes;  // Period for a content being open for reviewing.
    mapping (address => Member) public member; // Member information linked to address.
    enum Account {Normal, CEO, Member} // Different type of member privileges.
    Content[] public contents; // Stores the content in a array list.
    uint public numberOfContents; // The total amount of content right now.

    //Event listener that activates when organisation settings are changed.
    event OrganisationSettings(
        Token updatedTokenAddress,
        uint updatedTokenPrice,
        uint updatedMinimumReviews,
        uint updatedMajorityMarginForContent,
        uint updatedReviewReward
    );
    //Event listener that activates when content is added.
    event ContentAdded(uint contentID, string author, string title, string description);
    //Event listener that activates when content is reviewed.
    event Reviewed(uint contentID, bool position, address reviewer, string justification);
    //Event listener that activates when content is executed.
    event ContentTallied(uint contentID, uint result, uint quorum, bool active);

    struct Content {
        address publisherAddress; // The address of the publisher.
        string author; // The author of the content.
        string title; // The title.
        string description; // The description of the content.
        uint reviewDeadline; // The deadline for content being reviewed.
        uint currentResult; // The current result / points of reviewing.
        uint numberOfReviews; // The total amount of reviews.
        bool executed; // Checks if the content executed.
        bool contentPassed; // Checks if the content passed.
        mapping (address => bool) reviewed; // Checks the reviewers permission.
    }

    struct Member { // Address should be added to the struct.
        address memberAddress;
        string name; // Name of the member.
        uint registered; // Registration date.
        Account privilege; // Account privilege.
    }

    // Allows execution by the CEO only or the this contract.
    modifier ceoOnly {
        assert(member[msg.sender].privilege == Account.CEO
            || this == msg.sender);
        _;
    }

    // Allows execution by Members and CEO only.
    modifier memberOnly {
        assert(member[msg.sender].privilege == Account.Member
            || member[msg.sender].privilege == Account.CEO);
        _;
    }

    /**
    @dev the constructor of the organisation, which is executed when you first
    time setup.

    @param token the address of token being used in this organisation.
    @param pricePerContent the price per content.
    @param minutesForReview Sets the period of reviewing a content.
```

```
    @param minimumReviewsForContents minimal amount of reviews required for content.
    @param marginOfReviewsForMajority the amount of reviews for margin.
    @param tokenRewardForReview the token reward for reviewing.
    */
    function Organisation(
        Token token,
        uint pricePerContent,
        uint minutesForReview,
        uint minimumReviewsForContents,
        uint marginOfReviewsForMajority,
        uint tokenRewardForReview)
        public
        payable
    {
        // Add the organisation creator as the CEO.
        addMember(msg.sender, "Founder", Account.CEO);
        /*
        Function that changes organisation settings.
        Check the corresponding function for more information.
        */
        changeOrganisationSettings(
            token,
            pricePerContent,
            minutesForReview,
            minimumReviewsForContents,
            marginOfReviewsForMajority,
            tokenRewardForReview
        );
    }

    /**
    @dev changes the setting of the organisation.

    @param token the address of token being used in this organisation.
    @param pricePerContent the price per content.
    @param minutesForReview Sets the period of reviewing a content.
    @param minimumReviewsForContents minimal amount of reviews required for content.
    @param marginOfReviewsForMajority the amount of reviews for margin.
    @param tokenRewardForReview the token reward for reviewing.
    */
    function changeOrganisationSettings(
        Token token,
        uint pricePerContent,
        uint minutesForReview,
        uint minimumReviewsForContents,
        uint marginOfReviewsForMajority,
        uint tokenRewardForReview)
        public
        ceoOnly
    {
        tokenAddress = Token(token); // Set the token address.
        contentPrice = pricePerContent; // Set the content price.
        reviewPeriodInMinutes = minutesForReview; // Set the review period.
        minimumReviews = minimumReviewsForContents; // Set the minimum reviews.
        majorityMarginForContent = marginOfReviewsForMajority; // Set the majority margin.
        reviewReward = tokenRewardForReview; // Set the token reward.

        /*
        Activates the event listener with the corresponding name.
        */
        emit OrganisationSettings(
            tokenAddress,
            contentPrice,
            minimumReviews,
            majorityMarginForContent,
            reviewReward
        );
    }

    /*
    @dev the function adds a member to the organisation.

    @param targetMember the address being added.
    @param memberName the name of the member.
    @param accountType the privilege of the new member.
    */
    function addMember( // Need a second function that uses this function when ownership is transferred to this contract.
        address targetMember,
        string memberName,
        Account accountType)
        public
        ownerOnly
    {
        require(member[targetMember].memberAddress != targetMember); // Checks if the targetMember has an memberAddress.
        member[targetMember] = Member(targetMember, memberName, now, accountType); //Sets the targetMember to member mapping.
    }

    /*
    @dev enables members to add new content to the organisation by input title
    and description. The author will be automatically set to the function executor.

    @param contentTitle the title of the content.
    @param contentDescription description of the content.
    */
    function newContent(string contentTitle, string contentDescription)
        public
        memberOnly
    {
```

```
        tokenAddress.pay(msg.sender, contentPrice); // Uses the token address function to pay to this contract.
        uint contentID = contents.length++; // Set ID based on contents.length++.
        Content storage c = contents[contentID]; // Creates content struct that holds the content information.
        c.publisherAddress = msg.sender; // Set the publisher address.
        c.author = member[msg.sender].name; // Set the author of the content.
        c.title = contentTitle; // Set the content title.
        c.description = contentDescription; // Set the description of the content.
        c.reviewDeadline = now + reviewPeriodInMinutes * 1 minutes; // Deadline for reviewing content.
        c.numberOfReviews = 0; // Amount of reviews.
        c.executed = false; // Checks if the content is executed.
        c.contentPassed = false; // Checks if content passed the criterias.

        emit ContentAdded(contentID, c.author, c.title, c.description); // Event listener activated.
        numberOfContents = contentID+1; // Add the total number of content in organisation with 1.
    }

    /*
    @dev review function that is being used to review and award the reviewer

    @param contentNumber the number of the content.
    @param supportsContent bool that determines support for the content.
    @param justificationText additional text that the reviewer wants to add.
    */
    function reviewContent(
        uint contentNumber,
        bool supportsContent,
        string justificationText
    )
        public
        memberOnly
    {
        Content storage c = contents[contentNumber]; // Checks a content based on number.
        require(now < c.reviewDeadline); // Checks if time is within deadline.
        require(c.publisherAddress != msg.sender); // Checks that the reviwer is not the publisher.
        require(c.reviewed[msg.sender] == false); // Checks if the reviewer reviewed.
        c.reviewed[msg.sender] = true; // Sets review to true
        c.numberOfReviews++; // Increase reviews by one.
        if (supportsContent) {
            c.currentResult++; // If the content is supported, increase by one.
        } else {
            if(c.currentResult > 0) {
            c.currentResult--; // If the reviewer does not support, decrease by one.
            }
        }

        tokenAddress.transfer(msg.sender, reviewReward); // Transfer tokens from this contract to the function executor
.
        emit Reviewed(contentNumber, supportsContent, msg.sender, justificationText); // Event listener activated based
 on the variables.
    }

    /*
    @dev executes content to see if it passed or not. It can only be activated after deadline.

    @param contentNumber the number of content being executed.
    */
    function executeContent(uint contentNumber) public
        memberOnly
    {
        Content storage c = contents[contentNumber]; // Sets a content based on number.
        require(now > c.reviewDeadline // Checks if the deadline has passed.
          && !c.executed // Checks if the content has been executed or not.
          && c.numberOfReviews > minimumReviews); // Checks if content passed minimum review criteria.
        if (c.currentResult >= majorityMarginForContent) { // Checks if the content passed a certain margin.
            c.executed = true; // Execute.
            c.contentPassed = true; // Passed.
        } else {
            c.executed = true; // Execute.
            c.contentPassed = false; // Failed.
        }

        emit ContentTallied(contentNumber, c.currentResult, c.numberOfReviews, c.contentPassed); // Content information
 being broadcasted.
    }

    /*
    @dev stake function that works with members only. It uses the tokens stake function.

    @param to the address being staked.
    */
    function stake(address to) public
        memberOnly
    {
        require(member[to].memberAddress == to); // Checks if the address of member equals the one to.
        tokenAddress.stake(to); // Stake based on the function on token address.
    }

    /*
    @dev purchase token based on token price set by token address.
    */
    function purchase() // Does not give the exact amount because float doesn't exist.
        public
        memberOnly
        payable
    {
        uint amount = div(msg.value, tokenAddress.tokenPriceInWei()); // Calculates the amount of token that will be
bought.
        tokenAddress.buy(msg.sender, amount); // Buy function in token address.
    }

    /*
```

```
    @dev sell token based on amount set.

    @param amount the amount of token.
    */
    function sell(uint256 amount) public
        memberOnly
    {
        tokenAddress.sell(amount, msg.sender); // Sell the token to this contract.
        msg.sender.transfer(amount * 1 ether); // This contract send ether to the msg.sender.
    }

    /*
    @dev appoints a new ceo function.

    @param to the address given will be the new ceo.
    */
    function appointNewCEO(address to)
        public
        ceoOnly
    {
        require(member[to].memberAddress == to); // This function is broken. Checks if the address to is registered.
        check(to); // Checks if the address is valid.
        require(to != msg.sender); // Checks if the msg.sender is the address to.
        member[msg.sender].privilege = Account.Member; // Sets current CEO to Member.
        member[to].privilege = Account.CEO; // Address to gets appointed as CEO.
    }

    /*
    @dev kill the organisation function. msg.sender will receive all ether.
    */
    function kill() public ceoOnly { selfdestruct(msg.sender); }
}
```