MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Computer Science (CS)

CRYPTOCURRENCY TOKEN FOR INTERNATIONAL SETTLEMENTS

Author:
Luca Srdjenovic

Under the direction of:
Prof. Dr. Ninoslav Marina
HE-Arc

External expert:

Lausanne, HES-SO//Master, May 28, 2023

Information about this report

Contact information

Author: Luca Srdjenovic
MSE Student
HES-SO//Master
Switzerland
Email: *luca.srdjenovic@master.hes-so.ch*

Declaration of honor

I, undersigned, Luca Srdjenovic, hereby declare that the work submitted is the result of a personal work. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and the author quotes were clearly mentioned.

Place, date: _____

Signature: _____

Validation

Accepted by the HES-SO//Master (Switzerland, Lausanne) on a proposal from:

Prof. Dr. Ninoslav Marina, Thesis project advisor
, , Main expert

Place, date: _____

Prof. Dr. Ninoslav Marina
Advisor

Dean, HES-SO//Master

Abstract

Key words: Blockchain, Cryptography, Ethereum, Smart Contracts, ERC20

Contents

1 Analysis

This chapter is a comprehensive review of the tools, frameworks, and standards available to us. By conducting this analysis, we can identify the project constraints, evaluate the feasibility, and make informed decisions about the project's design and implementation. Our project focuses on building a decentralized platform for international settlements to provide a more efficient and cost-effective alternative to the current international settlement systems. As part of this analysis, we aim to create our own cryptocurrency token digitally representing value within our platform. Additionally, our platform must be able to facilitate the exchange of our cryptocurrency token for other cryptocurrencies and fiat currencies.

In section 1.1, we explore Ethereum as a decentralized application platform. We examine the Ethereum blockchain, its core features, architecture, and the Ethereum Virtual Machine (EVM) role, enabling the execution of smart contracts. This section is fundamental to our project as we will be using Ethereum as the foundation of our platform.

In section 1.2, we delve into Solidity, the primary programming language used to develop smart contracts on the Ethereum blockchain. We present its syntax, features, and how it facilitates the development of smart contracts of secure and robust smart contracts that will be used as the backend of our platform.

In section 1.3, we highlight the importance of token standards in the Ethereum ecosystem. We explore various token standards, such as ERC-20, ERC-223, ERC-721, and ERC-777, and how they are used to create tokens on the Ethereum blockchain, and evaluate which token standard is best suited for our platform cryptocurrency token.

In section 1.4, we explore the ERC-20 token standard, the most widely used token standard in the Ethereum ecosystem. We analyze its features, methods and how it aligns with our project requirements for creating versatile and interoperable tokens for international settlements and token exchange.

In section 1.5, we introduce OpenZeppelin, a reputable open-source framework for building secure smart contracts on the Ethereum blockchain. We discuss how it can enhance the security and reliability of our smart contracts implementation, providing us with a solid foundation for building our cryptocurrency token and ensuring a secure exchange of our token for other cryptocurrencies and fiat currencies.

The section 1.6 explores Truffle, a development environment and testing framework for Ethereum. We discuss the advantages of Truffle for our project, including its ability to automate our smart contracts' compilation, deployment, and testing. It will aid us in efficiently developing and deploying our decentralized platform.

Finally, section 1.7 introduces Web3.js, a JavaScript library allowing our web application to interact with the Ethereum network. We explain why web3.js is essential for our project, as it simplifies integrating Ethereum functionality into our frontend web application. This allows us seamless interaction with our cryptocurrency token and the Ethereum network.

1.1 Ethereum, a Decentralized Application Platform

The Ethereum blockchain is a decentralized platform that runs smart contracts. Smart contracts are applications that run exactly as programmed without any possibility of downtime, censorship, fraud, or third-party interference. As described in the Ethereum white paper by Vitalik Buterin [1]:

> *The intent of Ethereum is to create an alternative protocol for building decentralized applications, providing a different set of tradeoffs that we believe will be very useful for a large class of decentralized applications, with particular emphasis on situations where rapid development time, security for small and rarely used applications, and the ability of different applications to very efficiently interact, are important.*
>
> *(Vitalik Buterin, Ethereum White Paper, 2014)*

Ethereum, compared to other blockchains, such as Bitcoin, is a more general-purpose blockchain. For example, while Bitcoin is a blockchain to transfer value, Ethereum runs smart contracts. In addition, Ethereum is Turing-complete, which means that Ethereum is programmable.

1.1.1 Ethereum Accounts

In Ethereum, there are two types of accounts, (1) *externally owned accounts* and (2) *contract accounts*. Externally owned accounts are controlled by private keys, meaning people control them. Contract accounts are controlled by their contract code. They can only be activated by an externally owned account, meaning that a contract account is one with which no private key is associated. An account can transfer ether to another account by creating a transaction. A transaction is a message sent from one account to another. In the Ethereum white paper [1], Vitalik Buterin describes an account as follows:

> *...with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts.*
>
> *(Vitalik Buterin, Ethereum White Paper, 2014)*

An Ethereum account contains four fields:

- **Nonce**: A counter used to make sure each transaction can only be processed once.
- **Ether balance**: The number of Wei[1] owned by the account.
- **Contract code**: The code is executed when the account receives a message call. It is mainly used for smart contracts.
- **Storage**: The storage of the account.

[1] 1 Wei is the smallest denomination of Ether.

1.1.2 Messages and Transactions

In Ethereum, there are two types of messages, (1) *messages* and (2) *transactions*. A message is a message that is sent from one contract account to another contract account. A transaction is a signed data package sent from an externally owned account. In this data package, there are six fields, (1) the recipient of the transaction, (2) the signature of the transaction, (3) the amount of Ether to transfer, (4) an optional data field, (5) a `STARTGAS` value, and (6) a `GASPRICE` value. The `STARTGAS` value is the maximum number of computational steps the transaction execution is allowed to take. The `GASPRICE` value is the fee the transaction's sender is willing to pay for each computational step. These two values are explained in more detail in the next section 1.1.6 and in section 1.1.7.

The first three fields are the same as in Bitcoin. The optional data field has no function in Ethereum. The `STARTGAS` and `GASPRICE` fields are essential for preventing denial of service attacks. These fields are part of the anti-denial of the service model of Ethereum. The main goal of the anti-denial of service model is to avoid situations where code execution is infinite or takes an extremely long time, accidentally or intentionally.

With this gas and fees system, Ethereum assures that any attacker must pay for every resource he consumes. This mechanism helps to maintain the stability of the Ethereum network and prevent malicious actors from spamming the network.

1.1.3 Messages

A message is described in the Ethereum white paper [1] as follows:

> *Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment.*

A message contains :

- **sender**: The sender of the message.
- **recipient**: The recipient of the message.
- **amount**: The amount of Ether to transfer.
- **data**: The optional data field.
- **STARTGAS**: The maximum number of computational steps the transaction execution is allowed to take.

A message can be described as a transaction, but without the signature and the `GASPRICE` field, and is sent from one contract account to another . Transactions and messages are processed similarly, but they have some differences. The main difference is that a transaction is sent from an externally owned account, and a message is sent from a contract account. The other difference is that a transaction can call a function that, as a result, creates a message and call another function in the same or another contract using the opcode `CALL` and `DELEGATECALL` (see section 1.1.4). An amount of gas is used for a message sent from a transaction that triggers a function call.

1.1.4 Code Execution

Ethereum provides a virtual machine called the Ethereum Virtual Machine (EVM). The EVM is a Turing-complete virtual machine that is used to execute smart contracts, which by definition are code that is stored on the blockchain. To interact with the EVM, the code must transform into a bytecode. In the Ethereum white paper [1], Vitalik Buterin describes the code as follows:

> *The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes, where each byte represents an operation.*
>
> *(Vitalik Buterin, Ethereum White Paper, 2014)*

In code execution, there is typically an infinite loop that follows a sequence of operations. The loop continuously operates as indicated by the current program counter, which starts at zero. When the execution of an operation is finished, the program counter is increased by one, and the next operation is executed. The process continues until one of the following conditions is met, (1) the end of the code is reached, (2) an error occurs or (3) the `STOP` or `RETURN` instruction is executed. The `STOP` instruction halts the execution of the code, and the `RETURN` instruction returns the output data of the code execution. During execution, the EVM has access to three types of memory as illustrated in Figure 1.1:

- **Stack**: The stack is a last-in-first-out (LIFO) data structure and has a depth of 1024 items. It is used to store local variables.

- **Memory**: A byte array with a maximum size of $2^{256} - 1$ bytes. The memory is used to store data.

- **Storage**: A key-value store with a maximum size of $2^{256} - 1$ key-value pairs. The storage is used to store data permanently even after the end of the computation, unlike the stack and the memory.

In addition, the code execution has access to the value, the sender, and the data of the message that triggered the code execution, as well as the block header data. The code can return a byte array as output data. The output data is stored in the transaction that triggered the code execution.

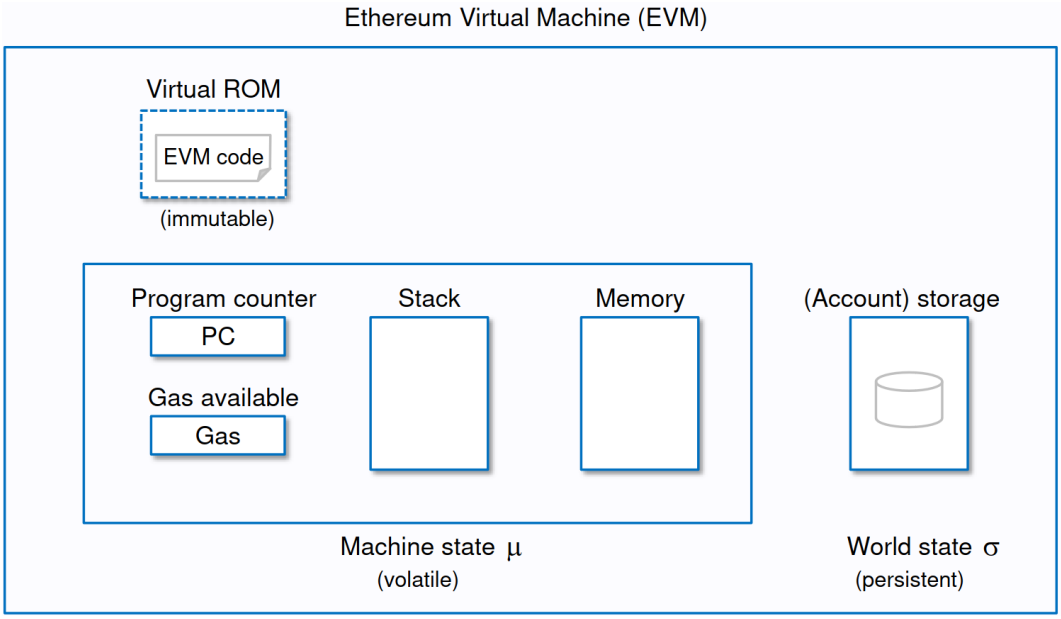

Figure 1.1: Diagram of EVM architecture as described in Ethereum Yellow Paper [2].
Source:-`https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`

1.1.5 Currency

The Ethereum white paper [1] describes the currency and issuance as follows:

> *The Ethereum network includes its own built-in currency, ether, which serves the dual purpose of providing a primary liquidity layer to allow for efficient exchange between various types of digital assets and, more importantly, of providing a mechanism for paying transaction fees.*
>
> *(Vitalik Buterin, Ethereum White Paper, 2014)*

The currency of Ethereum is called *ether* and is used to pay for transaction fees and computational services. The smallest unit of ether is called *wei*. The denomination of ether is as follows:

Denonimation of Ether		
Unit name	**Value of Wei**	**Value of Ether**
Wei (wei)	1	1×10^{-18}
*K*wei (babbage)	10^{3}	1×10^{-15}
*M*wei (lovelace)	10^{6}	1×10^{-12}
*G*wei (shannon)	10^{9}	1×10^{-5}
*T*wei (szabo)	10^{12}	1×10^{-5}
*P*wei (finney)	10^{15}	1×10^{-3}
ether (buterin)	10^{18}	1

Table 1.1: Ether denominations.

The wei denomination is designed for an internal representation of the data. The wei unit is used in most cases, and the values are displayed in either denomination or other denominations in the user interface. It means the user does not need to know the wei denomination to use the Ethereum network. Therefore, users interact with the Ethereum network using the ether denomination.

The Ethereum network has a fixed supply of ether, and the issuance of ether is the way to create new ether. This process goes through mining[2], which involves solving a computationally difficult puzzle by miners[3]. Miners contribute their computational power to the network and are rewarded with ether by successfully mining a block by solving the puzzle. Note that the issuance of ether through mining has several purposes. First, it incentivizes miners to secure the network by validating transactions and maintaining the integrity of the blockchain. Second, it ensures an adequate supply of ether, allowing actors to perform transactions and interact with decentralized applications on the Ethereum network.

One core of the principles of Ethereum is the concept of *Proof of Work* (PoW) consensus mechanism, where a fixed amount of ether is locked for each successful block mined. Nevertheless, Ethereum plans to switch to a *Proof of Stake* (PoS) consensus mechanism, known as Ethereum 2.0. With the PoS consensus mechanism, the issuance of ether will occur through staking, where actors lock their ether as collateral to participate in the consensus mechanism and receive rewards for their work. The Ethereum Foundation [3] explains the switch to PoS as follows:

> *The Ethereum network began by using a consensus mechanism that involved Proof-of-work (PoW). This allowed the nodes of the Ethereum network to agree on the state of all information recorded on the Ethereum blockchain and prevented certain kinds of economic attacks. However, Ethereum switched off proof-of-work in 2022 and started using proof-of-stake instead.*
>
> *(Ethereum Foundation)*

The issuance of new ether plays a crucial role in the functioning of the Ethereum network, providing necessary incentives for miners and ensuring availability of ether for various transactions and interactions on the platform.

1.1.6 Gas

Gas is essential to the Ethereum network. The Ethereum white paper [1] describes gas as follows:

> *In order to prevent accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. The fundamental unit of computation is "gas"; usually, a computational step costs 1 gas, but some operations cost higher amounts of gas because they are more computationally expensive, or increase the amount of data that must be stored as part of the state.*
>
> *(Vitalik Buterin, Ethereum White Paper, 2014)*

[2]Mining is the process of adding new blocks to the blockchain.

[3]Miners are the nodes that participate in the mining process. They are called miners because of the analogy with traditional mining, where miners are rewarded with gold for their work.

Gas represents the unit that measures the computational effort required to execute operations or run programs in the Ethereum network. Each transaction requires a certain amount of resources to be executed. Thus each transaction requires a fee. Gas refers to the fee required to execute a transaction or a message in Ethereum even if the transaction succeeds or fails.

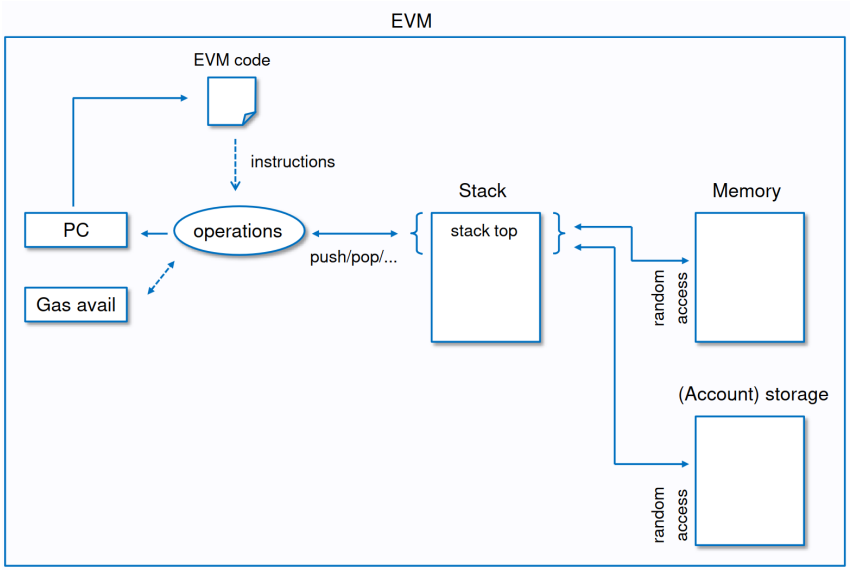

Figure 1.2: Diagram of execution model as described in Ethereum Yellow Paper [2]. **Source:**-`https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`

Figure 1.2 above shows the execution model of the Ethereum Virtual Machine, where the gas is used to pay for the execution of the code. Another value is `GASLIMIT`. This value is the maximum amount of gas used on a transaction. For example, some complicated transactions may require more gas than the default gas limit. If the gas limit is too low, the transaction will fail, and the gas will be consumed. Standard transactions have a gas limit of 21,000 gas.

1.1.7 Fees

As in Bitcoin, the fees are a crucial part of the Ethereum network because they prevent the network from being overloaded by computations. The difference between Bitcoin and Ethereum is that in Bitcoin, the fees are paid per byte of data, whereas in Ethereum, the fees are paid per computational step. As mentioned in the previous section 1.1.6, the gas is used to prevent infinite loops or other computational wastage in code. The `STARTGAS` (see section 1.1.2) value is the maximum number of computational steps the transaction execution is allowed to take. Every transaction requires a limit of gas to prevent infinite loops. Each computational step costs a certain amount of gas, typically one gas. However, some operations cost more due to their computational complexity or due to storage space.

The transaction fee is an integral part of each transaction and serves as a deterrent against spam attacks. Transactions requiring more resources will incur higher fees compared to those with fewer resources, making the fee proportional to the transaction size. The calculation of the transaction fee is demonstrated by the equation below.

$$\text{Transaction Fee} = \texttt{STARTGAS} \times \texttt{GASPRICE} \tag{1.1}$$

One advantage of decoupling the execution cost from a specific currency, such as setting the cost of a computation step to three wei, is separating transaction execution cost from the fluctuations in Ether's value compared to fiat currencies. For example, if the price of Ether significantly rises relative to a currency like the dollar, a fixed price per computational step could become unaffordable. To address this concern, the `GASPRICE` (see section 1.1.2) is used to set the price of gas in Ether, ensuring a consistent amount of gas consumed by the transaction while allowing for adjustments in the overall gas price.

1.2 Solidity, a high-level language for Ethereum

To interact with the Ethereum network, we need to write smart contracts. For this purpose, we use Solidity, a high-level language for implementing smart contracts. Solidity is a statically typed language that supports inheritance, libraries, and complex user-defined types , among other features. From the official documentation [4], Ethereum foundation describes Solidity as follows:

> *Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs that govern the behavior of accounts within the Ethereum state. Solidity is a curly-bracket language designed to target the Ethereum Virtual Machine (EVM). It is influenced by C++, Python, and JavaScript.You can find more details about which languages Solidity has been inspired by in the language influences section.*
>
> *(Ethereum Foundation)*

The Solidity compiler, known as *solc*, is responsible for compiling Solidity code into bytecode that can be executed on the Ethereum Virtual Machine (EVM). Developed in C++, the Solidity compiler is licensed under the GNU Lesser General Public License v3.0 (LGPL). It offers both command-line utility and library functionalities, providing flexibility in its usage.

In summary, all smart contracts for our project are written in Solidity and compiled into EVM-executable bytecode. The upcoming sections will delve into the Solidity language and discuss the fundamental structure necessary to understand how we implemented the smart contracts in our project.

1.2.1 State Variables

State variables in Solidity serve as a contract's persistent data stored on the blockchain, retaining their values throughout multiple function calls. These variables define the properties and attributes of a contract and are accessible to all functions within it.

To declare a state variable in Solidity, we specify its type, visibility, and optional modifiers. Solidity offers a range of data types, including integers, booleans, strings, arrays, structs, and user-defined types. These data types enable the representation and manipulation of different kinds of data within the smart contract.

In the Listing 1.1, the contract `SimpleStorage` has a state variable `storedData` of type `uint256`. By utilizing state variables, Solidity provides a powerful mechanism for managing and preserving the state of contracts on the blockchain, facilitating the development of robust and feature-rich decentralized applications.

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint256 public storedData;
    string public name = "SimpleStorage";

    function set(uint256 x) public {
        storedData = x;
    }

    function get() public view returns (uint256) {
        return storedData;
    }
}
```

Listing 1.1: Example of a contract with a state variable.

1.2.2 Functions

Functions in Solidity play a crucial role in implementing the logic of a contract and are similar to functions in other programming languages. They can be declared with different visibilities, including `external`, `public`, `internal`, or `private`. The default visibility is `public`.

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract MySimpleFunction {
    constructor() {
        // constructor
    }

    function mint(address account, uint256 amount) public onlyOwner {
        _mint(account, amount);
    }

    function decimals() public view virtual override returns (uint8) {
        return 2;
    }

}
```

Listing 1.2: Example of a contract with a function.

In the provided Listing 1.2, we show a contract named `MySimpleFunction` that implements two functions: `mint` and `decimals`. The `mint` function is declared as `public`, allowing anyone to call it. It is responsible for minting tokens by invoking the `_mint` function. On the other hand, the `decimals` function is declared as `public view`, indicating that anyone can call it and only reads data from the contract without modifying the state. In this case, the `decimals` function returns the number of decimals for the token without requiring a transaction and, therefore, is free of charge of gas.

1.2.3 Function Modifiers

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract MyToken{
    address public owner;

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can call this function.");
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

Listing 1.3: Example of a contract with a function modifier.

In the Listing 1.3, the contract `MyToken` has a function modifier `onlyOwner`. The function modifier `onlyOwner` restricts access to certain functions. In this case, only the owner of the contract can call the function `transferOwnership`. The variable `msg.sender` is a global variable that contains the address of the sender of the message. The underscore (`_`) in the function modifier indicates where the function body will be executed.

1.2.4 Events

Events are used to notify external applications about the occurrence of a specific event. This allows direct interaction with EVM logs. Events are an essential part of the Ethereum ecosystem because they allow to interact with smart contracts in a decentralized way, for example dApps (Decentralized Applications) and Oracles[4].

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;


contract MyTokenAction{
   event Transfer(address indexed from, address indexed to, uint256 value);


   function transfer(address _to, uint256 _value) public returns (bool success) {
       emit Transfer(msg.sender, _to, _value);
       return true;
   }
}
```

Listing 1.4: Example of a contract with an event.

[4]Oracles are third-party services that provide smart contracts with external information.

In Listing 1.4, the contract `MyTokenAction` shows the usage of an event called `Transfer`. The `Transfer` event serves to notify external applications about the transfer of tokens. The `transfer` function, responsible for transferring tokens from one address to another, emits the `Transfer` event after completing the token transfer. The `emit` keyword triggers the event, providing the necessary arguments.

1.2.5 Structs

Structs in Solidity serve as a mechanism for defining custom data types. They are similar to structs in C or C++ and are utilized to group variables, creating a cohesive data structure. As a result, structs are particularly valuable for organizing and storing data within smart contracts.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.8.0;
3
4  contract Orderbook {
5
6      struct Order {
7          uint amount;
8          uint price;
9          uint timestamp;
10         address trader;
11         bytes2 status;
12     }
13 }
```

Listing 1.5: Example of a contract with a struct.

In Listing 1.5, we present the contract `Orderbook`, which incorporates a struct named `Order`. This struct is responsible for storing order-related data. Within the `Order` struct, several properties are defined, including `amount`, `price`, `timestamp`, `trader`, and `status`. By utilizing the `Order` struct, the contract can effectively store and manage order data within the order book.

1.2.6 Common Data Types in Solidity

Solidity supports various data types that allow developers to define and manipulate variables within smart contracts. Here are some common data types used in Solidity:

- **Integers**: Solidity provides signed and unsigned integer types with different bit sizes. For example, `uint` represents an unsigned integer, and `int` represents a signed integer. You can specify the number of bits by appending the number, such as `uint256` or `int8`.
- **Booleans**: Solidity includes a `bool` type that can store either `true` or `false`.
- **Bytes**: Solidity provides a `bytes` type that can be used to store byte arrays. For example, `bytes32` represents a byte array of 32 bytes. The `bytes` type is similar to the `string` type, but it is not UTF-8 encoded and does not support string operations.
- **Address**: The `address` type represents a 20-byte Ethereum address. It can store Ethereum and contract addresses and provides various member functions to interact with.

- **Strings**: Solidity supports string types (`string`) for storing and manipulating variable-length text data.

- **Arrays**: Solidity allows the declaration of fixed-size and dynamic arrays. Fixed-size arrays have a predefined length, such as `uint[5]` for an array of five unsigned integers. Dynamic arrays can have a variable length, such as `uint[]` for an array of unsigned integers.

- **Mappings**: Solidity provides mappings and key-value stores that associate values with unique keys. Mappings can be used to implement data storage and retrieval efficiently. For example, `mapping(address => uint)` represents a mapping that associates unsigned integers with addresses.

- **Enums** allows us to define a set of named constants. Each constant has an associated integer value. Enums help define states or options within a contract.

- **Function Types**: Solidity supports function types, which enable us to declare variables or parameters that can store or refer to functions. Function types help implement callback mechanisms or pass functions as arguments.

These data types have been selected based on their relevance to our project and are commonly used to develop smart contracts. Using these data types in our project ensures efficient and secure data handling within the contract. It is important to note that Solidity offers a broader range of data types beyond those mentioned in this list. The official Solidity documentation [4] refers to a comprehensive understanding of all available data types and their specific use cases.

1.2.7 UML Diagram for Solidity Code

Unified Modeling Language (UML) diagrams are graphical representations that visualize the structure and relationships of various elements in software systems. While UML diagrams are commonly used for object-oriented programming languages, they can also be adapted to represent Solidity code. This project uses UML diagrams to represent the structure and relationships between our Token and exchange contracts. To create UML diagrams in LaTeX, we use the `tikz-uml` package. This package provides a set of macros for creating UML sequence diagrams, class diagrams, and other UML diagram types.

Here is an example of how to represent a UML class diagram for Solidity code using the `tikz-uml` package:

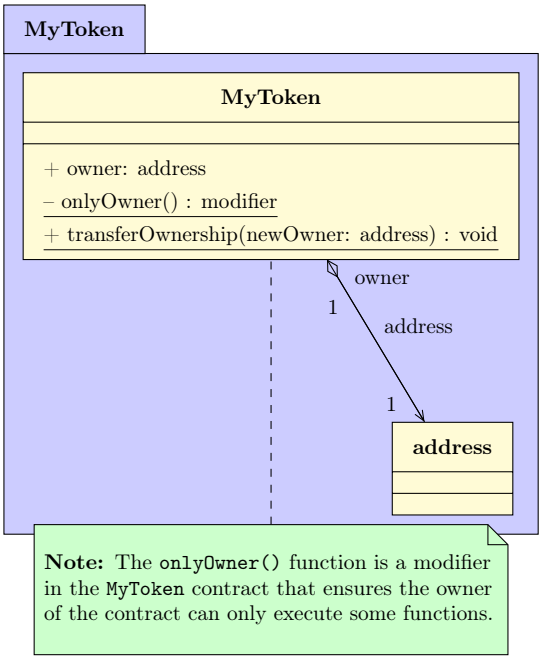

Figure 1.3: UML Class Diagram for Solidity Code

Figure 1.3 shows a UML class diagram for the Solidity code presented in the Listing 1.2. The `tikz-uml` package provides macros for creating UML classes, sequences, and other UML diagram types.

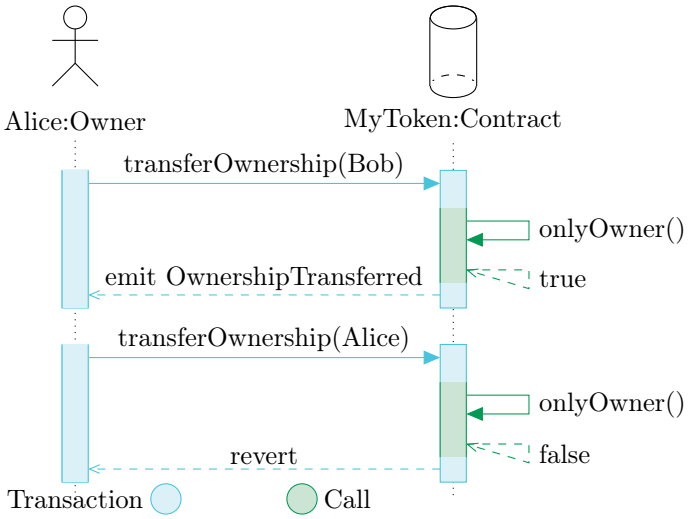

Figure 1.4: UML Sequence Diagram for presenting the function `transferOwnership`.

Figure 1.4 illustrates a UML sequence diagram corresponding to the Solidity code presented in Listing 1.2. The diagram captures the sequence of interactions between the involved actors, namely Alice and Bob, and the contract's functions. In the diagram, Alice initiates the process by calling the function `transferOwnership`, which transfers her ownership to Bob. This function internally invokes the `onlyOwner` modifier to verify whether the caller is the contract owner. If the caller is not the owner, the `onlyOwner` function reverts the transaction, preventing the transfer of

ownership. The transaction is reverted in this scenario because Alice is not the contract owner since she has already transferred her ownership to Bob. The diagram distinguishes between transactions and calls using different colors. Blue circles represent transactions, while calls are depicted with green circles. This UML sequence diagram provides a visual representation of the interactions and control flow within the contract, aiding in understanding its functionality and behavior.

1.3 Ethereum Token Standards

In blockchain technology and cryptocurrencies, a token is essential in allowing more possibilities and use cases of decentralized applications and ecosystems. A token represents a digital asset that can be compared in various forms of value, such as money, loyalty points, gold certificates, and more. To enable interoperability, consistency, and compatibility between different tokens, the Ethereum community has created a set of token standards. These standards are called Ethereum Request for Comments (ERC). The ERC standards are Ethereum Improvement Proposals (EIP) that define a set of rules that a token must implement to be ERC-compliant.

This section focuses on presenting token standards with a particular focus on the Ethereum blockchain. The Ethereum platform is known for its innovative contract capabilities and therefore is a pioneer in creating token standards through EIPs. Their proposals define the specifications and norms that a token must follow, ensuring uniformity and facilitating the integration of tokens in different applications.

In the following subsections, we describe the concept of a token, providing a general overview of the token standards. Then, we present the Ethereum Improvement Proposals created to define the token standards. Finally, we explore some of the cryptocurrency ecosystem's most popular token standards and innovations.

1.3.1 What is a Token ?

A token digitally represents an asset or utility within a decentralized network. It is often created and issued by a smart contract, a self-executing contract of rules with predefined conditions on the blockchain. We can compare a token to a physical token in the real world, where a token can represent ownership rights or access to a service. In the Ethereum Whitepaper [1], Vitalik Buterin defines a token as follows:

> *On-blockchain token systems have many applications ranging from sub-currencies representing assets such as USD or gold to company stocks, individual tokens representing smart property, secure unforgeable coupons, and even token systems with no ties to conventional value at all, used as point systems for incentivization.*
>
> *(Vitalik Buterin, Ethereum Whitepaper)*

We can illustrate the concept of a token with an example. First, let us consider an analogy for an IT Security convention like the DEF CON [5] convention. Imagine we attend the DEFCON convention, and as we enter the convention, we receive a special badge that gives us certain privileges. This badge is similar to a token because it allows us to access certain convention areas, participate in exclusive events , and receive certain benefits. However, in this case, the badge represents a value within the DEFCON convention, enabling us to engage in certain activities.

[5] DEFCON is one of the world's largest hacker conventions, held annually in Las Vegas, Nevada.

We can cite some examples of tokens such as Aragon (ANT)[6] and Basic Attention Token (BAT)[7]. Aragon provides governance rights within the Aragon Network. Holding Aragon tokens gives the holder the right to vote on certain decision-making processes and influence the direction of the project.

From a business perspective, a token can offer the company a new opportunity to issue shares, securities, or their own currency, allowing them to take control of their finances. In addition, companies can use tokens to get exclusive access to certain services or products within their ecosystem. For example, many companies use tokens to raise project funds through Initial Coin Offerings (ICO)[8]. An ICO is a fundraising mechanism in which new projects sell their underlying crypto tokens in exchange for bitcoin and ether. It is similar to an Initial Public Offering (IPO)[9], in which investors purchase company shares. Creating a cryptocurrency token for international settlements allows companies to avoid intermediaries and reduce costs and transaction time. However, regulatory compliance is critical for companies that want to issue tokens with specific characteristics.

1.3.2 Ethereum Improvement Proposals

The Ethereum Improvement Proposals (EIP) are crucial in defining standards and guidelines for token creation and implementation on the Ethereum platform. EIPs are formal documents submitted by Ethereum community members (like developers and researchers) to propose improvements , new features, and changes to the Ethereum protocol.

Regarding token standards, the Ethereum Requests for Comments (ERC) are EIPs that define a set of rules that a token must implement on the Ethereum platform to adhere to the standard. It ensures that the tokens created are compatible and interoperable among different projects and applications. ERCs are defined as a significant contribution to the Ethereum ecosystem and are widely used by the community. The official documentation for ERCs can be found on the Ethereum GitHub repository[10]. The ERC20 is by far the most popular token standard for fungible tokens[11], offering consistency and integration for developers. In addition, ERC20 can represent any digital asset, such as cryptocurrencies, loyalty points, and gold certificates.

In addition to the ERC20, other token standards are widely used in the Ethereum ecosystem. We can cite notable token standards such as the ERC721, introducing non-fungible tokens (NFTs)[12] , which are unique and indivisible digital assets. NFTs are widespread in gaming, art, and collectibles, enabling verifiable digital ownership and scarcity.

There are a lot of other token standards that are widely used in the Ethereum ecosystem. We can cite some examples, such as the ERC777 and ERC1155. Ethereum Improvement Proposals and token standards have significantly contributed to the Ethereum ecosystem's growth and development, enabling innovation, facilitating token creation, and diverse, decentralized applications on the Ethereum blockchain.

[6] Aragon Token: `https://aragon.org/`
[7] Basic Attention Token: `https://basicattentiontoken.org/`
[8] ICO: `https://en.wikipedia.org/wiki/Initial_coin_offering`
[9] IPO: `https://en.wikipedia.org/wiki/Initial_public_offering`
[10] Ethereum GitHub repository: `https://github.com/ethereum/EIPs`
[11] Fungible tokens are interchangeable and can be replaced by another token.
[12] Non-fungible tokens are unique and indivisible tokens that another token cannot replace.

1.3.3 Token standards

As explained in the previous section 1.3.2, the most approved token standard is the ERC20 which we describe more in detail in the section 1.4. Many other token standards are designed from the ERC20 for (1) suggesting improvements, (2) adding new features, (3) solving problems, or (4) proposing new functionalities. Some proposals define entirely new token standards, such as the ERC777. The table 1.2 shows some of the most popular token standards derived from the ERC20.

Token Standard	Description	Author	Date
ERC-20	The standard for fungible tokens on the Ethereum blockchain, ensuring interoperability and ease of integration.	Fabian Vogelsteller and Vitalik Buterin	November 2015
ERC-223 (Draft)	An improvement over ERC-20, introducing enhanced security features and efficiency, including the ability to reject incoming token transfers.	Dexaran	March 2017
ERC-777	A more advanced token standard that fixes some of the issues with ERC-20, including the sending mechanism for regular accounts and contracts.	Jordi Baylina, Jacques Dafflon, and Thomas Shababi	November 2017
ERC-827	An extension of ERC-20, providing additional functionalities like approving token transfers on behalf of the token holder.	Joseph Chow	January 2018

Table 1.2: List of some EIPs token standards derived from the ERC20.

Table 1.2 provides an overview of token standards, illustrating that the ERC20 token standard was the first to be established in November 2015. Subsequently, other token standards have emerged, building upon the foundation set by ERC20. It is important to note that the table represents a selection and is not exhaustive. To explore the complete documentation and comprehensive lists of token standards, one can visit the official website of Ethereum Improvement Proposals (EIPs) [5]. The EIPs website is a valuable resource for the community, providing access to drafts and proposals open for review and discussion.

1.4 ERC20 Token Standard

This section introduces the importance of selecting appropriate token standards for our settlement solution. Tokens serve as a medium of exchange and can represent any digital asset such as cryptocurrencies, enabling seamless and secure transactions. With numerous tokens at our disposal, the ERC20 token standard is the most popular and

widely adopted choice for fungible tokens because of its simplicity and interoperability. We provide an overview of the ERC20 token standard, explaining its functionalities and features.

1.4.1 Overview of ERC20

The ERC20 token standard stands for Ethereum Request for Comments 20 and has been proposed by Fabian Vogelsteller and Vitalik Buterin in 19th November 2015 [6]. It introduces a standardized interface for fungible tokens on the Ethereum blockchain, ensuring interoperability and ease of integration among different projects. ERC20 tokens are designed to be interchangeable and can be replaced by any other token of the same type and exchanged on a one-to-one basis. This particular feature called fungibility makes ERC20 tokens suitable for representing currencies, digital assets, and other financial instruments where uniformity and interchangeability are vital. This allows developers to create decentralized applications (DApps) that can interact with any ERC20 token without implementing custom code for each token. The ERC20 interface specifies a set of functions called ERC20 methods that the token contract must implement. The methods allow actors to transfer their tokens to other addresses, get the current token balance of an address, and approve the transfer of tokens from one address to another. The following code snippet shows the ERC20 interface in Solidity from the EIP-20 [6].

```
interface IERC20 {
    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function decimals() external view returns (uint8);
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
    function transferFrom(address sender, address recipient, uint256 amount) external
    ↪  returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

Listing 1.6: ERC20 interface in Solidity.

The listing 1.6 shows the ERC20 interface in Solidity representing nine functions and two events. The functions are described as follows:

- `name()` : Returns the token's name as the name suggests. E.g. the name of *Universal Export Token* is `Universal Export Token`.
- `symbol()` : Returns the token's symbol, usually a shorter name version. E.g. the symbol of *Universal Export Token* is `UET`.
- `decimals()` : Returns the number of decimals used to get its user representation. For example if `decimals` equals 2, a balance of 12345 tokens should be displayed to a user as 123.45.
- `totalSupply()` : Returns the total token supply.

- `balanceOf(address)` : Returns the token balance of a given address.
- `transfer(address, amount)` : Transfers a specified amount of tokens from the sender's address to the recipient's address. The function returns a boolean value indicating whether the operation was successful and must fire the `Transfer` event. The return value of 0 indicates that the transfer was successful.
- `transferFrom(from, to, amount)` : Transfers a specified amount of tokens from one address to another. The function returns a boolean value indicating whether the operation was successful and must fire the `Transfer` event. This method is used for a withdrawal workflow, meaning that a user can transfer tokens on behalf of another user (e.g. a decentralized exchange) with charge fees.
- `allowance(owner, spender)` : Returns the current allowance of a spender for a given owner. The allowance is the maximum amount of tokens a spender can spend on behalf of the owner.
- `approve(spender, amount)` : Allows a spender to transfer a specified amount of tokens on behalf of the token owner.
- `Transfer(address, address, amount)` : Emitted when tokens are transferred from one address to another. The event is triggered when tokens are transferred by the `transfer` and `transferFrom` functions.
- `Approval(address, address, amount)` : Emitted when the allowance of a spender for an owner is set by the `approve` function. The event is triggered when the allowance is set by the `approve` function.

All balances and amount parameters are represented on 256 bits unsigned integers, thus the maximum amount of tokens that can be handled is $2^{256} - 1$. The point is to make that some functions are declared as `view` functions like `name()`, `symbol()`, etc. It indicates that the function does not modify the state of the contract and does not consume any gas. The `view` functions are free to call and do not require gas. It can be compared to a getter function in object-oriented programming that returns a private variable's value. Another critical point is the `decimal()` function. When working with tokens, we usually work with a very small amount, representing a fraction of a token. For example, if one owns 4 UET tokens, transferring 3.49 UET to another address may be necessary, and keeping 0.51 UET in the sender's address. The EVM does not support floating point numbers, so Solidity allows us to define the integer number values of the token that should be displayed as a fraction of a token. It can be a problem because if one wants to transfer 3.49 UET, the number of tokens to transfer can be only 3 UET or 4 UET.

To solve this problem, ERC20 tokens include the `decimals()` function that returns the number of decimals used to get its user representation. For example, if `decimals()` equals 2, it is possible to transfer 3.49 UET by transferring 349 UET and keeping 51 UET in the sender's address. It is important to note that the `decimals()` function is only used for display purposes and does not affect the arithmetic of the token. A decimal value of 2 is usually common in many other ERC20 tokens. It means when we mint or transfer tokens, we work with a value of :

$$\text{value} = \text{UET} \times 10^{\text{decimals}} \tag{1.2}$$

For example, if we want to mint 1000 UET, we mint $1000 \times 10^2 = 100000$ UET, the same process for transferring tokens. However, some tokens like Ether have a decimal value of 18, which means that when we transfer 1 Ether, we transfer $1 \times 10^{18} = 1000000000000000000$ Wei. Some others have a decimal value of 0, which means that when we transfer 1 token, we transfer $1 \times 10^0 = 1$ token. It can make sense for some tokens like a token representing a share of a company. It is not possible to transfer a fraction of a share.

1.4.2 Methods of the ERC20 interface

The `allowance` method enables an owner, such as Alice, to verify the number of tokens that a spender, such as Bob, is permitted to transfer on their behalf. This method takes two parameters: the owner's address and the spender's address. It returns the number of tokens the spender is authorized to spend on behalf of the owner. Listing 1.7 exemplifies the implementation of the `allowance` function in the OpenZeppelin library. Notably, this function is classified as a `view` function, indicating that it does not modify the state of the contract and does not consume any gas during execution. The usage of the `virtual` keyword signifies that a child contract can override the function, offering flexibility for customization. Additionally, the `override` keyword indicates that the function overrides a corresponding function in the parent contract, ensuring the appropriate behavior and consistency within the contract hierarchy.

```
1 /**
2 * @dev See {IERC20-allowance}.
3 */
4 function allowance(address owner, address spender) public view virtual override returns
  ↪  (uint256) {
5     return _allowances[owner][spender];
6 }
```

Listing 1.7: OpenZeppelin implementation of the `allowance` function.

The `approve` method enables an owner, such as Alice, to grant permission to a spender, such as Bob, to transfer a specific quantity of tokens on their behalf. By invoking this method, the owner sets the allowance for Bob to the designated value. Listing 1.8 shows the implementation of the `approve` function in the OpenZeppelin library. This function provides a standardized and secure approach for granting token transfer permissions.

The implementation shown in Listing 1.8 includes two conditions on lines 15 and 16 that must be satisfied to execute the approval process. These checks ensure that both the owner and the spender addresses are not the zero addresses. The zero address, denoted as `address(0)`, holds special significance in Ethereum. It represents the absence of an address and is utilized in various scenarios. By casting the integer 0 to a 20-byte address, the zero address is used for initializing address variables and verifying whether an address has been set. It is also called the `0x0` or `0x0` account. Furthermore, the zero address plays a role in burning tokens, where tokens sent to the zero address are permanently lost. It is employed for checking invalid addresses

```
1  /**
2  * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
3  *
4  * This internal function is equivalent to `approve`, and can be used to
5  * e.g. set automatic allowances for certain subsystems, etc.
6  *
7  * Emits an {Approval} event.
8  *
9  * Requirements:
10 *
11 * - `owner` cannot be the zero address.
12 * - `spender` cannot be the zero address.
13 */
14 function _approve(address owner, address spender, uint256 amount) internal virtual {
15     require(owner != address(0), "ERC20: approve from the zero address");
16     require(spender != address(0), "ERC20: approve to the zero address");
17
18     _allowances[owner][spender] = amount;
19     emit Approval(owner, spender, amount);
20 }
```

Listing 1.8: OpenZeppelin implementation of the `approve` function.

as well. For instance, to validate an address, one can compare it to the zero address. When both conditions are fulfilled, the method updates the spender's allowance to the provided value and emits an `Approval` event.

The `transfer` method enables an owner, such as Alice, to initiate a transfer of a specified token amount to another address, such as Bob's address. This operation involves deducting the token quantity from Alice's balance and adding it to Bob's balance. Listing 1.9 presents the OpenZeppelin implementation of the `transfer` function. The function accepts two parameters: the recipient's address and the number of tokens to be transferred. It first verifies the validity of both the sender and recipient addresses. Subsequently, the method checks if the sender possesses a sufficient token balance for the transfer. If the sender holds an adequate amount of tokens, the function updates the balances of the sender and recipient accordingly and emits a `Transfer` event.

Finally, the `transferFrom` method facilitates the transfer of a specified token amount by a designated spender, such as Bob, on behalf of the token owner, Alice. Prior approval from Alice is required, which is obtained through the `approve` method. This functionality allows the spender to transfer tokens between two addresses, providing a convenient mechanism for delegated token transfers.

Listing 1.10 shows the OpenZeppelin implementation of the `transferFrom` function. This method accepts three parameters, the sender's address, the recipient's address, and the number of tokens to be transferred. It begins by verifying that the requested token transfer amount, specified in line 19 as `_spendAllowance(from, spender, amount)`, does not exceed the sender's approved allowance. If the allowance is insufficient, the function reverts, preventing the transfer. Additionally, if the allowance is set to 0, indicating that the spender has not been granted permission to transfer tokens on behalf of the sender, the method also reverts. By default, the allowance is 0 for all addresses, ensuring that no unauthorized transfers can occur. Finally, the function invokes the `_transfer` function (refer to Listing 1.9) to carry out the actual token transfer from the sender to the recipient.

```solidity
/**
* @dev Moves `amount` of tokens from `from` to `to`.
*
* This internal function is equivalent to {transfer}, and can be used to
* e.g. implement automatic token fees, slashing mechanisms, etc.
*
* Emits a {Transfer} event.
*
* Requirements:
*
* - `from` cannot be the zero address.
* - `to` cannot be the zero address.
* - `from` must have a balance of at least `amount`.
*/
function _transfer(address from, address to, uint256 amount) internal virtual {
    require(from != address(0), "ERC20: transfer from the zero address");
    require(to != address(0), "ERC20: transfer to the zero address");

    _beforeTokenTransfer(from, to, amount);

    uint256 fromBalance = _balances[from];
    require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
    unchecked {
        _balances[from] = fromBalance - amount;
        // Overflow not possible: the sum of all balances is capped by totalSupply, and
        ↪  the sum is preserved by
        // decrementing then incrementing.
        _balances[to] += amount;
    }

    emit Transfer(from, to, amount);

    _afterTokenTransfer(from, to, amount);
}
```

Listing 1.9: OpenZeppelin implementation of the `transfer` function.

The `transferFrom` function is essential for providing additional flexibility and control over token transfers within the ERC-20 standard. Its purpose is to enable token owners to delegate the authority of transferring tokens to another address while retaining certain levels of control. This delegation can be revoked at any time by the token owner. A practical use case for the `transferFrom` function is in decentralized exchanges (DEXs), where token owners may delegate the authority to the DEX to perform token transfers on their behalf during trading. For example, Alice can delegate the DEX to transfer her tokens to a buyer in exchange for another token. By delegating the authority, Alice can conduct transactions even when not online, as the DEX can initiate the transfers on her behalf. The DEX, in turn, can perform the transfers without concern for explicit approval from Alice as long as the delegated allowance is available. It is important to note that the delegation can be revoked using the `approve` method, ensuring the owner's control and security. The specifications of the ERC-20 standard, authored by Fabian Vogelsteller [6], provide further details and guidelines for implementing the `transferFrom` method.

> *...The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.*
>
> *Fabian Vogelsteller, author of the ERC-20 standard*

```
/**
 * @dev See {IERC20-transferFrom}.
 *
 * Emits an {Approval} event indicating the updated allowance. This is not
 * required by the EIP. See the note at the beginning of {ERC20}.
 *
 * NOTE: Does not update the allowance if the current allowance
 * is the maximum `uint256`.
 *
 * Requirements:
 *
 * - `from` and `to` cannot be the zero address.
 * - `from` must have a balance of at least `amount`.
 * - the caller must have allowance for ``from``'s tokens of at least
 * `amount`.
 */
function transferFrom(address from, address to, uint256 amount) public virtual override
↪  returns (bool) {
    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}
```

Listing 1.10: OpenZeppelin implementation of the `transferFrom` function.

The functionality of the ERC-20 token standard, including the `approve`, `transfer`, `allowance`, and `transferFrom` methods, is illustrated in Figure 1.5. The sequence diagram illustrates the interactions between Alice, Bob, and the ERC-20 contract *MyToken*. It illustrates the following steps: (1) Alice delegates the authority to Bob to transfer tokens on her behalf by approving an allowance of 100 tokens using the `approve` method. (2) Alice transfers 100 tokens to Bob using the `transfer` method. (3) Bob checks the allowance granted by Alice using the `allowance` method. (4) Bob transfers 50 tokens to Charlie on behalf of Alice using the `transferFrom` method. The sequence diagram visually represents the message flow and function calls between the participants, demonstrating the process of checking Alice's allowance, transferring tokens from Alice to Bob, and transferring tokens from Bob to Charlie on behalf of Alice. Note that certain verifications, such as checking the sender's and recipient's balances or the validity of the sender's address, are omitted for simplicity.

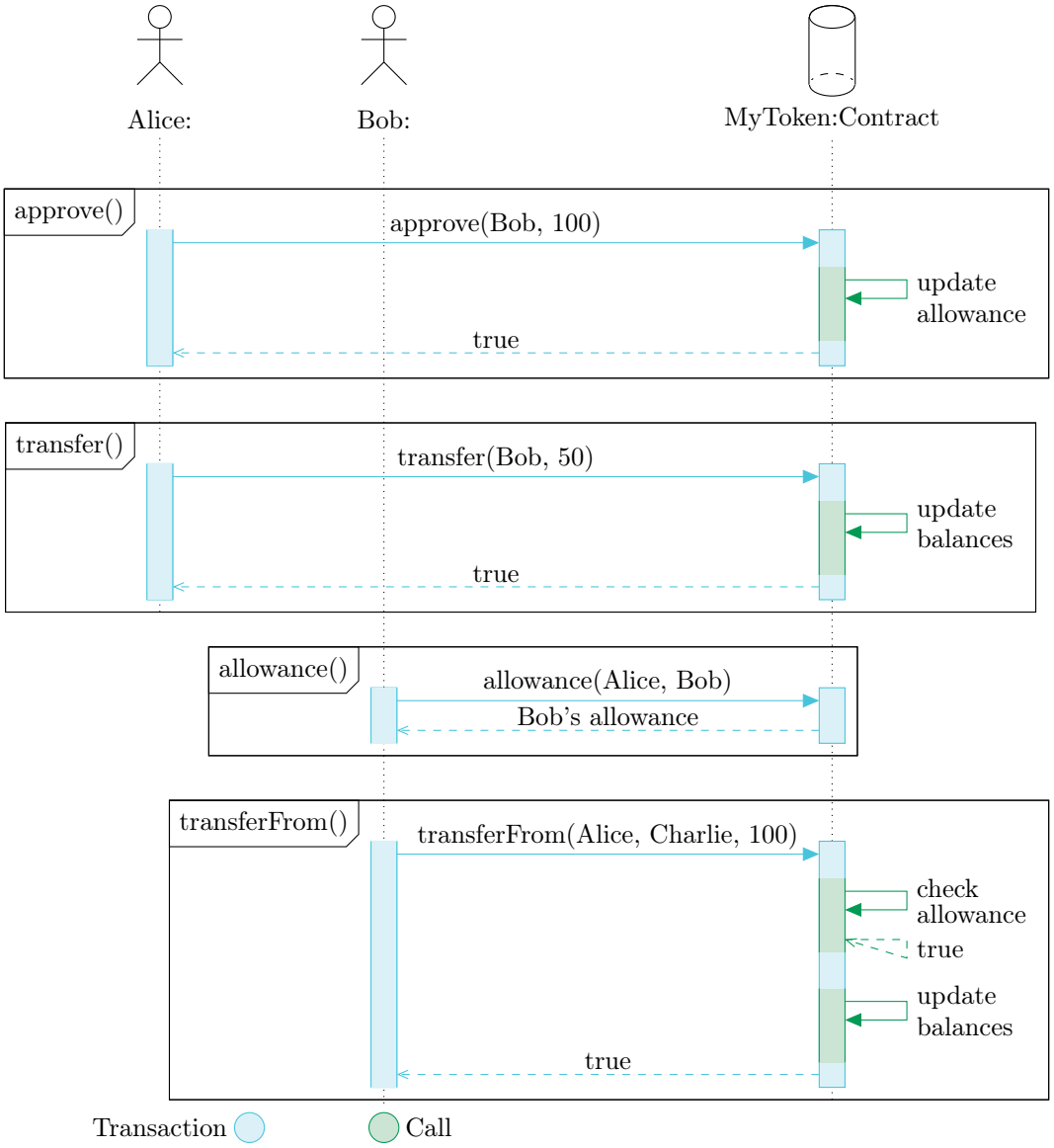

Figure 1.5: UML sequence diagram of the ERC-20 methods

1.5 OpenZeppelin, a secure library for smart contracts

As demonstrated in the previous section 1.4.2, some implementations of methods are illustrated with the codes coming from OpenZeppelin github repository [7]. The OpenZeppelin library is widely used and trusted for building secure smart contracts on the Ethereum blockchain. It offers a range of reusable and audited smart contract components, including implementing several standards tokens such as ERC-20. In the EIP-20 [6], Fabian Vogelsteller, the author of the ERC-20 standard, recommends to use the OpenZeppelin implementation of the ERC-20 standard [8]:

> *There are already plenty of ERC20-compliant tokens deployed on the Ethereum network. Different implementations have been written by various teams that have different trade-offs: from gas saving to improved security.*

Fabian Vogelsteller, author of the ERC-20 standard

By leveraging the work with OpenZeppelin, our ERC-20 token implementation can benefit from the following advantages:

- **Enhanced Security:** OpenZeppelin is widely recognized for its focus on security best practices. The library does regular audits and updates to address security issues, significantly reducing the risk of vulnerabilities in our smart contracts.
- **Community Trust:** OpenZeppelin's smart contract components have been extensively deployed and utilized by the Ethereum community in numerous projects. By utilizing battle-tested code, we can tap into the collective knowledge and experience of the community, ensuring the quality and reliability of our ERC-20 token implementation.
- **Compliance with Standards:** OpenZeppelin provides compliant implementations of popular token standards, including ERC-20 and ERC-721. This ensures that our smart contracts seamlessly integrate with the broader Ethereum ecosystem, enabling interoperability with other decentralized applications (dApps) and wallets.
- **Code Reusability:** OpenZeppelin's modular and reusable smart contract components enable us to save time and effort. By leveraging the audited codebase, we can focus on implementing the specific business logic and features required for our ERC-20 token rather than starting from scratch.

For instance, one notable advantage of utilizing OpenZeppelin is the mitigation of common security pitfalls. The library incorporates best practices for secure smart contract development, such as protection against integer overflow/underflow and reentrancy attacks [9]. By relying on OpenZeppelin, we can avoid the potential introduction of security vulnerabilities into our smart contracts without the need for extensive security audits.

The OpenZeppelin library does not only provide a secure implementation for building smart contracts, but it also offers additional functionalities that can enhance our ERC20 token implementation. One notable example is the `Ownable` contract, which provides basic access control functionalities. It allows us to restrict access to certain functions to the owner of the smart contract. This is useful for implementing administrative functions such as minting new tokens or pausing the token contract. The `transferOwnership` method and `renounceOwnership` method are included in the `Ownable` contract. It also includes the methods like `mint` and `burn` that are not part of the ERC-20 standard but are commonly used in ERC-20 token implementations. The `mint` method allows the smart contract owner to mint new tokens, while the `burn` method allows the owner to burn tokens. These methods can be crucial for different use cases such as token distribution, token supply management, token burning to remove tokens from circulation permanently, etc. We gain access to these additional methods by inheriting from the `ERC20` and `ERC20Detailed` contracts. The `ERC20Detailed` contract provides the `name`, `symbol`, and `decimals` methods, which are used to provide information about the token. Another example is the

`ERC20Capped` contract, which provides a way to cap the maximum supply of the token. This can be useful for implementing a fixed supply token or for limiting the token's total supply to a certain amount.

1.6 Truffle, a development environment for Ethereum

Truffle [10] is a popular development environment and testing framework for Ethereum based on Node.js. It provides tools that simplify developing, testing, and deploying smart contracts on the Ethereum network. Truffle offers a comprehensive development environment with built-in smart contract compilation, linking, migration, testing, and deployment. It also provides a configurable build pipeline that supports custom build processes.

We can cite the following advantages of using Truffle:

1. **Simple and Easy to Use:** Truffle simplifies the development workflow by providing a suite of command-line tools and a project structure that organizes our smart contract code, tests, and deployment configurations. The development lifecycle is, therefore, easier to manage and maintain.

2. **Automated Smart Contract Compilation and Deployment:** Truffle automatically compiles our smart contracts and deploys them to the Ethereum network. It provides a configuration file that allows us to define the network to deploy to, the smart contracts to deploy and manage the contract migration across various stages of development or network.

3. **Automated Smart Contract Testing:** Truffle provides a built-in testing framework that allows us an easier way to create and execute test cases for our smart contracts. The tests are written in JavaScript and can be run against any Ethereum network. It ensures that our smart contracts behave as expected before we deploy them to the mainnet or helps us to identify bugs and vulnerabilities in our smart contracts.

4. **Network Management:** Truffle provides a convenient way to manage the Ethereum networks when we want to deploy our smart contracts. By supporting multiple networks, we can easily deploy our smart contracts to different networks such as the mainnet[13], testnet (such as Ropsten, Kovan, Rinkeby), or a local development network (such as Ganache[14]). This flexibility allows us to test our smart contracts in different environments and ensure they work as expected before deploying them to the mainnet.

5. **Support of External Tools:** Truffle integrates with external tools such as Ganache, Infura[15], and Metamask[16]. This allows us to easily connect to the Ethereum network and deploy our smart contracts, enhancing our development experience.

[13]The mainnet is the main Ethereum network where real transactions take place.

[14]Ganache is a personal blockchain for Ethereum development used to deploy contracts, develop applications, and run tests. `https://www.trufflesuite.com/ganache`

[15]Infura is a service that provides access to the Ethereum network. `https://infura.io/`

[16]Metamask is a browser extension that allows us to interact with the Ethereum network.

6. **Community and Documentation:** Truffle has a large community of developers and users, so we can easily find online help, support, resources, and tutorials. It also has comprehensive documentation that provides detailed information about the different features of the framework, with examples and code snippets.

Therefore, by using Truffle, we can significantly reduce the time and effort required to develop, test and deploy, enhancing our smart contract development experience.

1.7 Web3, a library for interacting with the Ethereum network

After the development of the smart contract with Truffle, we need to interact from a front-end perspective with the Ethereum network where our smart contract is deployed. As a result, we need a solution to interact with the Ethereum network from our front-end application, which is developed for web browsers. That is where the solution of web3 [11] comes in. The web3 concept emerged as a way to interact with the next generation of the internet that aims to decentralize the web from the current client-server model[17]. It emphasizes a web where users control their data, digital assets, and online interactions.

The term *Web3* gained popularity in 2014, with the advent of the Ethereum blockchain, which introduced the concept of smart contracts and decentralized applications (dApps)[18]. The Ethereum foundation [11] defines web3 as follows:

> *Web3 has become a catch-all term for the vision of a new, better internet. At its core, Web3 uses blockchains, cryptocurrencies, and NFTs to give power back to the users in the form of ownership.*
>
> *(Ethereum Foundation)*

As Ethereum gained popularity, the development of Web3 libraries and tools to interact with the Ethereum network also increased for web applications. This led to the development of the Web3.js library [12] designed explicitly for Ethereum. Web3.js is an essential component for interacting with the Ethereum network using JavaScript. We can compare it as a bridge between our front-end application and the Ethereum network that makes the office of the back-end from the decentralized world of Ethereum. With Web3.js, we can connect to the Ethereum nodes, send transactions, call smart contracts, and retrieve data, all within our front-end application using JavaScript.

It enables us to build dApps that can leverage all the advantages of the Ethereum blockchain. For example, if we integrate Web3.js into our front-end application, we can allow a company to manage its supply chain using a dApp with direct access to the Ethereum network. It enables the company to interact with smart contracts, which means that they can manage their digital assets, such as tokens, and participate in the decentralized finance (DeFi)[19] applications.

[17] The client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.

[18] A decentralized application (dApp) is an application that many users run on a decentralized network with trustless protocols.

[19] Decentralized finance (DeFi) is a blockchain-based form of finance that does not rely on central financial intermediaries such as brokerages, exchanges, or banks to offer traditional financial instruments and instead utilizes smart contracts on blockchains, the most common being Ethereum.

Therefore, utilizing Web3.js in our front-end application unlocks the possibilities for our platform, including peer-to-peer payments, decentralized governance, and secure data management. For building our decentralized financial platform for international settlements, web3.js is the crucial component that allows us to seamlessly integrate Ethereum functionalities into our front-end application and provide users with a better experience.

List of Figures

List of Tables

List of sources

A | An appendix

References

[1] Vitalik Buterin. *Ethereum Whitepaper*. 2014. URL: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf (visited on 05/23/2023).

[2] Dr. Gavin Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf (visited on 05/23/2023).

[3] Ethereum Foundation. *Proof of Stake*. 2023. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/ (visited on 05/23/2023).

[4] Ethereum Foundation. *Solidity Documentation*. 2019. URL: https://docs.soliditylang.org/en/v0.8.20/ (visited on 05/23/2023).

[5] Ethereum Foundation. *Ethereum Improvement Proposals*. 2015. URL: https://eips.ethereum.org/ (visited on 05/23/2023).

[6] Fabian Vogelsteller and Vitalik Buterin. *ERC-20 Token Standard*. Nov. 2015. URL: https://eips.ethereum.org/EIPS/eip-20 (visited on 05/23/2023).

[7] OpenZeppelin. *Github Repository of OpenZeppelin Contracts*. Version 4.x. 2023. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts (visited on 05/23/2023).

[8] OpenZeppelin. *ERC20*. Version 4.x. 2023. URL: https://docs.openzeppelin.com/contracts/4.x/erc20 (visited on 05/23/2023).

[9] Ethereum Foundation. *Reentrancy attack*. 2023. URL: https://docs.soliditylang.org/en/v0.8.20/security-considerations.html#re-entrancy (visited on 05/23/2023).

[10] Truffle Suite. *Truffle Suite*. 2023. URL: https://www.trufflesuite.com/ (visited on 05/23/2023).

[11] Ethereum Foundation. *Web3*. 2014. URL: https://ethereum.org/en/web3/ (visited on 05/23/2023).

[12] Ethereum Foundation. *Web3.js*. 2016. URL: https://web3js.readthedocs.io/en/v1.5.2/ (visited on 05/23/2023).