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

Lausanne, HES-SO//Master, June 2, 2023

Abstract

Today, blockchain has become an ever-growing instrument for designing and developing secure exchange systems between multiple parties, especially where traditional payment systems reach limitations. As the growth of multilateral trade and globalization increases, the need for a security tool to settle various currencies is becoming more critical. Blockchain-based exchange address one of these limitations, where this solution allows us to create a virtual currency (cryptocurrency) to settle transactions between pairs of national currencies. This project aims to demonstrate the current settlement process between national currencies and propose a suited blockchain-based solution by creating a virtual currency (cryptocurrency). For the project and to have a proof-of-concept, a simulation of trade between the US dollar and the created virtual currency will be implemented. The goal of this project is to create an automated and secure blockchain-based inter-currency settlement platform, which will be able to exchange our new cryptocurrency with other currencies.

Key words: Cryptography, Blockchain, Ethereum, Smart Contracts, ERC20, Solidity, Web3

Contents

1 Introduction

This section aims to provide an overview of the context of the project, the state of the art, and the project's objectives. It introduces the problem of current payment systems and the potential of blockchain technology to address some of the limitations of these systems. It also presents the objectives of the project and the methodology that will be used to achieve them.

1.1 Context

The exchange of goods and services has been a part of the beginning of human society throughout our history. It results in the creation of various payment methods to facilitate transactions. From the barter system in the early days to the introduction of currency in the roman empire, humans have always found ways to make transactions with each other in fair trade. Nowadays, the most common payment methods are through bank systems based on fiat currencies. However, these traditional payment systems suffer high transaction costs, lengthy settlement times, and, most important, transparency.

High transaction costs associated with traditional systems, such as fees charged by financial institutions for international transactions, impose a financial burden on businesses and individuals. Moreover, the lengthy settlement times for transactions, especially international ones, create delays and hinder the smooth flow of funds in a globalized world.

The challenge is more transparency in traditional payment systems, impacting the trust between parties. Indeed, multiple intermediaries or third parties can be involved in financial transactions, rendering the transactions unclear. Moreover, this opacity opens the door to fraud, errors, and illegal activities, posing risks to businesses and individuals.

While alternative payment systems like the Hawala system have emerged in certain regions, they also have limitations. Based on trust and personal relationships, the Hawala system offers low-cost and faster transactions, particularly in areas lacking access to formal banking services. However, it lacks scalability and regulatory systems, raising concerns about transparency and potential risks associated with fraud and illegal activities.

Overall, traditional payment systems have cost, settlement times, and transparency limitations. Cryptocurrencies and blockchain technology offer an alternative solution for addressing these challenges.

1.2 State of the art

This chapter provides an overview of blockchain technology's state-of-the-art and critical features. It also addresses the challenges associated with international settlements, as discussed in Section 1.1, and explores how blockchain technology, specifically the Ethereum blockchain, can offer potential solutions. The focus is on developing a new cryptocurrency token shaped for international settlements, taking advantage of the capabilities and features provided by the Ethereum blockchain.

Blockchain technology and its features

Blockchain technology has become essential for designing and developing secure exchange systems between multiple parties. As the growth of multilateral trade and globalization increases, the need for a secure and transparent payment system is becoming more critical to settling currencies. Blockchain-based payment systems can provide a solution where virtual currencies (cryptocurrencies) can settle transactions between pairs of national currencies. The first cryptocurrency was Bitcoin, created in 2008 [1] by an unknown person or group under the pseudonym Satoshi Nakamoto. We can name Bitcoin the first decentralized cryptocurrency that introduced the concept of a decentralized and distributed system for financial transactions. After Bitcoin, other cryptocurrencies have been created since then, such as Ethereum, Monero, Litecoin, Etc.

The blockchain acts like a digital ledger that enables the secure and transparent transfer of digital assets between two parties without needing a trusted intermediary. The immutability of the blockchain is achieved through cryptography and consensus algorithms, providing high security and transparency. References such as *Blockchain Basics* by Daniel Dresched [2] and *Mastering Bitcoin* by Andreas Antonopoulos [3] provide an excellent introduction to blockchain technology.

Current international settlements and their challenges

Traditional international settlements must be more robust to limitations such as high transaction costs, lengthy settlement times, and lack of transparency. These limitations are due to the reliance on trusted intermediaries and third parties to process and verify transactions. In his book *Global Financial Systems* [4], Jon Danielsson offers valuable insights into the current challenges of international settlements.

Blockchain technology versus these challenges

Blockchain technology can offer some promising solutions to the current challenges of international settlements. Due to its decentralized and distributed nature, it can provide a low-cost, fast, and secure payment system. Settlements can be executed directly between two parties reducing the need for intermediaries and third parties and thus reducing the transaction costs. The immutability and transparency of the blockchain can also provide real-time visibility of transactions, reducing the risk of fraud and illegal activities. In the book of *Blockchain Revolution* by Don Tapscott and Alex Tapscott [5], the authors explore the potential of blockchain technology to transform the financial sector and the economy.

Ethereum and smart contracts

The Ethereum blockchain is the second most popular blockchain after Bitcoin among developers, with its new features allowing to develop smart contracts and the concept of Decentralized Applications (dApps). A smart contract is a digital one stored on the blockchain and executed automatically when certain conditions are met, ensuring automated and transparent settlement transactions. The book of *Mastering Ethereum* by Andreas Antonopoulos and Gavin Wood [6] provides in-depth insights into the potential of Ethereum blockchain and Smart Contracts development.

1.3 Objectives of this project

Our objective aims to develop a new cryptocurrency token named "Universal Export Token" (UET) for international settlements using the Ethereum blockchain. The token must be used to settle transactions between two parties. For this project, to have a real case, we will simulate the case of a company that wants to use its own cryptocurrency token to settle transactions with suppliers, for example. To project the scenario further, we will consider the case of a company that can exchange its cryptocurrency token with other cryptocurrencies or national currencies (e.g. USD). For this purpose, we must develop a secure Blockchain-based inter-currencies settlement platform.

2 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 decide on 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 global settlement systems. As part of this analysis, we aim to digitally create our own cryptocurrency token 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 2.1, 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 2.2, 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 2.3, 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 2.4, 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 2.5, 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 2.6 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 2.7 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 library allows us to interact with our cryptocurrency token and the Ethereum network.

2.1 Ethereum, a Decentralized Application Platform

The Ethereum blockchain is a decentralized platform allowing us to develop and deploy decentralized applications (DApps) using smart contracts, which is revolutionary from Bitcoin's blockchain. Smart contracts, as their name suggests, are executed automatically under certain conditions, preventing fraud or third-party interference. As described in the Ethereum white paper by Vitalik Buterin [7]:

> *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.

2.1.1 Ethereum Accounts

In Ethereum, there are two types of accounts, (1) *externally owned accounts* and (2) *contract accounts*. People can control externally owned accounts with their private keys protected with cryptography. On the other hand, contract accounts are held by their contract code. Therefore, 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 message sent from one account to another. In the Ethereum white paper [7], 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.

2.1.2 Messages and Transactions

For communicating between nodes, Ethereum uses two type of data structures, (1) *messages* and (2) *transactions*. A message is data that is sent from one contract account to another contract account. A transaction is a signed data package sent from an externally owned account, which contains 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 `GASPRICE` value represent a maximum amount of computational steps the transaction execution is allowed to execute. The `GASPRICE` value is the fee the transaction's sender is willing to pay for each computational step. We explains with more details in the next section 2.1.6 and in section 2.1.7, the `STARTGAS` and `GASPRICE` fields.

Bitcoin and Ethereum have a similar transaction structure, because they both have the first three fields in common. The optional data field has no function in Ethereum. The `STARTGAS` and `GASPRICE` fields are essential for preventing denial of service attacks. 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 is the core of the Ethereum, allowing Ethereum to maintain the stability of its network and prevent malicious actors from spamming it.

2.1.3 Messages

A message is described in the Ethereum white paper [7] as follows:

> *Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment.*
>
> *(Vitalik Buterin, Ethereum White Paper, 2014)*

A message contains :

- **sender**: Sender of the message.
- **recipient**: Recipient of the message.
- **amount**: Amount of Ether to transfer.
- **data**: Optional data field.
- **STARTGAS**: Maximum number of computational steps the code execution is allowed to execute.

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 2.1.4). An amount of gas is used for a message sent from a transaction that triggers a function call.

2.1.4 Code Execution

Smart contracts are described as a piece of code stored in the Ethereum blockchain world. To execute this code, we use a virtual machine called the Ethereum Virtual machine (EVM) that works as a Turing-complete virtual machine. The EVM can only understand bytecode. Therefore, we need to translate our code into bytecode. In the Ethereum white paper [7], 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, each representing 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. At the end of the execution of a function, the program counter is increased by one, and the next operation is executed. Then, the process continues until one of the following conditions is met, (1) at the end of the code is reached, (2) when an error occurs or (3) at the `STOP` or at `RETURN` instruction execution. 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 2.1:

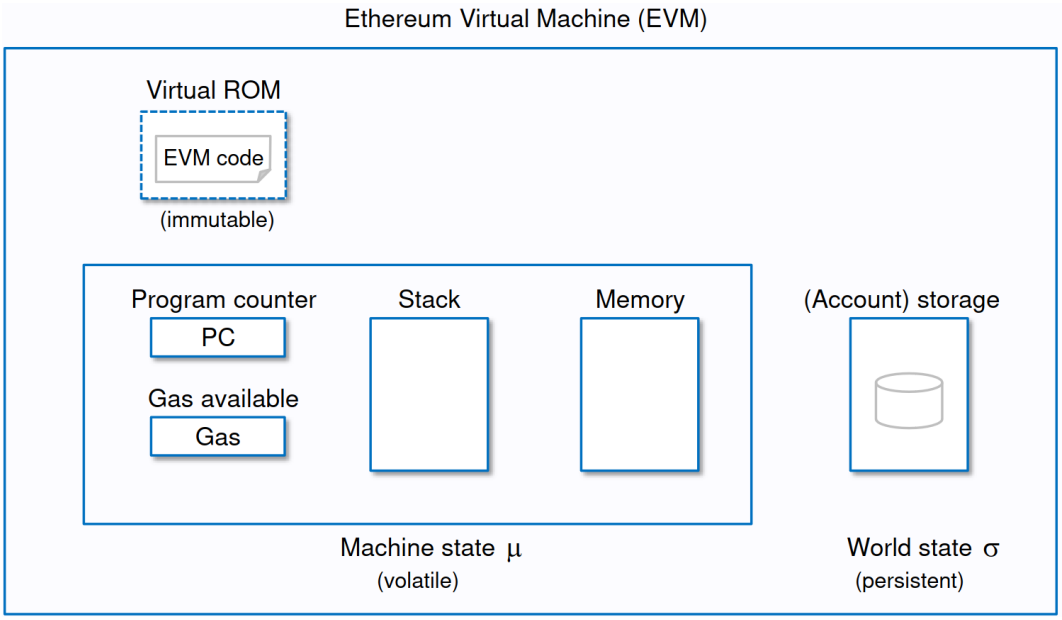

Figure 2.1: Diagram of EVM architecture as described in Ethereum Yellow Paper [8].
Source:-`https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`

Here is a description of the three types of memory illustrated in Figure 2.1:

- **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 representing the gas as a fuel with the 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.

2.1.5 Currency

The Ethereum white paper [7] 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 in Table 2.1:

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 2.1: 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

[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.

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 [9] 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.

2.1.6 Gas

Gas is essential to the Ethereum network. The Ethereum white paper [7] 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)*

We can represent the gas as a fuel of the Ethereum network, measuring the amount of work required to perform a specific operation. Each transaction requires a certain amount of resources to be executed. Thus each transaction requires a *fee* to execute a transaction or a message in Ethereum even if the transaction succeeds or fails.

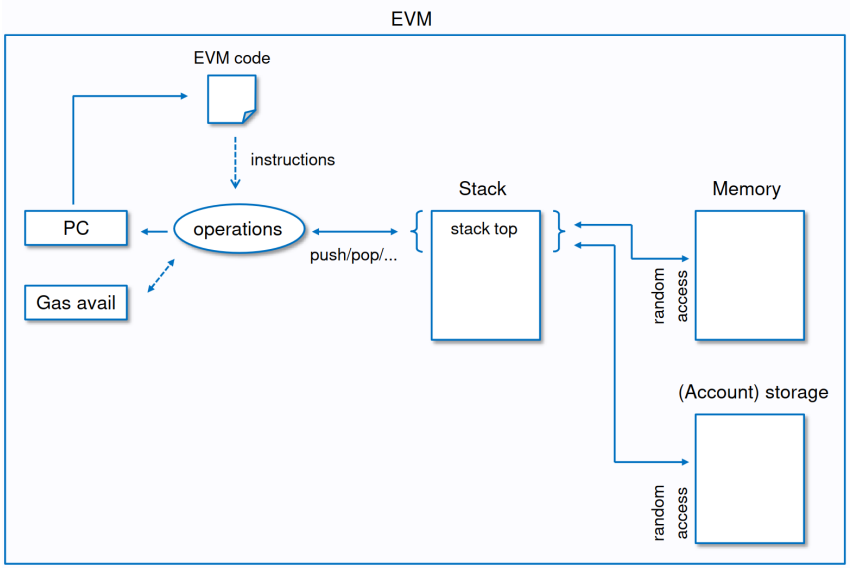

Figure 2.2: Diagram of execution model as described in Ethereum Yellow Paper [8]. **Source:**-`https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`

Figure 2.2 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.

2.1.7 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 2.1.6, the gas is used to prevent infinite loops or other computational wastage in code.The `STARTGAS` (see section 2.1.2) value represents the maximum amount of gas, meaning we cannot exceed a certain number of computational steps. Every transaction requires a limit of gas to prevent infinite loops, and 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 fee is crucial for each transaction and serves as a mitigation against spam attacks. Transactions requiring more resources will incur higher fees than those with fewer resources, making the fee proportional to the transaction size. The calculation of the transaction fee is demonstrated by the equation below.

$$\text{Transaction Fee} = \texttt{STARTGAS} \times \texttt{GASPRICE} \tag{2.1}$$

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 a dollar, a fixed

price per computational step could become unaffordable. To address this concern, the `GASPRICE` (see section 2.1.2) is used to set the gas price in Ether, ensuring a consistent amount of gas consumed by the transaction while allowing for adjustments in the overall gas price.

2.2 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 [10], 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 that targets the Ethereum Virtual Machine (EVM). C++, Python, and JavaScript influence it. In the language influences section, you can find more details about which languages Solidity has been inspired by.*
>
> *(Ethereum Foundation)*

Solidity owns its compiler, the compiler *solc*, whose purpose is to translate the Solidity code into bytecode, the only language understood by 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 smart contracts in our project.

2.2.1 State Variables

State variables in Solidity are 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.

```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 2.1: Example of a contract with a state variable.

In the Listing 2.1, 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 blockchain contracts, facilitating decentralized applications' development.

2.2.2 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 2.2: Example of a contract with a function.

In the provided Listing 2.2, 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.

2.2.3 Function Modifiers

```
// 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 2.3: Example of a contract with a function modifier.

In the Listing 2.3, 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.

2.2.4 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].

In Listing 2.4, 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.

2.2.5 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.

[4]Oracles act like source of information from the outside of a smart contract.

```
// 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 2.4: Example of a contract with an event.

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

contract Orderbook {

    struct Order {
        uint amount;
        uint price;
        uint timestamp;
        address trader;
        bytes2 status;
    }
}
```

Listing 2.5: Example of a contract with a struct.

In Listing 2.5, 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.

2.2.6 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. We 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 [10] refers to a comprehensive understanding of all available data types and their specific use cases.

2.2.7 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 describe 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:

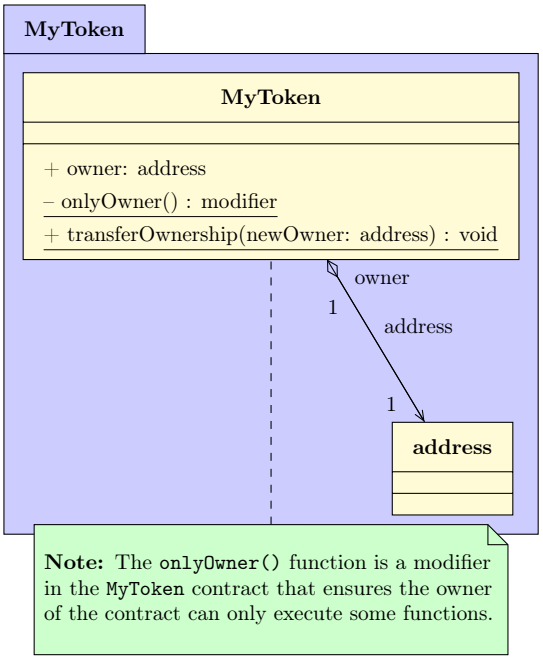

Figure 2.3: UML Class Diagram for Solidity Code

Figure 2.3 shows a UML class diagram for the Solidity code presented in the Listing 2.2. The `tikz-uml` package provides macros for creating UML classes, sequences, and other UML diagram types.

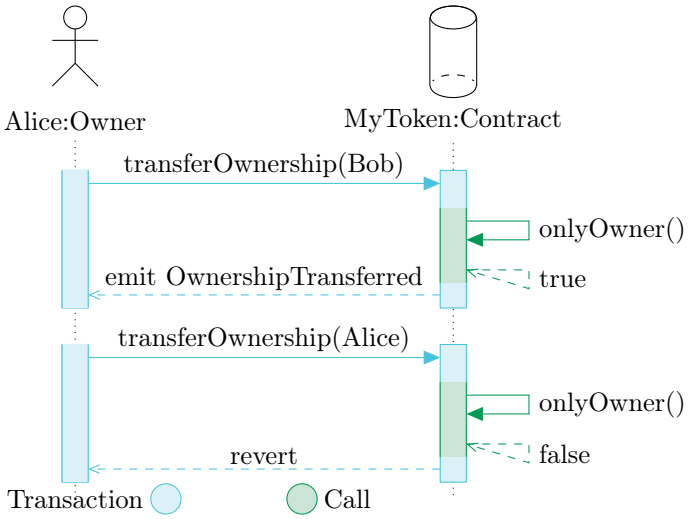

Figure 2.4: UML Sequence Diagram for presenting the function `transferOwnership`.

Figure 2.4 illustrates a UML sequence diagram corresponding to the Solidity code presented in Listing 2.2. The diagram captures the sequence of interactions between the actors, 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 right to Bob. The diagram distinguishes between transactions and calls using different colors. For example, 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.

2.3 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, 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.

2.3.1 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 [7], 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 DEFCON [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 acts like a crowdfunding mechanism where we can sell our tokens in exchange for cryptocurrencies such as Bitcoin (BTC) or Ether (ETH). 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.

2.3.2 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.

2.3.3 Token standards

As explained in the previous section 2.3.2, the most approved token standard is the ERC20 which we describe more in detail in the section 2.4. 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 2.2 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 2.2: List of some EIPs token standards derived from the ERC20.

Table 2.2 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) [11]. The EIPs website is a valuable resource for the community, providing access to drafts and proposals open for review and discussion.

2.4 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.

2.4.1 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 [12]. 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 [12].

```
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 2.6: ERC20 interface presented in Solidity.

The listing 2.6 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 *UET* 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 representation of the smallest fraction of a token because of the absence of floating-point numbers in Solidity. If `decimals` returns 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 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 of 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. Therefore 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 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 representation of the smallest fraction of a token with an integer value. For example, if `decimals()` equals 2, it can 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 :

$$\text{value} = \text{UET} \times 10^{\text{decimals}} \tag{2.2}$$

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. However, it is not possible to transfer a fraction of a share.

2.4.2 Methods of the ERC20 interface

The `allowance` method shows how many tokens Alice (owner) allows Bob (spender) 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 2.7 shows the implementation of the `allowance` function in the OpenZeppelin library. Notably, this function is classified as a `view` function, which means that the state of the contract is not modified. Thus no gas is consumed 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 2.7: OpenZeppelin implementation (v4.8.3) of the ERC20 `allowance` function.

The `approve` method enables Alice (owner) to grant permission to Bob (spender) 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 2.8 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 2.8 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

```solidity
/**
* @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
*
* This internal function is equivalent to `approve`, and can be used to
* e.g. set automatic allowances for certain subsystems, etc.
*
* Emits an {Approval} event.
*
* Requirements:
*
* - `owner` cannot be the zero address.
* - `spender` cannot be the zero address.
*/
function _approve(address owner, address spender, uint256 amount) internal virtual {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address");

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}
```

Listing 2.8: OpenZeppelin implementation (v4.8.3) of the ERC20 `approve` function.

the zero address are permanently lost. Finally, it is employed for checking invalid addresses 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 Alice (owner), 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 2.9 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 Bob (spender), 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.

```
/**
* @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 2.9: OpenZeppelin implementation (v4.8.3) of the ERC20 `transfer` function.

```
/**
* @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 2.10: OpenZeppelin implementation (v4.8.3) of the ERC20 `transferFrom` function.

Listing 2.10 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 2.9) to carry out the actual token transfer from the sender to the recipient.

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 [12], 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)*

The functionality of the ERC-20 token standard, including the `approve`, `transfer`, `allowance`, and `transferFrom` methods, is illustrated in Figure 2.5. 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.

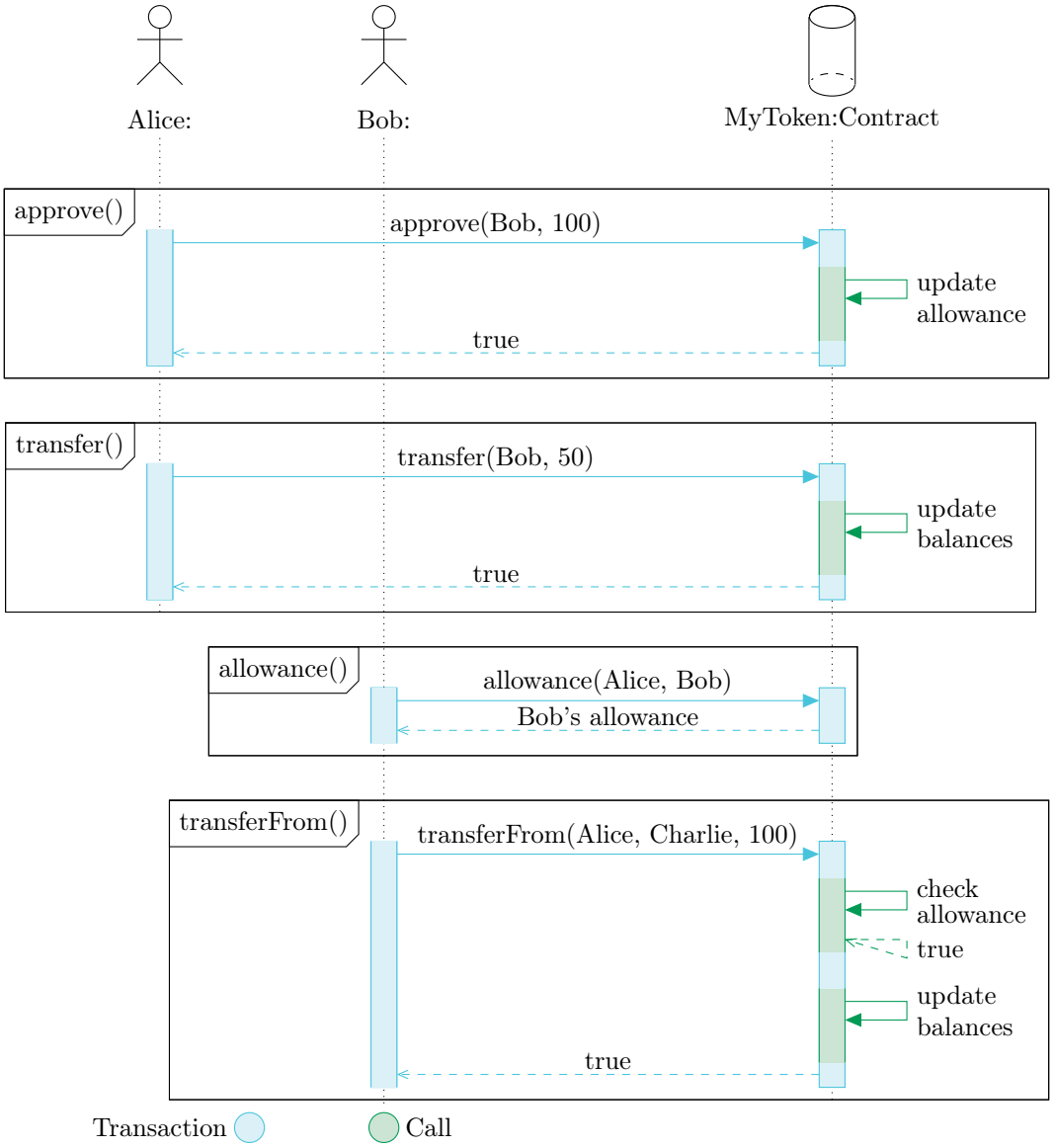

Figure 2.5: UML sequence diagram of the ERC-20 methods

2.5 OpenZeppelin, a secure library for smart contracts

As demonstrated in the previous section 2.4.2, some implementations of methods are illustrated with the codes coming from OpenZeppelin github repository [13]. 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 [12], Fabian Vogelsteller, the author of the ERC-20 standard, recommends to use the OpenZeppelin implementation of the ERC-20 standard [14]:

> *There are already plenty of ERC20-compliant tokens deployed on the Ethereum network. Various teams have written different implementations with 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 has been widely recognized for its focus on security best practices. In addition, 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. In addition, 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 [15]. 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 `Ownable` contracts. 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.

2.6 Truffle, a development environment for Ethereum

Truffle [16] 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.

2.7 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 [17] comes in. The web3 concept emerged as a way to interact with the following internet generation that aims to decentralize the web from the current client-server model. 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)[17]. The Ethereum foundation [17] 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 [18] 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.

We can develop dApps to connect our application to 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. Furthermore, it enables the company to interact with smart contracts, which means they can manage their digital assets, such as tokens, and participate in the decentralized finance (DeFi)[18] applications.

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

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

[18] `https://en.wikipedia.org/wiki/Decentralized_finance`

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.

3 | Methodology & models

This chapter describes the methodology and models used to design and implement our international settlement system to achieve the objectives of this project. We focus on the creation of a decentralized and secure platform for token exchange. We use a methodology and develop two proof-of-concept implementations. These proofs of concepts serve as concrete examples to demonstrate the functionalities and feasibility of our proposed solution.

In section 3.1, we present our first proof of concept, developing a wallet containing our new virtual currency, a fungible token. This wallet enables users to securely store and manage our tokens and others. By implementing this proof of concept, our purpose is to give the possibility to the users to store and manipulate tokens within our international settlement system, which is decentralized and secure. This proof of concept is the foundation for our overall project and provides the basis for developing the second proof of concept.

After developing the first proof of concept, the section 3.2 presents our second proof of concept, focusing on creating a decentralized exchange for token exchange. This component is the core of our international settlement system, enabling users to exchange tokens transparently and securely as a decentralized marketplace. In developing our platform exchange, we aim to demonstrate the need for a reliable and secure environment where users can confidently engage in token exchange without intermediaries. This proof of concept demonstrates the functionalities of our international settlement system and serves as a concrete example of its potential to replace the current financial system.

3.1 Wallet for fungible tokens

This section presents the design and implementation of our first proof of concept, which demonstrates the development of a digital asset wallet for our international settlement system. The digital asset represents a smart contract, more specifically our virtual currency which is a fungible token named "Universal Exports Token" (UET).

To simplify the design and implementation of the smart contract, we leverage the ERC20 standard [12] with the OpenZeppelin library [14]. This combination reduces the complexity involved in creating the smart contract. Once the smart contract is deployed, we build a blockchain application that serves as a digital asset wallet. The application is developed using the Truffle framework [16] and the ReactJS framework [19]. These frameworks enable us to seamlessly connect to a local Ethereum network facilitated by Ganache-cli [20]. This local network provides a controlled environment for testing and interaction with the digital asset wallet.

3.1.1 Token Smart Contract

For our digital asset, we create a smart contract in Solidity [10] that implements the ERC20 standard [12] using the OpenZeppelin library [14]. The diagram in Figure 3.1 shows the class diagram of the *UEToken* to have a better understanding of the design of the smart contract. The diagram shows the inheritance of the *UEToken* smart contract from the *ERC20Capped* and *Ownable* smart contracts. The *ERC20Capped* contract inherits from the *ERC20* contract, which implements the ERC20 standard.

This design choice allows us to leverage the features and security enhancements the OpenZeppelin library provides, ensuring a standardized implementation of the ERC20 standard and incorporating best practices for secure smart contract development.

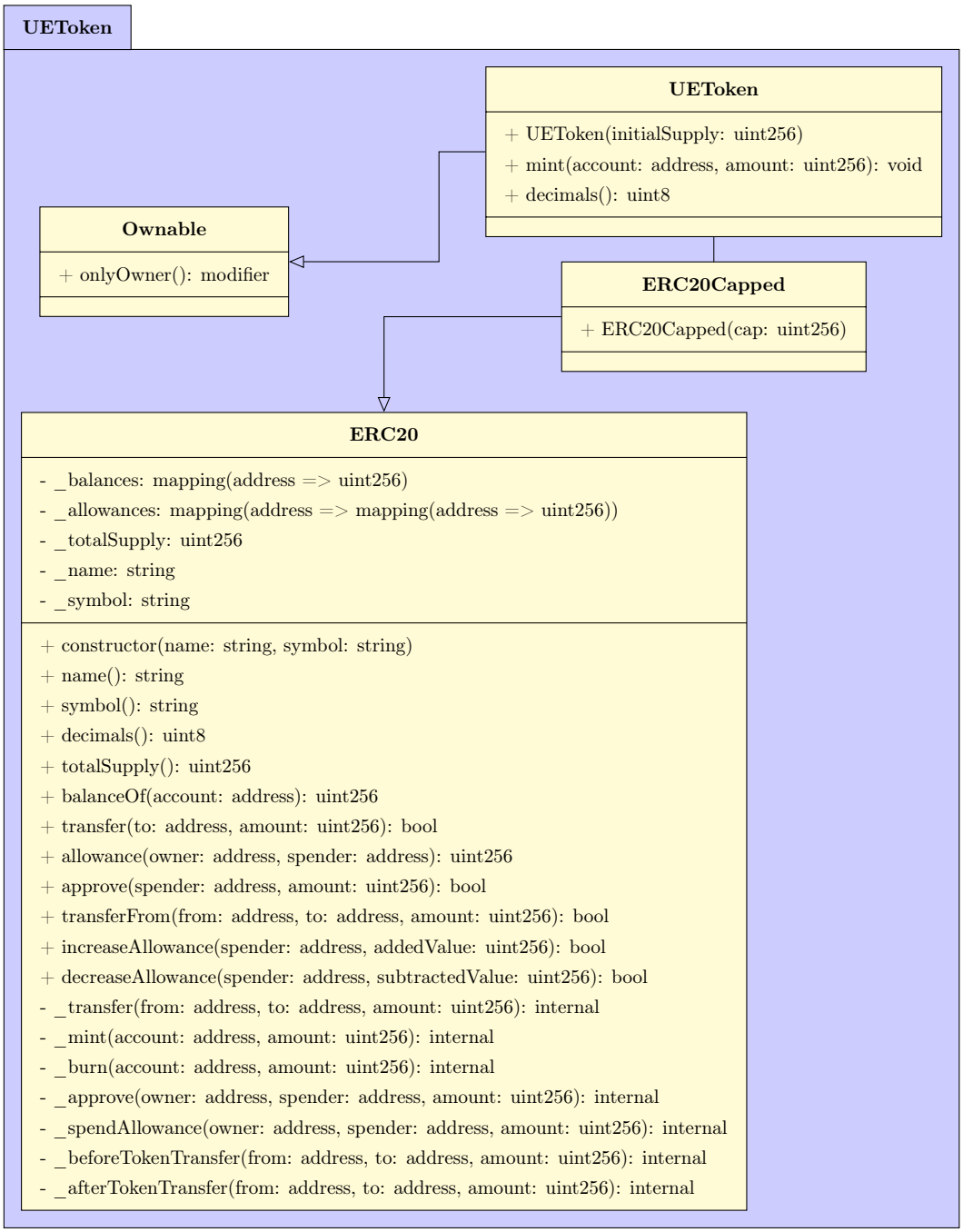

Figure 3.1: Class diagram of the UEToken smart contract using OpenZeppelin's ERC20 implementation.

Now that we have a better understanding of the design of the smart contract, we can delve into the details of the *UEToken* smart contract. The code snippet in Listing 3.1 represents the Solidity source code of our *UEToken* smart contract.

```
1  // contracts/GLDToken.sol
2  // SPDX-License-Identifier: MIT
3  pragma solidity ^0.8.0;
4
5  import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol";
6  import "@openzeppelin/contracts/access/Ownable.sol";
7
8  contract UEToken is ERC20Capped, Ownable {
9      constructor(
10         uint256 initialSupply
11     ) ERC20("Universal Export Token", "UET") ERC20Capped(1000000000) {
12         _mint(msg.sender, initialSupply * 10 ** decimals());
13     }
14
15     function mint(address account, uint256 amount) public onlyOwner {
16         _mint(account, amount);
17     }
18
19     function decimals() public view virtual override returns (uint8) {
20         return 2;
21     }
22 }
```

Listing 3.1: Solidity source code of the UEToken smart contract.

Line 3 of the code snippet in Listing 3.1 shows the Solidity version used for the smart contract. For the writing of this report, we are using the latest version of Solidity, version 0.8.0.

Lines 5 and 6 illustrate the import of the *ERC20Capped* and *Ownable* contracts from the OpenZeppelin library. Let us dive into the *ERC20Capped* contract to understand how it is designed. In figure 3.2, we can show the design of the *ERC20Capped* contract in detail. As illustrated in Figure 3.1, the *ERC20Capped* contract inherits from the *ERC20* contract. ERC20Capped acts as an abstract contract that inherits from the *ERC20* and overrides the `_mint` function to add a token supply cap by verifying that the total supply of tokens does not exceed the cap.

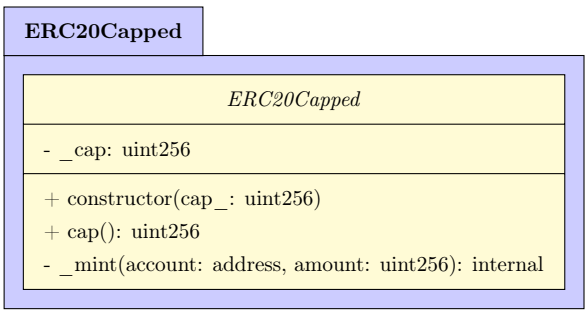

Figure 3.2: Class diagram of the OpenZeppelin ERC20Capped smart contract.

By extending the `ERC20Capped` contract, our token has a capped supply, meaning that the total supply of tokens cannot exceed a specified limit. By setting this cap on the supply of tokens, we control the maximum number of tokens that can never be created. It can be helpful in some use cases, such as creating a stablecoin, where the supply of tokens is backed by a reserve of assets, or in maintaining a fixed token economy. In our use case, we are not backing our token with a reserve of assets or fiat

currency, but we are using the cap to limit the supply of tokens to a maximum number of tokens. Therefore, we set the cap to 1 billion tokens, the maximum number of tokens that can be minted. This provides us transparency and trust by incorporating this cap to token holders and users of our token. The cap is set in the constructor of the `UEToken` smart contract, as shown in Listing 3.1. The other smart contract we import from the OpenZeppelin library is the `Ownable` smart contract. We illustrate in Figure 3.3 the detailed design of the OpenZeppelin `Ownable` smart contract.

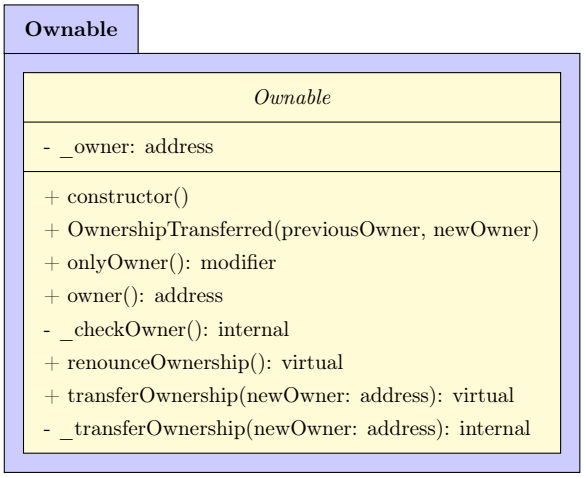

Figure 3.3: Class diagram of the OpenZeppelin Ownable smart contract.

The `Ownable` contract implements a direct access control mechanism in our `UEToken` smart contract. By inheriting the `Ownable` smart contract into our token implementation, we can grant exclusive access to specific functions, such as the `mint` function used for creating new tokens. Our token uses the `onlyOwner` modifier, which limits the execution of these functions to the contract owner. Consequently, only the owner who deployed the smart contract can execute the `mint` function, as it is annotated with the `onlyOwner` modifier. In our case, we aim to demonstrate the controlled minting of new tokens exclusively by the contract owner, preventing the possibility of unauthorized minting that could devalue the tokens. By default, the `Ownable` contract sets the contract owner in its constructor. However, the ownership can be transferred by calling the `transferOwnership` function, offering flexibility in contract management when necessary. The `Ownable` contract is utilized in scenarios where specific critical functions, such as the `mint` function, or administrative tasks must be restricted to the contract owner.

Lastly, in line 11 of the code snippet in Listing 3.1, the name and symbol of the token are defined in the constructor using the `ERC20` constructor inherited from the *ERC20Capped* contract. As explained in the previous chapter (Chapter 2), specifically in Section 2.4, the ERC20 token provides a standard interface for fungible tokens. In Figure 3.1, the `ERC20` constructor accepts the name and symbol of the token as parameters, which are set to `UEToken` and `UE`, respectively. Additionally, our constructor takes an `initialSupply` parameter, which sets the initial supply of tokens upon deployment of the smart contract.

In line 12, we invoke the `_mint` function from the *ERC20Capped* contract to mint the initial supply of tokens. The `_mint` function requires the address of the account to be minted (i.e., the contract owner) and the number of tokens to be minted (i.e.,

the initial supply of tokens). Furthermore, in our contract in line 15, we create the `mint` function, utilizing the `_mint` private function from the *ERC20Capped* contract. The `mint` function accepts the account's address to be minted and the number of tokens to be minted as parameters. The `onlyOwner` modifier is applied to the `mint` function to restrict its execution solely to the contract owner. Our platform aims to demonstrate the contract owner's minting of new tokens exclusively.

In line 19, we override the `decimals` function from the *ERC20* contract to set the number of decimals to 2. By default, the `decimals` function returns 18 decimals. However, for our token representing standard currency (e.g., Euro), we set the number of decimals to 2 to align with the typical decimal representation of currency. This adjustment ensures compatibility with financial calculations and representations.

All these explications complete the necessary code for our proof of concept token smart contract. While we could have included additional functionality, such as a `burn` function for burning tokens, our focus remains on the primary feature of our platform, which is the exchange of tokens between users. The complete OpenZeppelin code can be found in the GitHub repository [13].

Our fully functional code serves as a template for creating other ERC20 tokens. For instance, we can create a stablecoin called `USDC` (USD Coin) by modifying the name, symbol, and maximum supply of tokens. Additionally, the return value of the `decimals` function can be adjusted to set the desired number of decimal places for the token. The extensibility of our token smart contract is one of the advantages of adhering to the ERC20 standard. We can leverage our token smart contract as a foundation to create multiple tokens according to our requirements.

3.1.2 Token Deployment

To deploy our token smart contract, *UEToken*, we can utilize Truffle [16], a development environment framework for Ethereum. Truffle offers a comprehensive suite of tools, including a development environment, testing framework, and asset pipeline for Ethereum. One of the critical features of Truffle is the ability to manage contract deployments through migrations.

Migrations in Truffle are written in JavaScript and allow us to define the deployment phases of our contracts. Each migration file represents a separate deployment step, and Truffle ensures that migrations are executed in the correct order. This ensures that the contracts are deployed in a coordinated manner.

Additionally, Truffle provides a console, a JavaScript runtime environment that exposes the Truffle environment to the command line. The console allows us to interact with our contracts and execute JavaScript commands against the deployed contracts.

Before deploying our contract, we need to compile it to transform the Solidity code into bytecode that can be understood by the Ethereum Virtual Machine (EVM) on the blockchain. Truffle simplifies the compilation process by providing a built-in compiler. The snippet in Listing 3.2 shows the compilation of our *UEToken* smart contract using the Truffle compiler :

```
$ truffle compile
Compiling your contracts...
===========================
> Compiling ./contracts/UEToken.sol
> Compiling @openzeppelin/contracts/access/Ownable.sol
> Compiling @openzeppelin/contracts/token/ERC20/ERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/IERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol
> Compiling @openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol
> Compiling @openzeppelin/contracts/utils/Context.sol
> Artifacts written to /truffle/build/contracts
> Compiled successfully using:
   - solc: 0.8.19+commit.7dd6d404.Emscripten.clang
```

Listing 3.2: Compiling the *UEToken* smart contract.

As we can see in Listing 3.2, Truffle searches for all Solidity files in the *contracts* directory and dependencies in the *node_ modules* directory. It compiles them according to the Solidity version specified in the *truffle-config.js* file. Finally, the compiled contracts are written to the *build/contracts* directory.

Once the compilation is successful, we can proceed to the migration process of our contract. To begin the migration process, we must create a new migration file, such as *uetoken_ migration.js*, in the Truffle migrations directory *migrations*. The migration file name should be descriptive of the contract that is being deployed. Therefore, we name the migration file *uetoken_ migration.js*, meaning we are deploying the *UEToken* contract. In this file, we need to specify the deployment code for our token contract. The code for deploying our token contract is shown in Listing 3.3.

```
const UEToken = artifacts.require("UEToken");

module.exports = function(_deployer) {
  // Use deployer to state migration tasks.
  const initialSupply = 1000000
  _deployer.deploy(UEToken, initialSupply);
};
```

Listing 3.3: Deployment code for the *UEToken* smart contract.

In the above code, we use the `artifacts.require` function to import the *UEToken* contract artifact. Then, in the migration deployment function, we specify the initial supply of tokens to mint and deploy the *UEToken* contract using the `_deployer.deploy` function.

Once the migration file is created, we can deploy our token contract on a local test network. We use a local test network for our proof of concept because it is faster and cheaper to deploy our contract on a local test network than on the Ethereum mainnet or testnet such as Ropsten. We use Ganache-CLI [20] for the local test network, simulating our private Ethereum blockchain. Ganache-CLI is a command-line version of Ganache, a personal blockchain for Ethereum development. Ganache-CLI allows us to customize our local blockchain with numerous options, such as the gas limit, gas price, and chain ID, allowing us to simulate the Ethereum mainnet or testnet freely.

We can start Ganache-CLI by running the following command:

```
$ ganache-cli -m "other strike boat weekend address want pink oval sister cry
↪  excuse myth" --chainId 1 --networkId 1337

Ganache CLI v6.12.2 (ganache-core: 2.13.2)

Available Accounts
==================
(0) 0x216384d2f868d00ddC2907151F01b392aE0de155 (100 ETH)
[snip]
(9) 0x2d9d21e8ef8b0ebb4bcba4a9b79cd5d8dcfba7a7 (100 ETH)

Private Keys
==================
(0) 0xb406b5f9eb128076dea028c9c149918a8811d193d93dfbde0489c316129d8f31
[snip]
(9) 0x94dec33a7e428dbd78a527fb9e9837645f2b488c040f54e90847d401a1008767
```

Listing 3.4: Starting Ganache-CLI.

The output shown in Listing 3.4 indicates that Ganache-CLI has successfully started a local Ethereum blockchain with ten accounts, each having a balance of 100 ETH. To ensure consistency, we utilize the `-m` argument to specify a mnemonic phrase for the blockchain. This allows us to start the identical blockchain with the same accounts whenever needed, which is handy for testing. Additionally, we use the `-chainId` argument to set the chain ID of the blockchain. The chain ID is used to identify the specific blockchain network, and in our case, we set it to one for testing purposes. This aligns with the configuration we use in Metamask [21] to connect to our local blockchain. It is worth mentioning that when importing ERC20 tokens into Metamask, we need to specify the chain ID as 1 (Ethereum mainnet) due to a bug in Metamask [22]. Furthermore, we utilize the `-networkId` argument to define the network ID of the blockchain. The network ID serves as an identifier for the blockchain network, and for testing purposes, we set it to 1337.

To deploy our token contract on the Ganache-CLI blockchain, we need to execute the following command:

```
$ truffle migrate --network ganache

[snip]

uetoken_migration.js
====================

   Replacing 'UEToken'
   -------------------
   > transaction hash:
   ↪  0xb79defe9f743e4592560da60dabee61e0a66ee017a4f90a0c1d20720061659e0
   > Blocks: 0            Seconds: 0
   > contract address:    0xd543BdeE60836107Ad2F70b2384a01827a61AF12
   > block number:        2
   > block timestamp:     1685382153
   > account:             0x216384d2f868d00ddC2907151F01b392aE0de155
   > balance:             99.93655312
   > gas used:            1586172 (0x1833fc)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.03172344 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:          0.03172344 ETH

Summary
=======
> Total deployments:   1
> Final cost:          0.03172344 ETH
```

Listing 3.5: Deploying the token contract on the ganache-cli blockchain.

In Listing 3.5, we can see that the command instructs Truffle to execute the migration script `uetoken_migration.js` to the specified network. In this case, the network is `ganache`, defined in the `truffle-config.js` file, and is the local blockchain we started with Ganache-CLI. Upon successful deployment, we can see in Listing 3.5 that Truffle summarizes the migration process, including the transaction hash, the contract address, the block number, and transaction details. We can then interact with the deployed contract using the contract address.

To verify with Truffle that our token functionalities work as expected, we can run the following command:

```
$ truffle console --network ganache

truffle(ganache)> let instance = await UEToken.deployed()
undefined
truffle(ganache)> let accounts = await web3.eth.getAccounts()
undefined
truffle(ganache)> let name = await instance.name()
undefined
truffle(ganache)> name
'UEToken'
```

Listing 3.6: Calling the `name()` function our token contract from Truffle.

In Listing 3.6, we can see that we first get an instance of the deployed token contract, then we get the list of accounts from the blockchain, and finally, we call the `name()` function of the token contract. The result is the name of the token, which is `UEToken`. Leveraging Truffle's console, we can easily interact with the deployed token contract.

3.1.3 Frontend Development

After the deployment of our new token contract on the blockchain network in the previous section, we must develop a front-end application for interacting with this smart contract acting as backend. For the proof-of-concept, we focus on creating a simple front-end application acting like a wallet that allows us calls the core functionalities of our token, i.e., the smart contract functions. Thus, this section demonstrates an overview of the main components of our front-end application and how we interact with the token contract from the front.

For the frontend development, we utilize React [19], a widely-used JavaScript library for building user interfaces. Additionally, we use the Web3.js [18] library (refer to Section 2.7), which provides a collection of modules that enable interaction with Ethereum and our deployed token contract.

Web3.js is a package that can be installed using the Node Package Manager (NPM). It allows us to connect to an Ethereum network and interact with our deployed token contract within our front-end application. In addition, the library offers various modules to handle different tasks, such as contract instantiation, function calls, and even listening.

We first connect with our web3 instance to initiate our development process. In our case, we rely on Metamask[1] to connect to our local blockchain (Ganache). Metamask injects a global variable called `web3` into the browser's window object.

We initialize our web3 instance by verifying the installation and unlocking of Metamask. Once confirmed, we instantiate our web3 instance using the `Web3` constructor. Below is a code snippet demonstrating our web3 initialization process:

[1]Metamask is a browser extension that enables us to connect to an Ethereum network directly from our browser.

```
import Web3 from 'web3';
// Code snipped for brevity
mount = async () => {
var account;
    if (window.ethereum) {
        try {
            await window.ethereum.request({ method: 'eth_requestAccounts'
            ↪ }).then((accounts) => {
            account = accounts[0];
            this.web3.eth.defaultAccount = account;
            // Code snipped for brevity
            erc20Tokens.forEach((token) => {
                let erc20Token = new this.web3.eth.Contract(token.abi, token.address);
                // Call any function to get the token name
                erc20Token.methods.name().call().then((name) => {
                app.setState({
                    tokens: [...app.state.tokens, { ...token, name }]
                    });
                });
            });
        });
        } catch (error) {
            console.error('Error connecting to web3:', error);
        }
        const ethereum = window.ethereum;
        window.web3 = new Web3(ethereum);
        this.web3 = new Web3(ethereum);
    } else {
        console.log('No ethereum browser detected');
    }
}
```

Listing 3.7: Web3 initialization.

In Listing 3.7, we demonstrate how to manage the web3 instance and the connected account state in our React component. Upon refreshing the browser, we utilize the `mount()` lifecycle method to handle this process. Firstly, we check for the availability of MetaMask and request the user's accounts. Once connected, we create a new instance of the `Web3` object and set the default account to the first account in the list provided by MetaMask.

To interact with our deployed token contract, we leverage the web3 instance. For instance, we can call various functions of our token contract, such as `name()`, `symbol()`, `totalSupply()`, and `balanceOf()`. This involves accessing the `methods` object of our token contract instance and invoking the respective functions. By retrieving the returned values, such as the token name *Universal Export Token* (UET), symbol, total supply, and balance, we can update the state of our React component accordingly.

To obtain an instance of our token contract, we must provide the token contract's ABI and address to the `Contract` constructor of the `web3.eth` object. The ABI is obtained from the JSON file located in the `build/contracts` directory, as demonstrated in the deployment section (Section 3.1.2). It serves as a blueprint for Web3, enabling it to interact correctly with our token contract by understanding its functions and events.

In our proof-of-concept wallet platform, we can list all the user's tokens. But first, we must supply each token's respective ABI and contract addresses. Our frontend application, implemented as a React component, renders an array of tokens, each

with its own instantiated `Contract` object. This allows us to call the `balanceOf()` function of the token contract, retrieving the user's balance for that particular token. Additionally, buttons associated with each token enable users to perform actions such as transferring tokens, approving transfers, and minting new tokens.

For example, to call the `transfer()` function, the following code snippet can be utilized:

```
Transfer = async () => {
    // Code snipped for brevity
    tokenContract = new this.web3.eth.Contract(this.state.transferDetail20.abi,
    ↪  this.state.transferDetail20.address);
    var amount = this.state.fields.amount * (Math.pow(10,
    ↪  this.state.transferDetail20.decimal));
    try {
        await tokenContract.methods.transfer(recipient, amount).send({ from:
        ↪  this.web3.eth.defaultAccount });
        console.log('Transfer successful!');
    } catch (error) {
        console.error('Error transferring tokens:', error);
    }
}
```

Listing 3.8: Calling the `transfer()` function of the web3 token contract instance.

In the provided code snippet (Listing 3.8), we begin by creating a new instance of the `Contract` object, which requires the ABI (Application Binary Interface) and address of the token contract. Then, to ensure the accuracy of the transfer amount, we multiply the user-entered amount by the decimal value of the token (as described in Equation 2.4.1). Next, we call the `transfer()` function of the token contract, passing the recipient's address and the desired transfer amount as arguments. This function call returns a JavaScript promise, allowing us to handle the response using the `then()` function. Through this, we can update the state of our React component accordingly. The exact process applies to other functions, such as `approve()` and `mint()`, with their respective arguments.

3.2 Decentralized Exchange (DEX)

This section describes our second proof-of-concept, the development of a decentralized exchange (DEX) platform responsible for trading ERC20 tokens, as mentioned in the objectives of this project. Traditionally, trading exchanges are centralized, meaning that they are owned and operated by an authority for facilitating the trading between a buyer and a seller. This authority is responsible for posting a buy or sell order on behalf of the user where the order is visible to other users of the exchange. This process is called an order book, where the exchange maintains a list of buy and sell orders submitted by users. Traders can submit orders to the order book, and the exchange will match the orders based on the price and quantity of the order. In the case of a match, the exchange will execute the trade and update the users' balances accordingly. In our proof-of-concept, we implement a decentralized version of this process. We remove the centralized authority and allow users to interact directly via

a smart contract replacing the order book. In our use case, we implement a simple order book that allows traders to trade our ERC20 token `UEToken` for a stablecoin[2] , `USDC`[3] which are also an ERC20 token.

In our project, we are building the following components:

Two ERC20 tokens

The UEToken is the token we created in the previous section and represents the base asset[4] of our exchange. The USDC token is a stablecoin representing the fiat currency, the US dollar, in the form of a cryptocurrency where in theory, 1 USDC is always equal to 1 USD. Stablecoin represents the counter asset[5] of our exchange. Due we are using our private Ethereum network, we need to simulate the USDC token. To do so, we create a new ERC20 token contract and mint 1000 USDC tokens, for example. In a real case, a centralized authority can issue the USDC token, namely, Circle Internet Financial Ltd. [23]. If we were to deploy our exchange on the Ethereum mainnet, we would use the real USDC token with the address `0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48` [6].

Order book smart contract

The order book smart contract matches buy and sell orders submitted by traders. The order book is implemented as a smart contract deployed on the Ethereum blockchain. The order book contract keeps track of the buy and sell orders submitted by traders and allows traders to view the order book and submit orders against an existing order stored in the contract. The order book contract is implemented in Solidity and is deployed on the local Ethereum blockchain using the Truffle framework with the help of the Ganache CLI.

Trading platform

The trading platform is a web application that allows traders to interact with the order book smart contract. It shows users their balances of the two tokens, base token, and counter token, and allows them to submit buy and sell orders to the order book. When a new order is submitted, an engine verifies if the new request matches an existing order in the order book. If a match is found, the trade is executed, and the users' balances are updated accordingly by invoking the order book smart contract trade function. On the other hand, if no match is found, the order is added to the order book and is visible to other traders by calling the add order function of the order book smart contract. The trading platform is implemented using the React JavaScript framework and the web3.js library for interacting with the Ethereum blockchain.

[2]A stablecoin is a cryptocurrency that is pegged to a stable asset such as gold or fiat currencies.

[3]USDC is a stablecoin pegged to the US dollar, meaning that 1 USDC is always equal to 1 USD. `https://en.wikipedia.org/wiki/USD_Coin`

[4]The base asset is the asset that is being traded.

[5]The counter asset represents the asset we want against.

[6]The USDC token contract address on the Ethereum mainnet. `https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48`

3.2.1 Design of the Order Book Smart Contract

The order book smart contract needs to be designed in the most simple possible, facilitating the trading of ERC20 tokens between buyers and sellers. It keeps track of the buy and sell orders and executes the trades when conditions are met. The design consists of the following components as shown in the class diagram in Figure 3.4.

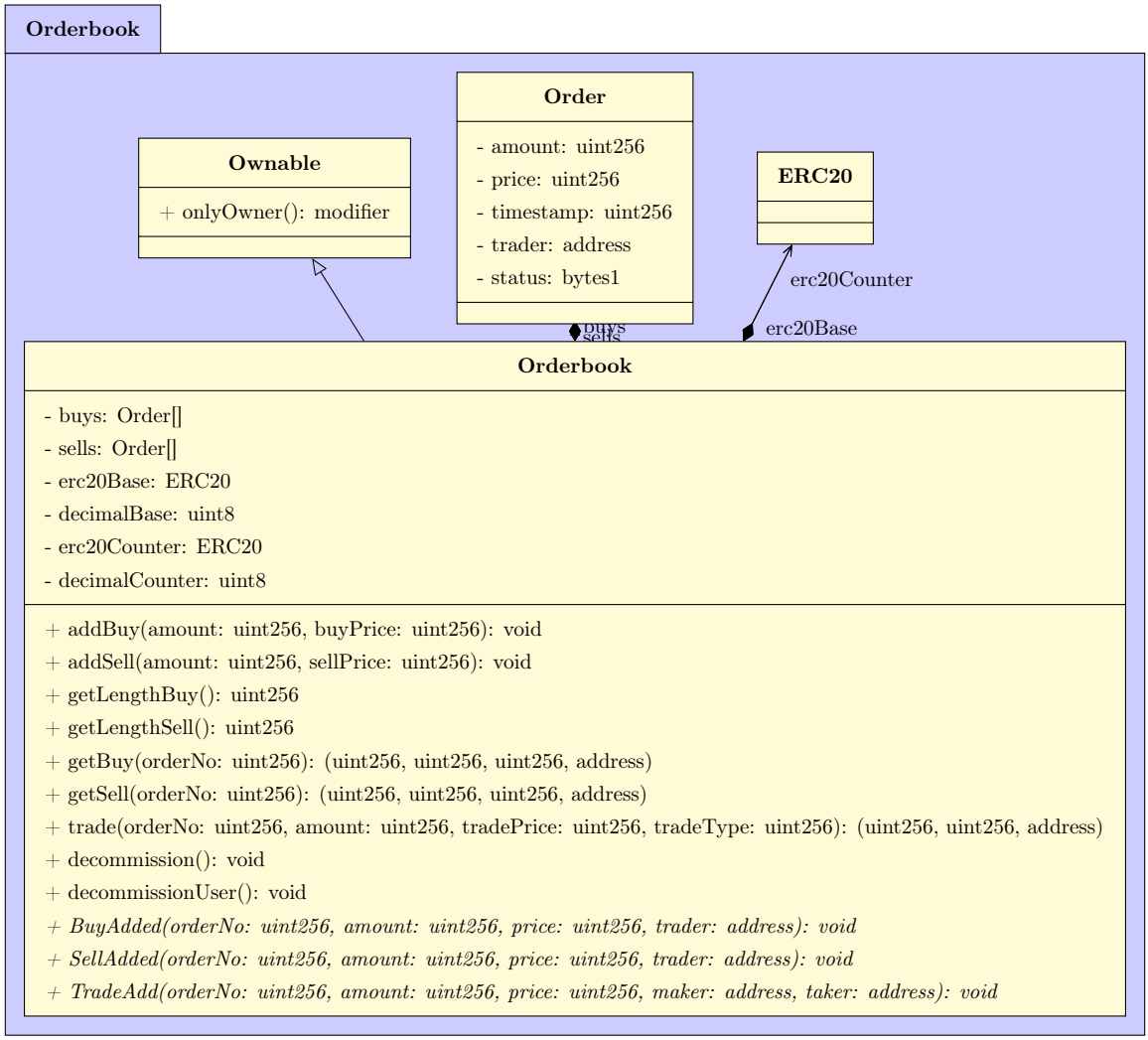

Figure 3.4: Class diagram of the Orderbook smart contract.

Struct Order

The `Order` struct is used for an individual buy or sell order. It contains the following fields:

- **amount**: The number of tokens bought or sold.
- **price**: The price at which the tokens are bought or sold.
- **timestamp**: When the order was created.
- **trader**: The trader's address who placed the order.

- **status**: The status of the order, represented as a two-byte value (e.g., "A" for active, "T" for traded).

buys and sells Arrays

The `buys` and `sells` arrays are used to store the buy and sell orders, respectively. Each array stores instances of the `Order` struct, representing the active orders in the order book.

erc20Base and erc20Counter Contracts

The `erc20Base` and `erc20Counter` contracts represent the ERC20 token being traded in the order book. They are instances of the ERC20 contract from the OpenZeppelin library [14]. The `erc20Base` contract represents the base asset while the `erc20Counter` contract represents the counter asset. Our order book smart contract is designed for only two tokens at a time, meaning that the base and counter assets are fixed. In our use case, the base asset is the `UEToken` ERC20 token while the counter asset is the `USDC` ERC20 token.

Constructor

The constructor of the order book smart contract is responsible for initializing the `erc20Base` and `erc20Counter` contracts. It takes as parameters the addresses of the two contracts and initializes the two contracts, and retrieves the decimals places for each token. The decimal places convert the tokens from the base unit to the token unit. For example, the `UEToken` ERC20 token has two decimals places, meaning that 1 UEToken is equal to 100 base units (refer to Equation 2.4.1 for further clarification).

Events

The contract defines three events:

- **BuyAdded**: Triggered when a buy order is added to the order book. It includes the order number, amount, price, and trader's address.
- **SellAdded**: Triggered when a sell order is added to the order book. It includes the order number, amount, price, and trader's address.
- **TradeAdd**: Triggered when a trade is executed between a buy and sell order. It includes the order number, amount, price, maker's address, and taker's address.

addBuy Function

The `addBuy` function allows a trader to add a buy order to the order book. It takes as parameters the amount and the desired buy price. The function transfers the total amount of tokens to the order from the trader's account to the order book contract using the `transferFrom` method of the `erc20Base` contract. It then adds the order to the `buys` array and emits the `BuyAdded` event.

addSell Function

The `addSell` function allows a trader to add a sell order to the order book. It takes as parameters the amount and the desired sell price. The function transfers the total amount of tokens to the order from the trader's account to the order book contract using the `transferFrom` method of the `erc20Counter` contract. It then adds the order to the `sells` array and emits the `SellAdded` event.

getLengthBuy and getLengthSell Functions

The `getLengthBuy` and `getLengthSell` functions return the length of the `buys` and `sells` arrays.

getBuy and getSell Functions

The `getBuy` and `getSell` functions allow users to view the details of a specific buy or sell order. They take the order number as a parameter and return the amount, price, timestamp, trader's address, and order status.

trade Function

The `trade` function executes a trade between a buy and sell order. It takes as parameters the order number, amount, trade price, and trade type (e.g., "B" for buy, "S" for sell). The function verifies the order number and trade parameters and then applies the trade whether the trade is valid or not. If the trade is valid, the function transfers the tokens between the trader and the counterparty using the `transfer` and `transferFrom` methods of the `erc20Base` and `erc20Counter` contracts, respectively. It updates the status of the order, emits the `TradeAdd` event, and returns the order number, amount, and trader's address of the trade.

`transferForm` is used when transferring tokens from the `msg.sender` (i.e., the caller of the function) to the trade counterparty, which can be either the buyer or seller, depending on the trade type. `transferForm` is used when an external account initiates the token transfer and requires approval from the account holder (i.e., the `msg.sender`). In this case, the `msg.sender` needs to approve the Orderbook contract to spend a certain amount of tokens on the token contract before initiating the `trade` function. Once the approval is granted, the Orderbook contract can execute the `transferFrom` function method to transfer the tokens from the `msg.sender`'s account to the counterparty's account. This ensures that the token transfer is authorized and controlled by the account holder, preventing the Orderbook contract from transferring tokens without the account holder's permission.

`transfer` is used when transferring tokens from the Orderbook contract to the `msg.sender`, which can be either the buyer or seller receiving their respective tokens after successfully executing the trade. In this case, the Orderbook contract acts as the account holder and can transfer tokens to the `msg.sender` without requiring any approval. Since the Orderbook contract holds the tokens in its balance, it can use the `transfer` method to send the tokens to the respective party.

decommission Function

The `decommission` function allows the contract owner to decommission the order book smart contract. It transfers any remaining tokens in the active orders back to the respective traders using the `transfer` method of the `erc20Base` and `erc20Counter`

contracts. It then clears the `buys` and `sells` arrays, resetting the order book to its initial state. Only the contract owner can call this function with the `onlyOwner` modifier.

decommissionUser Function

The `decommissionUser` function allows a trader to decommission all of his or her active orders in the order book. First, check if the trader has active orders in the order book. If the trader has active orders, the function transfers any remaining tokens in the active orders back to the respective trader using the `transfer` method of the `erc20Base` and `erc20Counter` contracts. It then clears the transferred orders from the `buys` and `sells` arrays. It is helpful if a trader wants to decommission all active orders in the order book without waiting to validate the order book contract owner.

The order book smart contract provides a reliable and efficient way to trade ERC20 tokens, ensuring that the trades are executed fairly and transparently without the need for a trusted third party.

3.2.2 Migration of the order book smart contract

To deploy and migrate the Orderbook smart contract to a local blockchain, we use Truffle's migration feature, as explained in Section 3.1.2. Before deploying the Orderbook smart contract, we need to deploy the `UEToken` and `USDC` ERC20 tokens to the local blockchain, as explained in Section 3.1.2. We need this to have our local blockchain, such as Ganache, have the tokens for trading. Once the tokens are deployed, we can deploy the Orderbook smart contract using the following steps.

Deployment Script

In the *migrations* folder, we create a new file called *1_initial_migration.js*. This file is responsible for deploying smart token contracts. Then we create a new file called *2_deploy_orderbook.js* to deploy the Orderbook smart contract in the same folder to handle the deployment of the Orderbook smart contract.

Migration scripts

In the deployment script, we first import the `UEToken` and `USDC` smart contracts artifacts from the *build/contracts* folder. These contracts are generated by Truffle when compiling the smart contracts. Then we need to also get their respective addresses from the summary of the deployment of the tokens. The development script is shown in Listing 3.9.

```
1 const Orderbook = artifacts.require('Orderbook');
2
3 module.exports = function (_deployer) {
4   const baseTokenAddress = '0x...'; // Address of the base token ERC20 contract
5   const counterTokenAddress = '0x...'; // Address of the counter token ERC20 contract
6   _deployer.deploy(Orderbook, baseTokenAddress, counterTokenAddress);
7 };
```

Listing 3.9: Orderbook migration script for development.

Deployment

Before deploying the Orderbook smart contract, we must ensure that the network configuration in the *truffle-config.js* file is set to the local blockchain network. Then we can run the migration command to display the Orderbook contract. In the command line, we run the following command `truffle migrate --reset`,

Deployment on the Rinkeby testnet

To deploy the Orderbook smart contract on the Rinkeby testnet, we need to ensure that the network configuration in the *truffle-config.js* file is set to the Rinkeby testnet. Then we can run the migration command to display the Orderbook contract. In the command line, we run the following command `truffle migrate --reset --network rinkeby`.

Verify the deployment

After successfully migrating the Orderbook smart contract, Truffle generates a summary of the deployment, which includes the addresses of the deployed contracts on the local blockchain.

Interacting with the order book smart contract

We can use the Truffle console to interact with the Orderbook smart contract. In the command line, we run the following command `truffle console`. This will open the Truffle console, which allows us to interact with the deployed smart contracts. We can then use the `Orderbook.deployed()` command to get the deployed Orderbook smart contract instance. We can then use the instance to call the functions of the Orderbook smart contract. For example, we can call the `addBuy` function to place a buy order in the order book. The following code snippet in Listing 3.10 shows how to place a buy order in

```
const orderbook = await Orderbook.deployed();
const amountUEToken = web3.utils.toWei('100', 'ether');
const amountBuyPrice = web3.utils.toWei('0.0001', 'ether');
await orderbook.addBuy(amountUEToken, amountBuyPrice);
const buy = await orderbook.getBuy(orderbook.getBuyLength() - 1);
console.log(buy);
```

Listing 3.10: Placing a buy order in the order book with Truffle console.

3.2.3 Implementation of the trading platform

The same as in Section 3.1.3, we use React to implement the trading platform. In this section, we explain the main components of the trading platform and how they interact with the Orderbook smart contract without explaining the implementation details. The main components of the trading platform are the following:

Event listener

The order book is displayed in a table with two columns, one for the buy orders and one for the sell orders. Each row of the table represents an order in the order book. The order book is updated in real-time using the `getSell` and `getBuy` functions of the Orderbook smart contract. When populating the order `buys` and `sells` arrays,

the Orderbook smart contract emits an event. We can listen to these events in our trading platform to update the order book in real-time with new orders. The following code snippet shows how to listen to, e.g., the `BuyOrder` event emitted by the Orderbook smart contract can be caught in the trading platform. of tokens from one address to another

```
watchOrderbook = async () => {
    await this.watchweb3.Contract(this.orderbookABI, orderbook.address).then((contract) =>
    ↪ {
        // Update the order book with the new buy orders
    });
}
```

Listing 3.11: Listening to the Orderbook smart contract events in the trading platform.

In Listing 3.11, we use the `watchweb3` library to listen to the events emitted by the Orderbook smart contract. Then, when a new event is emitted, we update the order book with the new orders and refresh the table to display the new orders.

Buy and sell orders

The following method in our trading platform is the ability to place buy and sell orders. To place a buy order, we call the `addBuy/addSell` functions of the Orderbook smart contract. For example, the following code snippet shows how to place a buy order in the order book.

```
addBuy = async () => {
    const app = this;
    const usdcAmount = this.state.fields.buyAmount
    const amountBuyPrice = this.state.fields.buyPrice;
    const approvalAmount = usdcAmount * amountBuyPrice;
    const usdcContract = await this.web3.eth.Contract(app.USDC.abi, app.USDC.address);
    const uetContract = await this.web3.eth.Contract(app.UET.abi, app.UET.address);
    await this.web3.eth.Contract(app.orderbookABI.abi,
    ↪  app.orderbookABI.address).then(async (contract) => {
        // check if the sells array is empty
        if(this.state.sells.length === 0)
        {
            await usdcContract.methods.approve(app.orderbookABI.address, approvalAmount*
            ↪  10 ** 2).send({ from: app.web3.eth.defaultAccount }).then(async (receipt)
            ↪  => {
                await contract.methods.addBuy(usdcAmount, amountBuyPrice).send({ from:
                ↪  app.web3.eth.defaultAccount }).then(async (receipt) => {
                    // Update the order book with the new buy orders
                });
            });
        } else {
            // Look for a matching sell order
            // Execute the trade
        }
    });
}
```

Listing 3.12: Placing a buy order in the order book from the trading platform.

In Listing 3.12, we start by getting the amount of UEToken and the buy price from the state of the trading platform. Then we call the `addBuy` function of the front-end platform with the amount of UEToken and the buy price as parameters. We also specify the account that is placing the buy order. Then, we check if the `sells` array is empty, meaning there are no sell orders in the order book. If the `sells` is empty, then we skip the next step, looking for a matching sell order. We look for a matching sell order if the `sells` array is not empty and if we find a matching sell order, then we execute the trade.

For the empty, we first use the `approve` function of the USDC smart contract to approve the Orderbook smart contract to transfer the amount of USDC specified in the buy order that is worth the same as the order amount from the user's account. The approval amount is multiplied by the decimals of the USDC, which is 2. Then, to receive the successful transaction receipt, we call the `addBuy` function of the Orderbook smart contract to place the buy order in the order book. Finally, with our `watchOrderbook` method, we update the order book with the new buy orders.

Matching buy and sell orders

If the `sells` array is not empty, then we need to look for a matching sell order. To do so, we loop through the `sells` array and check if the amount of UEToken of the sell order is greater than or equal to the amount of UEToken of the buy order. If the amount of UEToken of the sell order is greater than or equal to the amount of UEToken of the buy order, then we execute the trade. The following code snippet shows how to execute the trade.

```
1  // approve the Orderbook smart contract to transfer the amount of UEToken specified in the
   ↪  buy order from user's account
2  await uetContract.methods.approve(app.orderbookABI.address, amountUEToken * 10 **
   ↪  18).send({ from: app.web3.eth.defaultAccount }).then(async (receipt) => {
3      // look for a matching sell order
4      while ( i < app.state.sells.length && amountUEToken > 0) {
5          let price = app.state.sells[i].price;
6          let sellAmount = app.state.sells[i].amount;
7          if (sellAmount >= amountUEToken) {
8              await contract.methods.trade(app.state.sells[i].orderNo, amountUEToken, price,
                   ↪  1).send({ from: app.web3.eth.defaultAccount }).then(async (receipt) => {
9                  // Update the order book with the new buy orders
10             });
11             amountUEToken = 0;
12         } else {
13             await contract.methods.trade(app.state.sells[i].orderNo, sellAmount, price,
                   ↪  1).send({ from: app.web3.eth.defaultAccount }).then(async (receipt) => {
14                 // Update the order book with the new buy orders
15             });
16             amountUEToken -= sellAmount;
17         }
18         i++;
19     }
20 });
```

Listing 3.13: Executing a trade from the trading platform.

In listing 3.13, in each iteration of the while loop, we check if the amount of UEToken of the buy order can fill the amount of UEToken of the sell order. If the amount of UEToken of the sell order can be filled by the amount of UEToken of the

buy order, then we execute the trade by calling the `trade` function of the Orderbook smart contract. Otherwise, we partially fill the sell order by calling the `trade` the function of the Orderbook smart contract until the amount of UEToken of the buy order is filled, then we stop the while loop.

Here, we describe adding a new buy order to the order book. The process of adding a new sell order to the order book is similar to the process of adding a new buy order to the order book. The only difference is that we call the `addSell` function of the Orderbook smart contract instead of the `addBuy` function of the Orderbook smart contract. We change all the `buy` keywords to `sell` keywords in the code snippet in Listing 3.12 and Listing 3.13.

3.2.4 Scenario

In this section, we describe a scenario to make easier the understanding of how the trading platform interacts with the Orderbook smart contract. The scenario is illustrated in Figure 3.5. The sequence diagram shows the interaction between the Buyer (Alice), the Seller (Bob), the Platform, and the Orderbook smart contract while placing a buy order and executing a trade. The following steps describe the scenario as follows:

1. Alice, the buyer, initiates a buy order by calling the `addBuy` function on the platform. She specifies the amount of 100 UET and the buy price of 10 USDC per UEToken.

2. The platform receives the buy order from Alice and forwards the buy order to the Orderbook smart contract by calling the `approve` function of the USDC smart contract to approve the Orderbook smart contract to transfer the amount of 1000 USDC from Alice's account. This approval ensures that the Orderbook can transfer the tokens on behalf of Alice when a matching sell order is found. Once the approval is successful, the platform calls the `addBuy` function of the Orderbook smart contract to place the buy order in the order book.

3. The Orderbook smart contract adds the buy order to its buys list. It updates the internal state and emits the 'BuyAdded' event with the order details (order number, amount, price, and trader).

4. The Platform verifies if there are any matching sell orders in the order book. In this case, there are no immediate matches.

5. The platform calls the `watchOrderbook` function of the Orderbook smart contract to update the order book with the new buy order. It receives the event `BuyAdded` from the Orderbook smart contract and updates the order book with the new buy order to display the order book to Alice and Bob.

6. Bob, the seller, initiates a sell order by calling the `addSell` function on the platform. He specifies the amount of 100 UET and the selling price of 10 USDC per UEToken.

7. The platform receives the sell order from Bob and forwards the sell order to the Orderbook smart contract by calling the `approve` function of the UEToken smart contract to approve the Orderbook smart contract to transfer the amount of 100 UET from Bob's account. This approval ensures that the Orderbook can transfer the tokens on behalf of Bob when a matching buy order is found.

8. Once the approval is successful, the platform calls the `addSell` function of the Orderbook smart contract to place the sell order in the order book. It provides the amount of 100 UET and the selling price of 10 USDC per UEToken.

9. The Orderbook smart contract adds the sell order to its sales list. It updates the internal state and emits the `SellAdded` event with the order details (order number, amount, price, and trader).

10. The Platform checks for matching buy orders in the order book. In this case, there is a match with Alice's buy order.

11. The Platform initiates the trade by calling the `trade` function on the Orderbook smart contract. It provides the necessary parameters (order number, amount, price, and trade type).

12. The Orderbook smart contract verifies the trade conditions and executes the trade. It transfers the required tokens between the Buyer (Alice) and the Seller (Bob) using the `transferFrom` and `transfer` functions.

13. The Orderbook emits the `TradeAdd` event with the trade details (order number, amount, price, maker, and taker).

14. The order book updates its internal buys and sells lists by removing the executed orders.

15. The Platform refreshes the buys and sells lists for Alice and Bob.

The sequence diagram illustrates the flow of interactions between the different actors and the Orderbook smart contract. It shows the process of adding orders, verifying matches, executing trades, and updating the order lists.

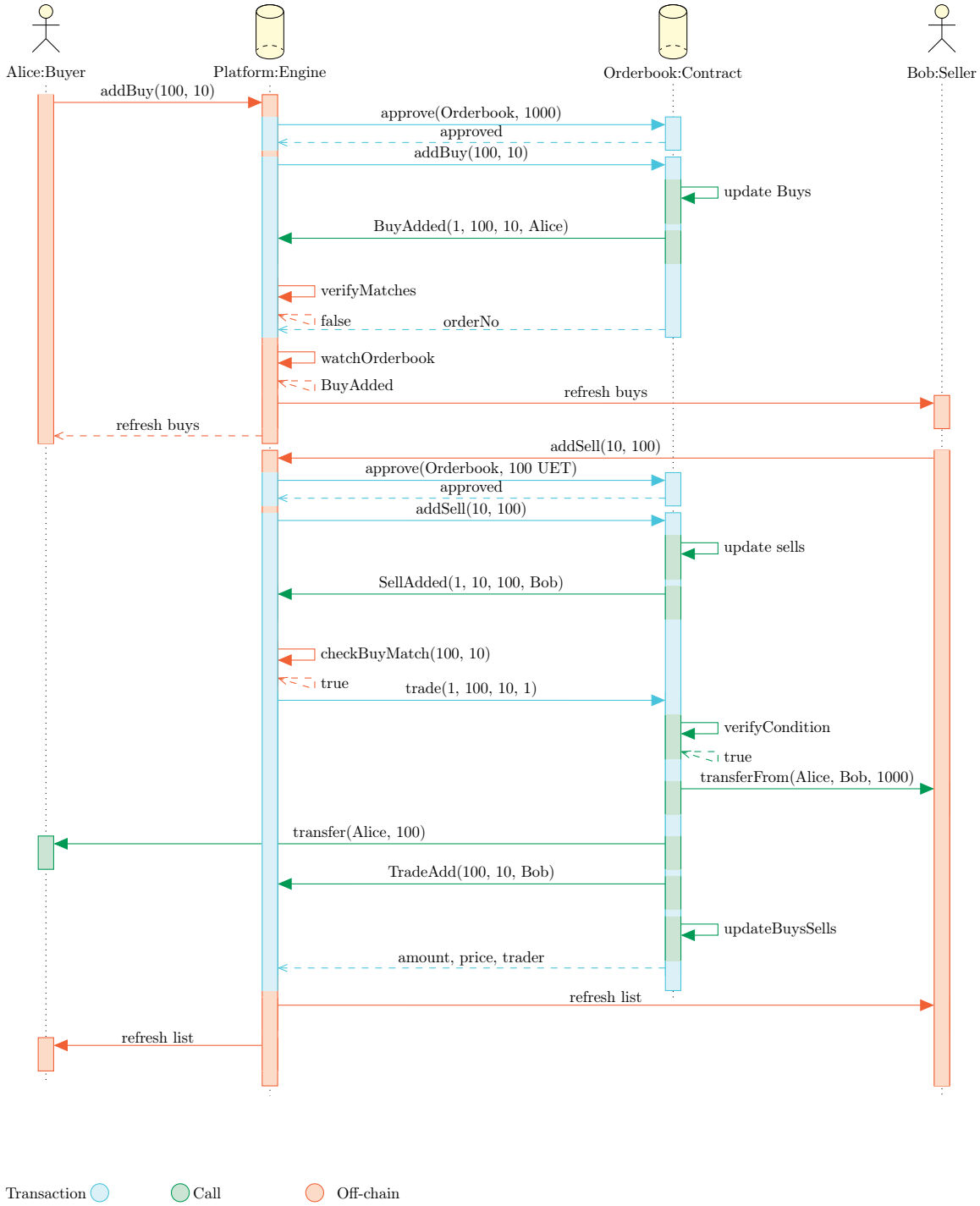

Figure 3.5: Sequence diagram of the Orderbook smart contract.

4 Discussion

In this chapter, we discuss the findings and limitations of our project, as well as propose some future work to improve the platform exchange and the tokens.

4.1 ERC777 token standard, an alternative solution to ERC20

In this project, the solution implemented is based on the ERC20 token standard. One alternative solution to the ERC20 token is the ERC777 token standard proposed by Jacques Dafflon, Jordi Baylina, and Thomas Shababi [24]. ERC777 is an advanced token standard that addresses specific limitations encountered in our platform exchange of two tokens. While the ERC20 token standard is widely adopted and supported by most token-based systems, there are certain limitations that can impact the efficiency and flexibility of our platform.

The ERC777 token standard introduces a more intuitive and efficient way to transfer tokens. It uses the `send` function, similar to the `transfer` function in the ERC20 token standard, but it facilitates users to understand and interact with the token contract. This transfer function can improve our platform exchange by allowing users to transfer tokens without worrying about the `approve` and `transferFrom` functions. Indeed, for example, if the recipient is a contract, users must first call the `approve` function, then the `transferFrom` function mechanism to transfer tokens. If a user forgets and only calls the `transfer`, the tokens will get stuck in the contract, and the user will not be able to retrieve them. There is no way to regain the tokens, and the user will lose them.

The ERC777 token standard also allows contracts and regular addresses to control or reject the tokens they send and receive. Using the `tokenToSend` and `tokensReceived` hooks, we can implement custom logic to validate and manage tokens transfer. These features can be helpful by providing more control and flexibility to our platform exchange. In addition, it ensures that only authorized and verified transactions are processed on our platform.

Finally, the ERC777 token standard also introduces one significant advantage over the ERC20 token standard: the ability to send tokens to a contract and notify the contract in a single transaction with the `tokensReceived` hook. This feature removes a two-step process like the `approve/transferFrom` required by the ERC20 token standard, making interaction with contracts more straightforward and efficient. In addition, this feature can be helpful by simplifying the interaction of our platform with external contracts, such as liquidity providers.

While the ERC777 token standard offers some different features and advantages over the ERC20 token standard, its adoption requires significant effort and time. We need to assess the impact of this change on our platform by assessing the compatibility and readiness of existing wallets and infrastructure to support the ERC777 token standard. Moreover, we need to update our platform to support the ERC777 token standard by updating the smart contracts and the front-end to accommodate the new features and functionalities.

4.2 Future Work: Liquidity and Automated Pricing

One of the main limitations of our platform exchange order book is the need for more liquidity functionality. Indeed, our exchange operates based on fixed prices and quantities for exchanging tokens. This means that users can only accept the price and quantity offered by the other counterparty. This can be a limitation for our platform, as it can discourage users from using it. However, a more dynamic and efficient solution would be to implement a liquidity mechanism that automatically determines the price of a token based on the market demand and supply[1] against another token. In traditional centralized exchanges, liquidity is often provided by market makers or specialized firms that provide liquidity to the exchange. However, in decentralized exchanges and platforms like ours, creating liquidity is more challenging, as there is no central authority to provide liquidity, and thus, it requires a different approach.

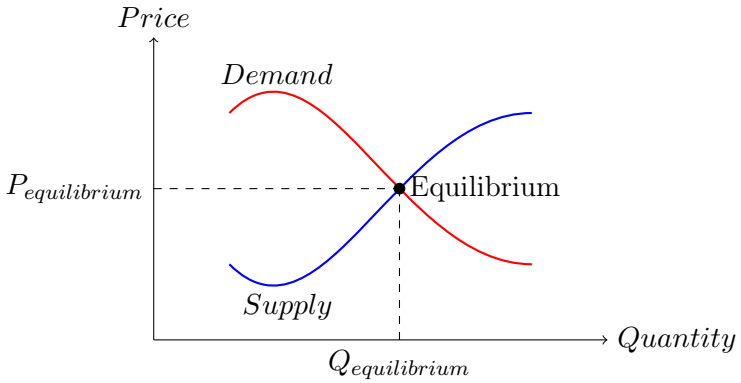

Figure 4.1: Supply and demand curves for computing the equilibrium point.

In figure 4.1, we illustrate that the equilibrium point between the supply and demand curves is at the intersection of the two curves. In our platform exchange, the supply of our UET token is initially set at its creation using the `initialSupply` parameter. This represents the total quantity of UET tokens available in the market. However, the demand for the token is determined by the interaction of users within the liquidity pool. A liquidity pool is a mechanism that enables users to provide liquidity by depositing their tokens into a smart contract.

Therefore, a possible solution to implement liquidity in our platform exchange is to create a liquidity pool. This smart contract holds a certain amount of our UET token and the other token. Traders can trade between these tokens by depositing them into the liquidity pool. On the other hand, liquidity providers supply the pool with their tokens and receive a portion of the transaction fees as a reward. These liquidity pools represent the share of the provider in the pool.

To have a better understanding of how liquidity pools work, we can take the example of using a liquidity provider like Uniswap [25]. Uniswap is a decentralized exchange protocol that allows users to swap between any two ERC20 tokens. Users create liquidity pools for various token pairs. To illustrate, we can create a liquidity

[1]The market demand and supply is the total quantity of a token that users are willing to buy or sell at a given price.

pool for our UET token against another, such as USD Coin (USDC). To do so, we need to deposit an equal value of UET tokens and USDC tokens into the liquidity pool. In return, we receive a liquidity pool token representing our share in the pool.

Once the liquidity pool is created, traders can then trade between UET tokens against USDC by interacting with the liquidity pool smart contract. Therefore, the pricing of the tokens is determined automatically within the liquidity pool based on a constant product formula, defined by Uniswap [26] as follows:

$$x * y = k \tag{4.1}$$

Where x and y are the quantities of the tokens in the liquidity pool, and k is a constant value. This formula considers the ratio of token balances in the liquidity pool. The automated pricing mechanism ensures the liquidity pool is always balanced by maintaining a balanced value between the two tokens for each trade.

By using a service like Uniswap or implementing our own liquidity pool mechanism, we can provide our platform exchange with a more dynamic and efficient solution for trading tokens for users. It allows users to trade with a seamless experience and access a large liquidity pool for various token pairs, e.g., UET/USDC, UET/ETH, UET/DAI, etc. Moreover, liquidity providers can earn a portion of the transaction fees as a reward based on the trading volume in the pool, which can incentivize them to supply liquidity and contribute to the growth of our platform exchange.

Establishing a liquidity pool mechanism requires significant careful planning and design. We need to assess factors, such as the initial supply of the tokens and incentives for liquidity providers, and we need to especially find a solution to attract traders to our platform exchange, which is crucial for the success of the liquidity pool. Furthermore, the integration with other decentralized exchanges and protocols can be a challenge, as it requires a significant effort and time to implement and test the integration.

5 Conclusion

The repository containing all the code sources of our project is available at the following link:

`https://gitlab.com/Skogarmadr/mse-pa`

The repository containing the code source of this report is available at the following link:

`https://github.com/crypt0log-hai/mse-pa-report`

This project has successfully met its objectives of exploring the potential of blockchain technology to mitigate the current payment systems limitations today by creating our own cryptocurrency for international settlements. We have developed two proofs of concept, including a token wallet and a secure exchange platform to trade tokens. Our new cryptocurrency is built using the ERC20 standard, and it incorporates critical features which allow us to create a robust and secure token smart contract providing fungibility and interoperability.

Our token wallet allows users to securely store and manage our and other ERC20 tokens, providing a convenient and user-friendly interface. By implementing features such as balance checking, token transfer, and token approval, we have created a reliable and efficient solution for managing our cryptocurrency and other cryptocurrencies using the Ethereum blockchain.

Through the development of a secure exchange platform, we allow users to trade their tokens with other users' tokens. Furthermore, implementing the ERC20 standard enables users to buy and sell token pairs at a determined price through an order book. Therefore, we have created a decentralized exchange ensuring fair and transparent trading among users, which valid the need to eliminate the trusted third party, such as a bank or a broker, enhancing transparency.

Finally, we have successfully achieved the objectives of this project by creating a new token ecosystem that meets the requirements for international settlements. Furthermore, we have demonstrated that our virtual currency complying with the ERC20 standard, can be traded to settle in a secure and transparent exchange platform between two parties.

There are still opportunities for future work to improve the platform exchange and the tokens on our project for enhancements and new features. One alternative is exploring other token standards, such as ERC777, to mitigate the risk of misusing the ERC20 standard and provide some additional features. Furthermore, we can also explore implementing a liquidity mechanism and expanding the platform exchange to support more tokens and token pairs.

Overall, the objectives for this project have been met, giving a solid insight into the development and implementation of a blockchain-based exchange platform. We are confident that our solution can be viewed as a foundation for future work in blockchain technology and maybe even be integrated into the financial sector.

List of Figures

List of Tables

List of sources

References

[1] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 05/23/2023).

[2] Daniel Drescher. *Blockchain Basics*. en. 1st ed. Berlin, Germany: APress, Mar. 2017.

[3] Andreas Antonopoulos. *Mastering Bitcoin*. 2nd ed. Sebastopol, CA: O'Reilly Media, June 2017.

[4] Jon Danielsson. *Global Financial Systems*. en. London, England: Pearson Education, Aug. 2013.

[5] Don Tapscott and Alex Tapscott. *Blockchain revolution*. Portfolio, June 2018.

[6] Andreas Antonopoulos and Gavin Wood. *Mastering Ethereum*. Sebastopol, CA: O'Reilly Media, Dec. 2018.

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

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

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

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

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

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

[13] 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).

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

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

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

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

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

[19] Facebook. *React*. 2023. URL: https://reactjs.org/ (visited on 05/23/2023).

[20] Truffle Suite. *Ganache CLI*. https://github.com/trufflesuite/ganache-cli-archive. 2023. (Visited on 05/23/2023).

[21] Metamask. *Metamask*. 2023. URL: https://metamask.io/ (visited on 05/23/2023).

[22] Metamask Community. *Not able to import custom tokens on MM from Ganache*. 2023. URL: https://community.metamask.io/t/not-able-to-import-custom-tokens-on-mm-from-ganache/22939 (visited on 05/23/2023).

[23] Centre Consortium. *USD Coin*. 2023. URL: https://www.centre.io/usdc (visited on 05/23/2023).

[24] Jacques Dafflon. *ERC777*. Nov. 2017. URL: https://eips.ethereum.org/EIPS/eip-777 (visited on 05/23/2023).

[25] Uniswap. *Uniswap Protocol*. 2018. URL: https://uniswap.org/ (visited on 05/23/2023).

[26] Uniswap. *Uniswap Formula*. 2023. URL: https://support.uniswap.org/hc/en-us/articles/8829880740109-What-is-a-liquidity-pool- (visited on 05/23/2023).