Tokenizing Mutual Funds on the Ethereum Blockchain

Benjamin Threader

A thesis submitted in partial fulfilment of the requirements for the degree of

MSc Computer Science

of

University College London

Department of Computer Science

University College London

2022

Abstract

This thesis investigates the viability of tokenizing an open-ended mutual fund on a blockchain through an implementation on the Ethereum blockchain using the Solidity programming language. This high level goal is split into separate sections, represented in this thesis by a chronological series of experiments.

The motivation is to create a new and innovative way of selling mutual fund shares with the hopes of increasing market participation, reducing transaction costs and increasing liquidity for clients. This research is fundamentally important in an environment where financial services firms seek to digitalise their products and determine new ways to make their offering stand out amongst competitors.

Experiments:

1. Know Your Customer (KYC). Assessing how best to implement customer identity verification on the blockchain so as to comply with the wider Anti-Money Laundering (AML) policy of the issuer.

2. Off-Chain Fund Implementation. Allowing customers to trade shares with the issuer as well as with other investors directly through the fund. Additionally, illustrating how an external exchange could use the contract to allow customers to trade with each-other. Finally, implementing functionality which facilitates investment in off-chain assets.

3. On-Chain Fund Implementation. Building a contract which executes a hard coded strategy consisting of investments in on-chain assets. Delivering fully automated liquidity management for clients and guaranteeing the immediate execution of buy and sell orders 24/7.

This thesis represents the following contributions to science:

1. Creation of a realistic digital representation of a fundamentally important investment vehicle. Open-ended mutual funds represented $71.1 trillion of wealth at the end of 2021 (Investment Company Institute, 2022). This thesis delivers a blue print which can be used by successive attempts at implementation whether they are proof of concept or production applications.

2. Detailed exploration of future concerns in the space, revealing key issues for regulators and fund managers. Thoughtful analysis of the limitations of the implementation proposed, revealing opportunities for further research.

Impact Statement

In a financial services sector accelerating down the path of a digital overhaul, this thesis provides a blueprint for mutual fund managers to automate large chunks of process and keep ahead of competitors. The future of the mutual fund is that of high customer liquidity, low fees, and full transparency for clients.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

This chapter presents an outline of this thesis, starting with the motivation behind the research area. We then clarify the objectives of the research and the experiments mentioned in the abstract. Finally, we end with an explanation of the structure of the thesis. The aim of this chapter is for the reader to understand the importance of the research area, the specific research objectives and the approach this author will be taking over the course of the thesis.

1.1 Background and Motivation

1.1.1 Smart Contracts, Blockchain and Tokenization

A high level definition of a smart contract is provided by Röscheisen et al. (1998) who define a smart contract as "a computer program or a transaction protocol which is intended to automatically execute, control or document legally relevant events and actions according to the terms of a contract or an agreement". Expanding on the point of automatic execution, the Solidity programming language website defines smart contracts as "programs that are executed inside a peer-to-peer network where nobody has special authority over the execution, and thus they allow to implement tokens of value, ownership, voting and other kinds of logics" (Solidity, 2022). We will follow this definition as we will be deploying our smart contracts on a special kind of peer-to-peer network; a blockchain. A blockchain is a database of transactions that is updated and shared across many computers on a network (Ethereum, 2022a). Blockchain technologies such as Ethereum allow us to deploy our smart contracts on a platform where we have resilient execution, resilient storage of associated data and also where we can integrate payments through digital currencies. We will be using the Ethereum blockchain throughout this thesis.

Tokenization is the process of transforming ownership and rights of particular assets (contract) into a

digital form (smart contract) (Baum, 2021). Once tokenized, an asset can be called token, a special form of smart contract. For example, a tokenization application could be digitising home ownership rights. Instead of signing physical papers or providing a written signature during the purchase process, a buyer and seller could simply fill in digital forms which they can then submit for verification. Rather than a person reading the form, a computer program could check if the inputs are valid and then transfer the digital representation of ownership to the buyer. Using a blockchain we ensure that no party can back out of their agreement and stop the execution of the contract, we also guarantee that the data involved in the transaction is stored in a resilient manner.

1.1.2 Mutual Funds

A mutual fund is a professionally managed investment fund that pools money from many investors to purchase securities. There are two types of mutual fund; open-ended and close-ended. Open ended funds can issue an unlimited number of shares, close ended funds offer a fixed amount of shares. Our research focuses on open ended funds which make up the majority of mutual funds (Investment Company Institute, 2022). However, much of our work will be easily extendable to close-ended funds.

It is currently legislated that open-ended fund managers must report the Net Asset Value (NAV) of their fund at given intervals and guarantee the opportunity for investors to buy or sell shares at the price of NAV per share[1]. Whilst rules differ depending on jurisdiction, many funds offer daily issuance and redemption.

Mutual funds are a popular asset class and the fund industry is competitive. Funds compete on a variety of different dimensions: performance, risk and cost, to name a few. We will now discuss the potential benefits of tokenizing mutual funds.

1.1.3 Tokenized Mutual Funds

Efficiency

Smart contracts automatically execute, whereas at present many separate but related processes must be completely synchronously by a fund manager and other agents to distribute shares of the fund. Take the example of performing the issuance and redemption process in Figure 1.1.

[1]For example see the UK (HM Revenue & Customs, 2019)

Figure 1.1: Example Issuance and Redemption Process

1. Buyer / seller places an order
2. Fund analyses the order book at the end of the day
3. Fund performs necessary actions to satisfy orders
    - If there are more sells than buys, sell positions to gain the cash to fulfill the orders
    - If there are more buys than sells, use the extra cash to invest in assets
    - Send shares or money directly to order placers or custodians
    - Adjust ownership data in the system

The best outcome for tokenization would be to put this entire process into one computer program. However, even if a fraction of these processes are codified, the benefits could be significant. According to Broadridge, the most significant opportunity for efficiency gains in the banking industry is in trade processing, which costs industry participants $17 billion to $24 billion per year (Broadridge, 2015). By accepting digital currency, a fund could eliminate the need for payment providing intermediaries like banks when selling shares to customers, reducing costs and increasing profits for the fund. Further, if ownership rights can be digitised and a fund had enough digital currency to satisfy sell orders, a fund could allow for issuance and redemption 24/7 through code. This would boost client experience and could allow shares of the fund to trade at a premium.

Additionally, by removing the administrative workload of performing these processes, asset managers could increase their focus on strategic tasks such alpha generation or even creating new funds. Such changes no doubt would increase job satisfaction for employees, allowing the fund to attract and retain quality talent.

To take automation gains to the extreme, consider an economy where all assets are tokenized. In this arrangement, rather than relying on the fund manager to withdraw cash from the contract for the purposes of investing, the fund smart contract could itself invest directly into assets. This could reduce costs by removing the need for intermediaries such as investment banks to access the financial markets. Financial reporting which is another time consuming process could simply become self-service. The smart contract could immediately calculate it's holdings, and therefore the NAV 24/7 because all the data is readily available on the blockchain.

Decentralization and Transparency

Underlying blockchain technology ensures that in the event of a fund server outage or hard drive corruption, the fund can still function due to the presence of other nodes in the network which have their own copy of the data and can process transactions. Additionally, due to the nature of the blockchain implementations, once smart contracts are deployed their code cannot be altered. The fund manager could even release this

code to investors via a copyrighted white paper.

1.2 Research Objectives

The main objective for this research is to implement two realistic, fully functioning mutual fund tokens; one for a fund with a portfolio of off-chain assets, one for a fund with a portfolio of on-chain assets. To do this we perform three experiments, each of which will deliver a well-tested codebase. Experiments will be carried out chronologically and may leverage code from previous experiments. For example, KYC functionality obtained from the first experiment will be utilised in the second and third.

1.3 Research Experiments

1. Know Your Customer (KYC). The first experiment has the objective of making sure our contract is suitably permissioned to operate within a regulated environment. Regulated funds are our interest and it is important to fund managers that not just anyone can trade shares of these fund. We build out infrastructure using a zero-trust principle, doing our best to ensure that the compromise of one fund-owned account does not destroy the contract. To do this we use a role based approach creating the roles of "admin", "verifier" and "verified". Our infrastructure of roles and permissions is in stark contrast to typical permission-less tokens on the Ethereum blockchain where investors are anonymous. The importance of this work is twofold. Firstly, by completing this experiment we show that Anti-Money Laundering (AML) features can be put into tokens, which is of interest to regulators. Secondly, we lay a basis to achieve our research objective of implementing realistic mutual fund tokens as these funds must be permissioned. Subsequent experiments will make extensive use of this work.

2. Off-Chain Fund Implementation. This experiment implements a sophisticated automated mechanism through which buyers can buy shares 24/7, and sellers can queue orders that all subsequent buy orders will execute. Finally, in the situation that there are outstanding sell orders at the close of business, the fund manager simply presses a button to close the orders. All orders set price using NAV per share, with NAV value being uploaded to the contract by the fund manager. To allow for off-chain investments we give agents of the fund with the "accountant" role permission to withdraw cash. Without the necessary cash the contract cannot close the sell orders at the close of business, thus we also provide functionality for accountants to check how much cash is needed and then move this into the contract.

3. On-Chain Fund Implementation. Leveraging work from the second experiment we then use a scheme of investments and weights to implement a fund which has issuance and redemption fully automated. The only intervention required by the fund manager other than managing clients is to routinely call a rebalance function during periods of order inactivity. Unlike the previous experiment which required

the fund manager to top the cash up, the smart contract will create the necessary cash by liquidating investments. This contract shows what is theoretically possible for a mutual fund which finds itself within a tokenized economy.

1.4 Scientific Contributions

This research contributes in the following ways:

1. Creation of a realistic digital representation of a fundamentally important investment vehicle. Whilst certainly not the first tokenization project for a mutual fund[2], to the best of the authors knowledge it is the first publicly viewable copyright-free one. As such this is a landmark contribution to the field. Furthermore, rather than implementing one representation, we provide two. The first representation, our off-chain invested fund, is pragmatic and serves to aid financial institutions beginning a gradual transition to tokenization. With emerging technologies it is often easy to focus too much on the long-term potential rather than immediate opportunities for implementation. Whilst our on-chain invested fund is certainly exciting and displays the full potential benefits of tokenization, it relies on many other components already being in place. Thus, the inclusion of two representations allows us to display both the quick wins of tokenization, and the potential big gains further down the line. This is a massive contribution to an industry which itself is unsure of what the future road map may look like, even more so when certain jurisdictions, such as the UK, don't yet allow for blockchain traded funds (Flood, 2022).

2. Detailed exploration of future concerns in the space, revealing key issues for regulators and fund managers. On top of providing thought provoking implementations, through trial and error our work reveals a wide variety of areas for discussion and future work. The industry is at a cross roads, and it is not clear at present what work needs to be done and by who. This work reveals challenges for implementations, important considerations for funds and also security issues and how they may be remediated. This is immensely important for funds who want to avoid making reputation damaging mistakes in their attempts at tokenizing their funds. Additionally the work is useful for regulators who will want to protect consumers from these mistakes by mitigating risks through legislation. In the UK case, the regulator could use the work to start making rules before blockchain traded funds are even allowed, protecting the consumer from the outset.

[2]See Noonan (2021) or Enzyme (2022)

1.5 Thesis structure

After covering the context of the thesis in the background and literature review chapter, there will be a chapter for each experiment. Each experiment chapter will begin with an introduction to explain the background of the experiment in more depth. There is then a tests section where we summarize what code was created and then detail the testing we performed for this code. Experiments will then end with a discussion that summarises the achievements of the chapter and then addresses issues and themes uncovered by the completion of the experiment. For example, after finishing the second experiment which implements an off-chain fund, we discuss the issues of transparency that come with manual NAV reporting in our contract. Finally, the thesis ends with a conclusion chapter that summarizes the research completed and the associated contributions to science, we then discuss how other academics or researchers can build on the work completed.

In addition to the work in this document, this author has made a short presentation covering the work completed which can be found on YouTube. The full source code for the project can be found on GitHub.

Chapter 2

Background and Literature Review

In this chapter we seek to comprehensively analyse related work, looking for relevant information to make the code implementations in our experiments as realistic as possible and avoid any potential pitfalls. Further, we seek to confirm the importance of the area of research and identify any themes to analyse when we conduct our experiments. Finally, we introduce the reader to important technical details underlining our code such as the programming language that the experiments will use. By the end of this chapter the reader should have a strong understanding of the context of the experiments and the foundational knowledge necessary to understand the work performed to complete them.

2.1 Relevant Developments in the Blockchain Space

2.1.1 The Ethereum Blockchain

As mentioned in the introduction, this thesis will use the Ethereum blockchain. Ethereum is the primary home for decentralized applications including decentralized finance (Amazon, 2022). Initially released in 2015, the maturity and open source nautre of Ethereum has made it immensely popular. This in turn has created a positive feedback loop by attracting developers who build applications which bring more users.

Ethereum stands out for the faith that large financial institutions have in it. ConsenSys is a software technology company that builds software services and applications that operate on the Ethereum blockchain. Consensys has attracted investments from major financial institutions such as Marshall Wace, Third Point, JPMorgan, Mastercard, and UBS (Ehrlich, 2021a). Ethereum is also a popular choice for building platforms outside of the main Ethereum network. For example Hyperledger Besu[1] is an enterprise-grade Ethereum

[1] https://www.hyperledger.org/use/besu

codebase which inherited code from a JPMorgan blockchain project called Quorum.

To send transactions on the Ethereum blockchain a sender must sign transactions and spend Ether, the native digital currency of Ethereum. Transaction senders are identified by their 42 hexadecimal character address. The presence of this native currency allows us to easily integrate payment into our code. However, the currency is highly volatile compared to traditional FIAT currencies (Statista, 2022). As an alternative, so called "stablecoins" have sought to achieve the ease of use inherent to digital currencies for smart contracts whilst keeping an acceptable level of volatility. For example, USD coin (USDC) is a stablecoin offered by Circle[2]. The coin operates by users sending USD to Circle's bank account, these users then receive freshly minted digital coins in exchange. USD reserves are attested to monthly but not audited by Grant Thornton[3]. Whilst this seems to be a step in the right direction, there is still a void of regulation in the area (as hinted to by the lack of audit). The UK government has confirmed its intention to bring certain stablecoins, where used as a means of payment, into the regulatory perimeter (HM Treasury, 2022). However, until then we should be sceptical of stablecoins. A demonstration of what can go wrong is the USDT scandal in which a stablecoin issuer called Tether was revealed to have broken it's promise of backing all USD tokens it offered with traditional currency. It transpired in April 2019 that the Tether didn't have reserves equal to the issued amount and instead it's parent company (Bitfinex) had siphoned funds to cover an alleged $850 million in losses (Ehrlich, 2021b). With a lack of regulated stablecoins, our contracts will instead use the simplest currency to integrate with the Ethereum blockchain; Ether. Smart contracts use Wei which is the smallest denomination of Ether; one Ether is equal to 10^{18} Wei.

2.1.2 A Standard Interface for a Token: EIP-20

In a tokenized economy, customers will want to have consistent entry points for token contracts, and developers will be used to standard practices for the benefits of safety and security; this is what EIP-20 (Vogelsteller and Buterin, 2015) sets out to do. By creating a standardised interface for all tokens without significantly restricting implementations, EIP-20 creates a sensible blue print for tokenizing an asset.

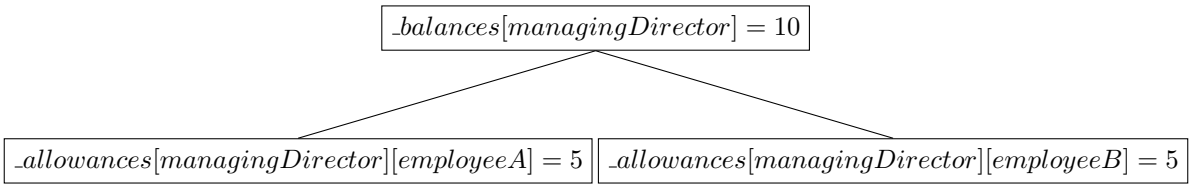
Whilst many parts of EIP-20 are self explanatory, the approval and allowance mechanisms are arguably more complicated. The basic premise of the allowances mechanism is that of hierarchical ownership. To give a simple context relevant example, consider the case where the managing director of a wealth manager wants to acquire 10 tokens. The managing director then wants their two employees, A and B, to each have the ability to trade 5 of those tokens without owning them directly. We represent this graphically in Figure 2.1 using OpenZeppelin state variable names from their ERC20 implementation[4].

[2]https://www.circle.com/en/usdc

[3]https://www.centre.io/usdc-transparency

[4]https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

Figure 2.1: Example Usage of the ERC20 Allowance Mechanism



The benefits of the approve mechanism over having the managing director simply transfer the tokens is that we can increase or decrease approved allowances, whereas we cannot undo the transfer of tokens. If employee A gets fired and their wallet still has 5 tokens sent via transfer, the managing director will have to request the fund manager to burn these shares and reissue them to their address. This is opposed to the managing director setting employee A's allowance to zero immediately. This allows owners to delegate the management of their funds with the fail safe of revoking access when needed. In our case, where large institutions are potential buyers of fund shares, this seems to be a logical addition.

Third parties can also use these allowances to facilitate multi-contract interactions between users, such as swaps. An important example of a third party using the approve mechanism is the Decentralized Exchange (DEX). Instead of giving custody of their wallet to the exchange administrators, users can give an allowance to the DEX. Thus users keep the tokens and can revoke the DEXs allowance whenever they like (although they would be unable to use the exchange). When it comes to performing transactions on the exchange between two parties, the DEX uses the allowances it has been given by the buyer and the seller to perform exchanges. This means that users can exchange tokens without having to transfer tokens to the exchange.

Our ERC20 implementation will be made inheriting the Solidity interface code from OpenZeppelin[5] and using code from their ERC20 implementation mentioned previously.

2.1.3 A Model for a Tokenized Equity: EIP-884

EIP-884 (Sag, 2018) is a draft for the Delaware State Commission to create a permissioned digital equity representation. In EIP-884 the set of verified wallets (i.e. customers) is maintained within the smart contract. Also, client identifying data (henceforth CID) is hashed and also put onto the smart contract. Thus the function for adding a verified customer to the smart contract becomes a function with two arguments; the address being verified and a hash. The reason for this functionality is that an issuer can readily produce an up to date list of the names and addresses of all shareholders. The contract itself acts as an immutable database of the customers. On top of managing the verification process, EIP-884 overlaps with the token interface in that it has a function "burnAndReissue". We will include this and believe it is useful for two reasons. Firstly, it allows a customer to move funds from a compromised account (e.g. key forgotten or

[5]https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol

account hacked) to a new account. Secondly, it allows the fund to seize assets from a client and moving them into an account owned by the fund manager. This infrastructure allows the fund to comply with the regulator in situations such as asset freezing due to sanctions.

A criticism of EIP-884 is that given the hash function is maintained off-chain, the method is not self contained. Thus, in of itself the hashed information on the smart contract is useless without the cooperation of the issuer. This begs the question whether it would be easier, and more secure to just maintain this CID off-chain. Under such an implementation the preparation of a list of shareholders would involve an on-chain query of the list of addresses, then an off-chain query, matching all those wallets to CID. The regulator may however prefer the hash methodology, as it provides evidence that the information definitely exists and simply needs to be decrypted. This, whilst not comprehensively satisfactory, gives some confidence that the issuer has some degree of KYC/AML in place. Additionally, the on-chain CID methodology could be implemented in a way such that the regulator could somehow decrypt the information themselves. This does of course pose privacy concerns.

2.2 Relevant Developments in the Fund Administration Space

2.2.1 Automated Investing

Investment funds have been using computers to identify and execute trades for years, with electronic and algorithmic trading now mainstream. However, few funds it seems are willing to completely take humans out of the investing process, and such activity has seemingly been the reservation of pioneers. In 2016 Ben Goertzel launched a hedge fund called Aidyia which ran on an automated system trading US equities. The fund was built in a way such that it required no human intervention, as Goertzel put it "if we all die, it would keep trading" (Metz, 2016). Perhaps less extreme is the proliferation of so called robo-advisers which are becoming "ubiquitous" in the financial advisory space (D'Acunto et al., 2019). Such technology simplifies and automates wealth management for investors (Jung et al., 2018). These developments show a desire for automation and cost reduction from both wealth managers and clients, and whilst orientated more towards the investment thesis of the fund rather than the distribution of shares, provide evidence that technology represents a strong area of competition in the industry.

2.2.2 Tokenized Mutual Funds

Those involved in the management of illiquid assets have been interested in tokenization for some time as they have the most to gain from the automation and transparency that tokenization offers (Laurent et al., 2018). This is due to the complexity of property rights, the slow speed of transactions and the high costs of trading (Laurent et al., 2018; Baum, 2021).

Industry interest in blockchain traded funds has culminated with the cited "breakthrough" coming in the form of Franklin Templetons OnChain US Government Money Fund which launched on 4 June 2021 (Noonan, 2021). The success of the launch is questionable given the astronomical increase in expenses, with a gross expense ratio currently at 8.76% (Franklin Templeton, 2022b) versus the non-blockchain equivalent which is sat at 0.52% (Franklin Templeton, 2022a). However, it does demonstrate (at least in the US) that regulators are willing to allow fund managers to try and improve their products using blockchain. The Investment Association, which is the asset management industry trade body in the UK, are currently lobbying the legislator to approve the use of blockchain technology for fund management (Flood, 2022). Interest in tokenized funds is also coming from those not directly involved in the industry but serve to benefit from a tokenization of funds. For example, tokenization in the real estate space is currently being held back by a lack of tokenization in lower level components such as funds and debt (Baum, 2021). According to Baum (2021), progress in the fund space could provide credibility for tokenized real estate applications.

Other action in the fund space has been in non-regulated areas. Enzyme is a decentralized asset management infrastructure built on Ethereum (Enzyme, 2022). Using Enzyme "Smart Vaults", individuals and communities can "build, scale and monetise investment (or execution) strategies that employ the newest innovations in decentralized finance". Rather than writing the code from scratch, Enzyme provides three templates for funds. Users then customize their fund by setting a variety of parameters; for example, users may limit the wallets able to make deposits in the fund. Enzyme funds are limited to investing in a finite list of on-chain assets decided by it's "Council". As a protocol, Enzyme forces developers to follow a predefined formula and charges usage fees. It seems unlikely that large institutions will opt to use this protocol before they have even had a shot at building their own, especially when they need to pay to use it and it limits what investments they can make. As such, we will make our own implementation from scratch and there-in identify potential innovations and important points of consideration for fund managers. Fund managers can then use these findings and information about the Enzyme protocol to make an informed choice as to what an on-chain fund may involve.

2.3 Writing, Testing and Running Code

Before we get started with our experiments, it's important to explain the technology we will be using to implement the smart contracts.

2.3.1 Solidity

There are two main programming languages being used to write smart contracts on the Ethereum blockchain. The first is Solidity, which is a statically-typed curly-braces programming language designed for developing smart contracts that run on Ethereum (Solidity, 2022). Then there is Vyper, a "pythonic" language built

with the aim of improving Solidity. Vyper aims to achieve three goals: security, simplicity (with respect to the language and the compiler) and audit-ability (Vyper, 2022). To do this, it opts to take a sub-set of Solidity features, for example choosing not to include contract inheritance.

This thesis uses Solidity due to the fact that - as the first ever language specifically designed to write smart contracts - it has a large community, great documentation, and is relatively simple yet has OOP features that are preferred by this author. To give the reader an idea of what code is being written throughout our work we will provide code snippets in the form of algorithms. These will be formatted with Solidity syntax, giving the reader strong intuition as to what the code is doing.

2.3.2 Style

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec) and can be found in the Solidity documentation[6]. We will attempt to make use of this. We will also use Solhint[7] to lint our code, making it consistent, for example by enforcing a maximum column width of 79. Finally, we adopt the approach of naming all state variables as-well as private and internal functions with an underscore at the start.

2.3.3 UML

To present our codebase succinctly, we will use UML class diagrams throughout. These give the reader an indication of the functionality of our contracts without having to read code, and also serves as documentation for the author. These diagrams will deviate from traditional UML due to the specificities of Solidity:

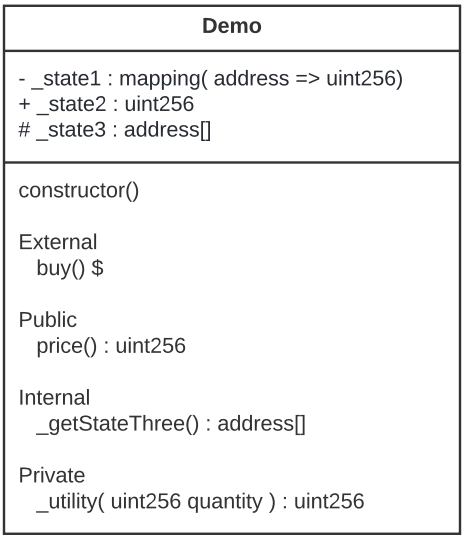
- Solidity has Events and Modifiers, which are different from functions and do not have user defined visibility. We still detail them as they are important documentation for our contract.
- Solidity has payable functions which can take money sent by a user in a transaction. These are marked by the "payable" modifier. We denote these functions with a $ after the function specification.
- Solidity has a visibility category that traditional UML doesn't have: "external". This visibility is used to define functions that can only be called from outside the contract; not by functions within the contract or functions in contracts which inherit the original contract.

Incorporating these specificities, we list state variables (with traditional visibility markers), constructors, events, modifiers, and finally functions by their visibility (external, public, internal, private), marking any function with a $ if it is payable. In Figure 2.2 we provide an example of our UML style. We use Solidity

[6]https://docs.soliditylang.org/en/latest/natspec-format.html
[7]https://protofire.github.io/solhint/

built-in types.

Figure 2.2: Demonstration of Modified UML for Solidity



2.3.4 Forge

To compile and test our code we will use the Forge toolkit[8] from the Foundry tool-chain due to its speed and modularity. Forge allows us for comprehensive testing of our smart contract, for example by changing the transaction sender, sending money to addresses or changing the money sent with transactions. As detailed in the introduction, each experiment will have a section on testing and we will use Forge to perform these tests. In the end, 40% of the code in our work (around 3000 lines of Solidity) was in files dedicated to testing.

2.3.5 Version Control

Finally, as mentioned in the introduction, the work in this thesis can be found in the GitHub repository here. Throughout the work done in our experiments we will reference the root of this directory, for example stating that a certain file may be found in src/folder/, this is a reference to a relative path from the GitHub repository to the file. A high level overview of this repository is that the contracts are found in the "src" folder and the corresponding tests are under the same folder names within the folder "test".

We also made use of GitHub Actions to ensure that all of our new code passed testing and linting before being integrated onto the main branch of the repository.

[8] https://github.com/foundry-rs/foundry/tree/master/forge

Chapter 3

Know Your Customer

In this chapter we build a mechanism through which the fund manager can control who can interact with each area of the contract. We also build a framework through which the fund manager can govern its own employees with varying roles and permissions to allow the fund manager to delegate fund administration, whilst ensuring the contract is robust to rogue agents.

3.1 Introduction

Anonymous, permission-free trading of our fund would be unacceptable to regulators. On the Ethereum blockchain, people are identified by their 42 hexadecimal character address. We need to build a gateway through which only certain addresses can perform certain interactions with our smart contract. Such infrastructure will be the basis of our fund contracts, for example restricting the functionality of purchasing shares to verified addresses.

3.2 Verifying Customers

3.2.1 Maintaining the Set of Verified Customers

It makes sense to store the on-chain set of verified customers in a hash map: we simply index the address to see if it's verified, which is a constant time operation. To add or remove verified customers, the fund would need employees who have the role of verifying addresses. These employees would be identified by the addresses from which they are interacting with the contract. We represent whether an address is a verifier using a mapping of addresses to bools.

3.2.2 Maintaining the Verifiers

An option to manage the verifiers would be to set some verifiers when the contract is invoked, then to allow the verifiers to remove each other, however it seems to be a significant and faulty assumption that those employees that manage customers are also always in a position to manage their fellow employees. The two roles are very different. It would be preferable to have another role called an admin which manages employees, restricting the scope of the verifiers to the customers.

3.2.3 Abstract Class

Before we go further and start implementing this admin role we implement an abstract contract AbstractVerify.sol with a virtual modifier "onlyAdmin". Future implementations will simply override this modifier and implement a method through which we determine the sender is an admin. Regardless of the implementation of the admin role, the functionality of the verifiers doesn't change, therefore this abstract contract can implement a variety of methods without us needing to worry about future inheritance.

We display the contract UML for AbstractVerify.sol in Figure 3.1. This contract takes heavily from EIP-884 in terms of functions and events. It's worth now detailing the "indexed" key word for events in Ethereum. Indexed parameters for logged events allow users to search for events using the indexed parameters as filters; Solidity allows for three of these. Any other fields which are not indexed are counted as "data". One point of note for readers is the presence of the "_balances" internal variable. It is detailed in EIP-884 that a contract shouldn't allow the removal of verification for a client with shares, therefore we needed to think ahead at this stage to represent share ownership, to do this we used a mapping pointing an address to an integer number of shares; "_balances[customer]" returning the number of shares owned by the customer. The inheritance of this contract in subsequent implementations can be found in the full UML diagram for the experiment in Figure 3.2.

Figure 3.1: AbstractVerify UML

3.2.4 Admin Implementation

One simple solution would be to add one admin, who has the sole ability to add and remove verifiers. This is implemented in SimpleVerfiy.sol. However, a whole number of things could go wrong with this implementation. Two obvious situations; (1) admin forgets the key they need to use their account, (2) the account gets hacked. In both these scenarios the ability to effectively administrate the fund would be compromised. Therefore it would be preferable to have more than one admin.

We then have the issue of managing the admins. If we think about the principles for the public blockchain in the first place we arrive at a better implementation which decentralizes the admin role. In ComplexVerfiy.sol the address which instantiates the contract becomes the first admin. In order to remove or add an admin, a successful election must take place, in which the majority of admins vote in favour of the decision.

We implement an election contract with the functionality to collect votes. This contract also has a state variable to track which addresses have voted, thus ensuring no address makes more than one vote in the same election; this contract can be found in Algorithm 3.1.

Algorithm 3.1 Election.sol

```
contract Election {

    /// ------------------------------
    ///         State
    /// ------------------------------

    uint256 public _votes;
    mapping(address => bool) public _hasVoted;

    /// ------------------------------
    ///         External
    /// ------------------------------

    /**
     *  @param voterAddr The address voting on the election
     */
    function vote(address voterAddr) external {
        require(
            _hasVoted[voterAddr] == false,
            "Election: Address has already voted"
        );

        _votes += 1;
        _hasVoted[voterAddr] = true;
    }

    /**
     * @param voterAddr The address voting on the election
     */
    function removeVote(address voterAddr) external {
        require(
            _hasVoted[voterAddr] == true,
            "Election: Address has not voted"
        );

        _votes -= 1;
        _hasVoted[voterAddr] = false;
    }
}
```

The address which instantiates the contract becomes the first admin. Admins after this have the following life cycle, consider the example for an address, A:

1. Existing admin votes to add A as an admin - new election created with one vote
2. A receives enough votes to become elected, A added as admin, election deleted
3. Admin votes to remove A - new election created with one vote
4. A receives enough votes to be removed, A removed, election deleted, any votes A made to elections which are still live removed

The last point is an important part of our contract. Given all elections reference the constantly updating figure of the total number of admins to quantify the majority, if we remove an admin without removing all their votes, all elections which they have voted on will become biased. Furthermore, when we consider that the admin has been removed for reasons such as their address being hacked, it would be prudent to remove all the votes because their votes may have been made in bad faith.

To ensure we can remove all the votes of an address, we need to keep track of all the addresses which have live elections so we can traverse them and iteratively remove the vote of the removed admin (if it exists). This adds a bit of complexity as we must adjust this array when elections are added or removed. To remove entries efficiently, we move swap the data item at the point of deletion and the end of the array, then pop the array. This gives us O(1) deletion time.

To give a quick idea of ComplexVerify.sol we show the functionality of voting to add an address as an admin to the contract in Algorithm 3.2.

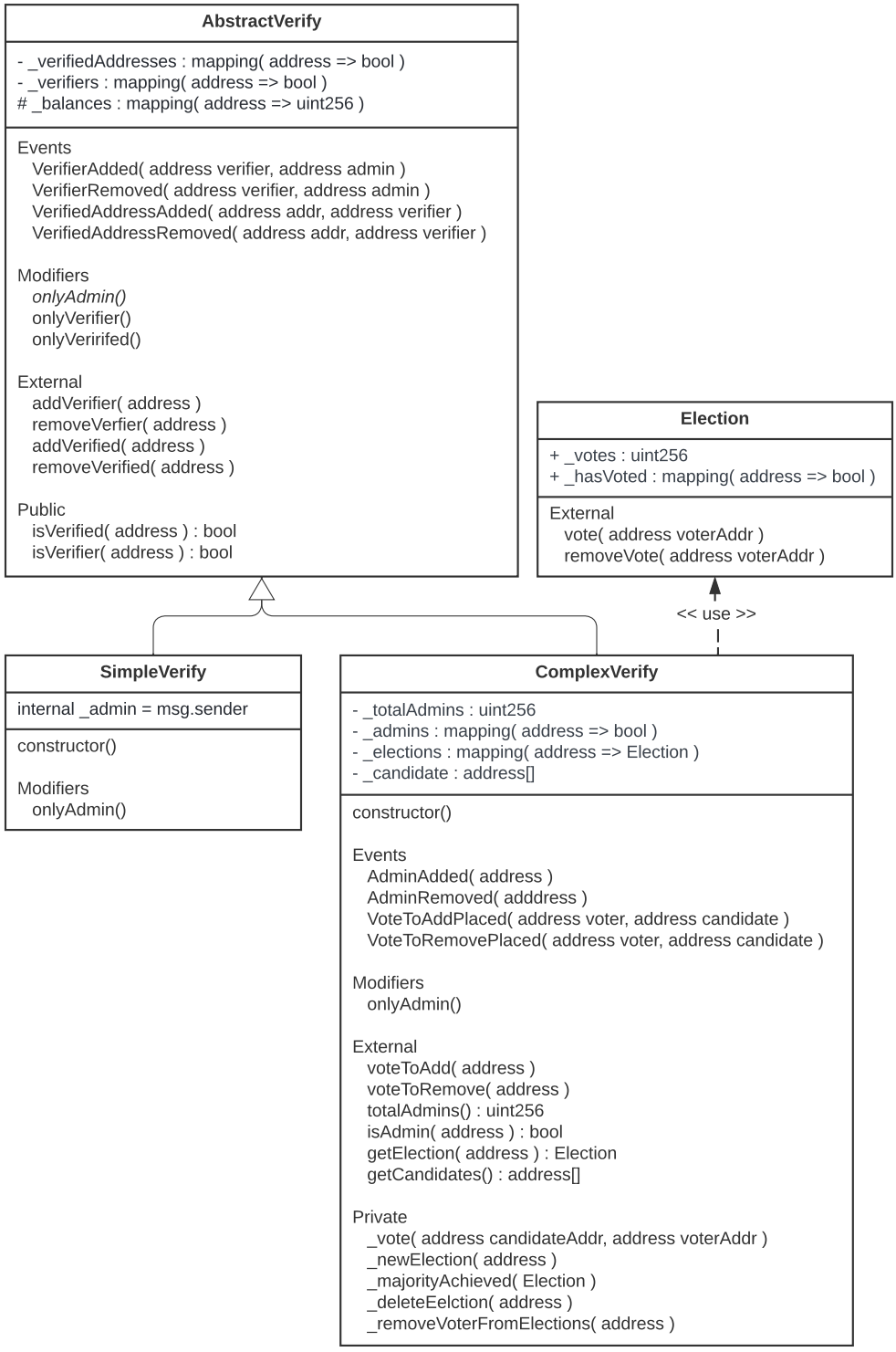
Algorithm 3.2 voteToAdd in ComplexVerify.sol

```
/**
 * @param candidateAddr The non-admin address we want to vote to add
 */
function voteToAdd(address candidateAddr) external onlyAdmin {
    require(
        _admins[candidateAddr] != true,
        "Verify: this address has already been added as an admin"
    );

    _vote({candidateAddr: candidateAddr, voterAddr: msg.sender});
    emit VoteToAddPlaced({voter : msg.sender, candidate: candidateAddr});

    if (_majorityAchieved(_elections[candidateAddr])) {
        _admins[candidateAddr] = true;
        _totalAdmins += 1;
        _deleteElection(candidateAddr);
        emit AdminAdded(candidateAddr);
    }
}
```

Given that admins are now managed in a sensible way, we can give all admins the ability to manage verifiers freely. In the worst case scenario where an admins account gets hacked and they remove all the verifiers, the other admins can vote to remove them. We can either revert all transactions from the point of contamination or use the logs to undo the changes made by the contaminated account.

We detail the UML for the full implementation in Figure 3.2. On top of the events in AbstractVerify.sol, we also add events relating to the installation or removal of admins.

Figure 3.2: UML for Full KYC Codebase



3.3 Tests

The work in this chapter amounted to one abstract and three non-abstract contracts:

```
./
└── src
    └── kyc
        ├── AbstractVerify.sol
        ├── ComplexVerify.sol
        ├── Election.sol
        └── SimpleVerify.sol
```

We test all non-abstract contracts and produce the following test files:

```
./
└── test
    └── kyc
        ├── ComplexVerify.t.sol
        ├── Election.t.sol
        └── SimpleVerify.t.sol
```

As the first section on testing, we will go into detail on our work to familiarise the user with the Forge workflow. We did all the testing of adding and removing verifiers and verified clients in SimpleVerify.t.sol, with ComplexVerify.t.sol exclusively covering the adding and removing of admins. This was logical as both contracts shared the same code for all functionality involving verifiers and verified clients.

To provide some examples we focus on ComplexVerify.t.sol. Throughout our testing we make use of a file GenericTest.sol, which provides us with three test accounts: acc1 (address 11), acc2 (address 12) and acc3 (address 13). These accounts were needed to perform role based testing.

Tests for the verification contracts began with us impersonating acc1 via the vm.prank(account) functionality in Forge to instantiate the contract, thus making acc1 the first admin. In Algorithm 3.3 we use acc1, who is an admin, to test that when they vote to add another account as admin (acc2), the address is immediately instantiated as an admin due to majority being achieved ($1 > 1/2$). To check this happened successfully we check that the expected event was emitted via the vm.expectEmit functionality in Forge. We also call functions to check the correct changes were made to state, using the Forge assertion library.

The expect emit functionality works by the user setting some parameters in vm.expectEmit to detail what the emissions should look like in terms of indexed attributes, data present and the address which emits the event. In our case we are expecting a "VoteToAddPlaced" event, which consists of two indexed attributed (out of three) and no data. This corresponds to (true, true, false, false), with the first three entries indicating the presence of indexed attributes and the last data. The user emits the emission expected in the code within the test contract. Forge will then compare the event emitted by the following transaction (the call by acc1 to add acc2 as an admin) with both the parameters in expectEmit and the previous provided emission. If

the details don't match up an exception is thrown and the test fails.

Algorithm 3.3 testOneVoteAdd in ComplexVerify.t.sol

```
function testOneVoteAdd() public {
    vm.startPrank(acc1);
    vm.expectEmit(
        true, true, false, false,
        address(verificationContract)
    );
    emit VoteToAddPlaced(acc1, acc2);
    verificationContract.voteToAdd(acc2);

    assertTrue(
        verificationContract.isAdmin(acc2),
        "acc1 not added to admins"
    );
    assertTrue(
        verificationContract.totalAdmins() == 2,
        "totalAdmins not updated"
    );
    vm.stopPrank();
}
```

Another feature Forge offers is vm.expectRevert. This allows us to both test that an exception is thrown and also that the exception message is the one expected. An example of our usage of this is when testing that a double vote fails in Election.t.sol, this is displayed in Algorithm 3.4.

Algorithm 3.4 testDoubleVoteFail in Election.t.sol

```
function testDoubleVoteFail() public {
    election.vote(acc1);
    vm.expectRevert(bytes("Election: Address has already voted"));
    election.vote(acc1);
    assertTrue(
        election._votes() == 1,
        "Total votes not correct"
    );
}
```

3.4 Discussion

Our contract is very delicate, as the functionality provided to a permissioned user can quickly allow them to remove the functionality for other users, there-in rendering the contract inoperable. Therefore, it is essential not to take short cuts, and to implement sensible mechanisms such as majority voting for adding or removing the most permissioned users. This work is fundamentally important as a foundation for further experiments. Our research objective is to implement realistic funds, and we cannot achieve this without controlling who can own shares in our fund. Furthermore, ignoring subsequent usage by other experiments, this work is a scientific contribution in that it successfully implements a system of permissions for an asset on the blockchain and may be used in other contexts.

Chapter 4

Off-Chain Fund Implementation

In this chapter we build out the fund, allowing investors to buy and redeem shares using the Ethereum currency. In doing so, we make a digital representation of a share. We also create code in the form of an abstract contract which we will be used by the third experiment.

4.1 Introduction

In a traditional mutual fund the fund manager is the sole entity providing liquidity. The fund will have a daily period where it consolidates the order book then adjusts ownership by burning shares or minting new ones depending on the net result. For example, if the total shares involved in sell orders exceed the total shares in buy orders then the fund must take some shares out of circulation.

Extending this guaranteed liquidity is problematic. With a very manual process of issuing shares and redeeming shares, the idea of doing this in continuous time seems impossible. Accountants would spend all day reconciling accounts, contacting custodians and moving paper work.

Another option to increase liquidity would be to create some sort of external exchange where the fund can be traded. This would effectively ship the administrative workload from the fund to exchange. This thesis targets mutual funds which unlike ETFs are not traded on exchanges. Therefore, in the aspiration of trying to keep the blockchain emulation as close to life as possible, we do not pursue this avenue. However, to display the full flexibility of the manner in which we build the fund, we will eventually implement a "Swap" contract to display how third parties might use allowances to facilitate external exchanges for customers.

As the first of the two fund implementations, this chapter displays the automation benefits of the blockchain for fund administration via the creation of a so called "NAV order". The implementation of this order allows

buyers to execute orders 24/7, and for sellers offers liquidity intra-day then guarantees it at the close of business. This is done by implementing a queue for sell orders, which is then checked by buy orders so that shares are traded rather than minted where possible. Finally, we provide infrastructure for the fund to close the sell orders at the end of the day. As we will see the digitization of property rights and currency ensures ownership can be accounted for efficiently by computer code. This allows the fund to simply leave the contract to manage itself, periodically stepping in to take money out of it for investing in off-chain assets, or put money into it to reconcile the sell orders daily. This offers a significant improvement over the prevailing workflow.

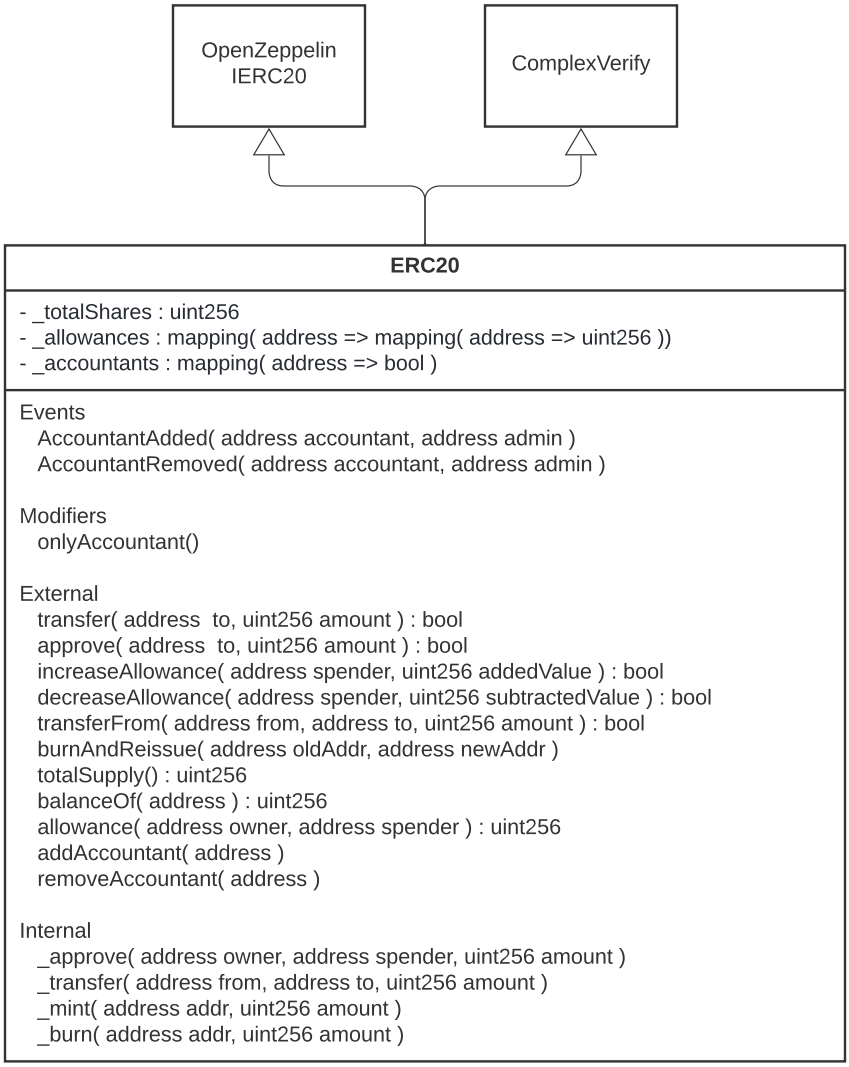
4.2 Precepts

4.2.1 ERC20 Implementation

As mentioned in our literature review, when implementing a token it is wise to use EIP-20 to build the contract. Our ERC20 implementation inherits from the OpenZeppelin ERC20 interface[1], and makes use of code from the OpenZeppelin ERC-20 implementation[2]. The finished code can be found in src/fund/ERC20.sol. The first important addition to this interface that we have added is the "accountant" role. We mentioned in the previous chapter that a separation of responsibility is a principle that can increase the security of the contract, as opposed to giving all members of staff unlimited privileges. The role of the accountant is to manage all processes that relate to money and shares within the fund. Within the ERC20 contract itself we require the function caller to be an accountant for the function "burnAndReissue" (which was a suggested feature from EIP-884). In accordance with the ERC20 standard we emit events to log activity on the blockchain. The UML for the implementation can be found in Figure 4.1.

[1] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol
[2] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

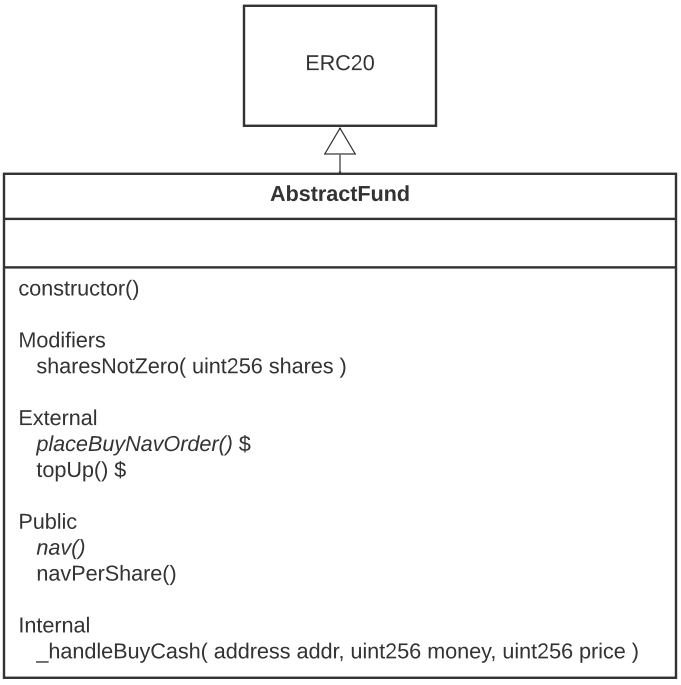
Figure 4.1: ERC20 UML



4.2.2 Abstract Fund

Despite being very different implementations, this experiment and the third experiment both implement funds and it is important to think ahead, imagining what functionality they may share to avoid repetition of code. We therefore implement an abstract contract AbstractFund.sol to define a common interface for both funds where possible, and any other functions which the contracts may share. The UML is provided in Figure 4.2.

Figure 4.2: AbstractFund UML



Both funds will be priced in terms of NAV per share. Therefore we include two public methods; one which returns the NAV and the other NAV per share. The latter is important for transparency and calculating price, the second is a convenience for clients, rather than separately calling the NAV function and the total supply function.

Across both implementations the buy order functionality will work by the client calling some function with a "msg.value", then the implementation will attempt to execute the maximum number of shares the customer can afford, repaying any excess. For example if shares cost £5 and the customer sends £11, they will get two shares (£10) and a £1 refund. As such, no arguments or return values are strictly required, thus we provide an external payable abstract method "placeBuyNavOrder" in AbstractFund. To handle the logic surrounding calculating the shares a customer can afford and performing any refunds, we add an internal utility function "_handleBuyCash" which will be used in both of the funds buy order implementation. The implementation of this function "_handleBuyCash" is detailed in Algorithm 4.1.

Algorithm 4.1 _handleBuyCash in AbstractFund.sol

```
/**
 * @dev Ensures clients are refunded any cash which is not used for buying
 * shares.
 *
 * @param addr The address of the buy order caller.
 * @param money The msg.value sent by the caller, throws for zero or less
 * than price
 * @param price The price at the point of call.
 *
 * @return sharesToExecute The shares the customer can afford to execute.
 */
function _handleBuyCash(address addr, uint256 money, uint256 price)
    internal
    returns (uint256 sharesToExecute)
{
    require(
        money > 0,
        "Fund: msg.value must be greater than zero to place buy order"
    );
    if (money < price) {
        // Refund
        payable(addr).transfer(money);
        // Throw
        require(
            false,
            "Fund: msg.value must be greater than or equal to price"
        );
    }
    sharesToExecute = money / price;
    // Client will spend less than they sent -> refund
    if ((sharesToExecute * price) < money) {
        payable(addr).transfer(
            money - (sharesToExecute * price)
        );
    }
}
```

We cannot define an abstract method for sell orders as the return value will be different; in the on-chain case the order won't be queued. When an order is queued, it is important for the client to receive the details for reference. For example, they may be supplied an ID which they can then use to check the number of shares that have been executed by subsequent buy orders, or delete the order and remove it from the queue.

We can however add a common modifier that is important; "sharesNotZero". Logically clients shouldn't be able to perform zero share sell order operations.

Finally, we include a payable function to top up the fund. In this experiment this will be used by accountants to increase the cash position of the contract in order to close sell orders. In the third experiment this will be used to initially set the share price.

4.2.3 Pricing and NAV Reporting

We price based on NAV per share, therefore the calculation is NAV / total shares. There are no continuous numerical types in Solidity and integer division is truncated to zero. This has the consequence of favouring buyers, because unless the "true" price is itself an integer, the given price will be lower. We do have the option of rounding up, however this would not be wise in the situation where all share holders sell, then price $\times$ shares will be larger than the NAV and the fund may be unable to pay. Additionally, the fund naturally would want to favour buyers, thus we round down.

In this experiment NAV resides off-chain therefore we must complete some process to move the data on-chain. One option would be to use an Oracle[3] which links some API to deliver the NAV to smart contract code. Typically Oracles are implemented with multiple APIs so that if one breaks, the contract can continue to function. Instead of making assumptions behind the API ecosystem owned by funds, we will instead use a simple entry point function "setNav" which is accessible by accountants. Calls to this function could easily be automated.

4.2.4 Sell Orders

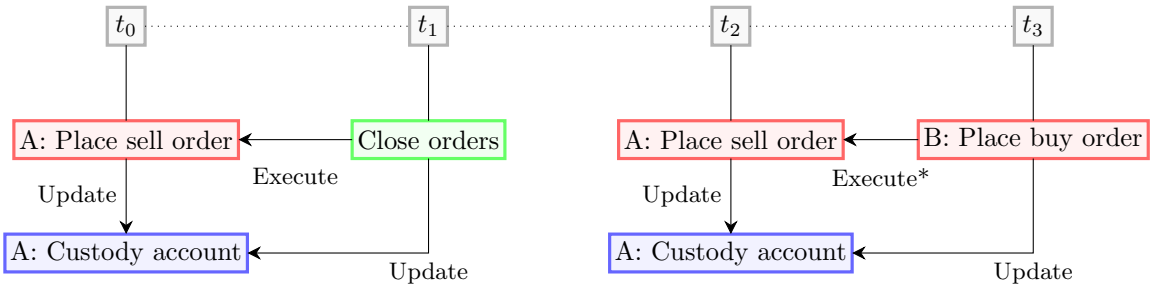
Before implementing the sell order functionality, it is worth considering what it may entail. Sell orders can be executed by buy orders, or the fund can close them out at the end of the day. All sell orders will be queued at first instance, because there are no outstanding buy orders as they are automatically executed. To guarantee that the sell order may eventually be executed, it seems logical for the fund to take custody of those shares until either the order is cancelled or the order is executed. To do this, the fund will reduce the balance of the client ("_balances"), and increase the balance of their custody account ("_custodyAccounts"). Of-course, if they cancel the order we simply reverse this. What this means is that when a buyer or the fund comes across the sell order, no special logic is needed to check the seller still has those shares available. It also gives the seller transparency as to what their true "balance" is.

[3]We refrain from discussing the technical details here, see Chainlink (2022) for further information.

Order Execution Workflow

On a high level we can now visualize what the work flow will look like when it comes to executing orders in Figure 4.3. Different time points are indicated by $\{t_i\}_{i=0}^{i=3}$; red blocks indicate calls to the contract by clients (A and B); green by the fund manager; and finally, blue boxes indicate state.

Figure 4.3: NAV Order Flow Chart



* Potentially partial execution depending on the size of the sell order and the funds supplied by B in comparison to NAV per share at t_4.

4.3 Fund Implementation

4.3.1 Sell Order Queue

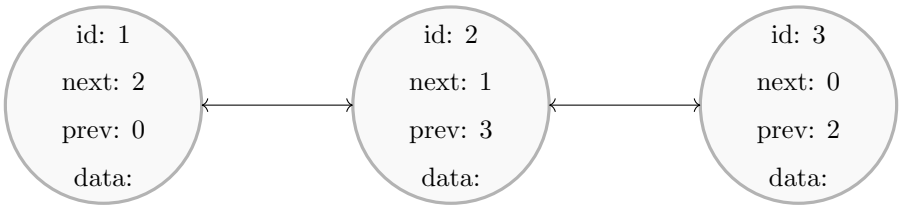
Data Representation

The sell order queue is a queue - we add orders to the back, and execute orders from the front. Deleting values from an array at a location other than the end whilst maintaining the order of entries is an O(N) operation. Insertion and deletion for a doubly linked list is O(1), therefore we implement this data structure to represent the order queue.

For the implementation we first started with an abstract contract, with each node having an ID, a next ID, a previous ID and also a data attribute. Setting data to be a byte array ensured that further implementations could change what was stored (i.e. (address, int) $\rightarrow$ (address, bool, int)) and create their own clients to manage this. The abstract contract also provided several functions that were agnostic to further implementations: for example dequeue will always remove the first entry, regardless of the data encoding.

The nodes themselves were stored in a mapping, with their key being their ID. The reason to also store the ID in the node (and thus duplicate the data) is that the ID in the node served as a null checker; we enforced that a node could never have an ID of value zero. Therefore "nodes[id].id == 0" is a check to see if the node doesn't exist.

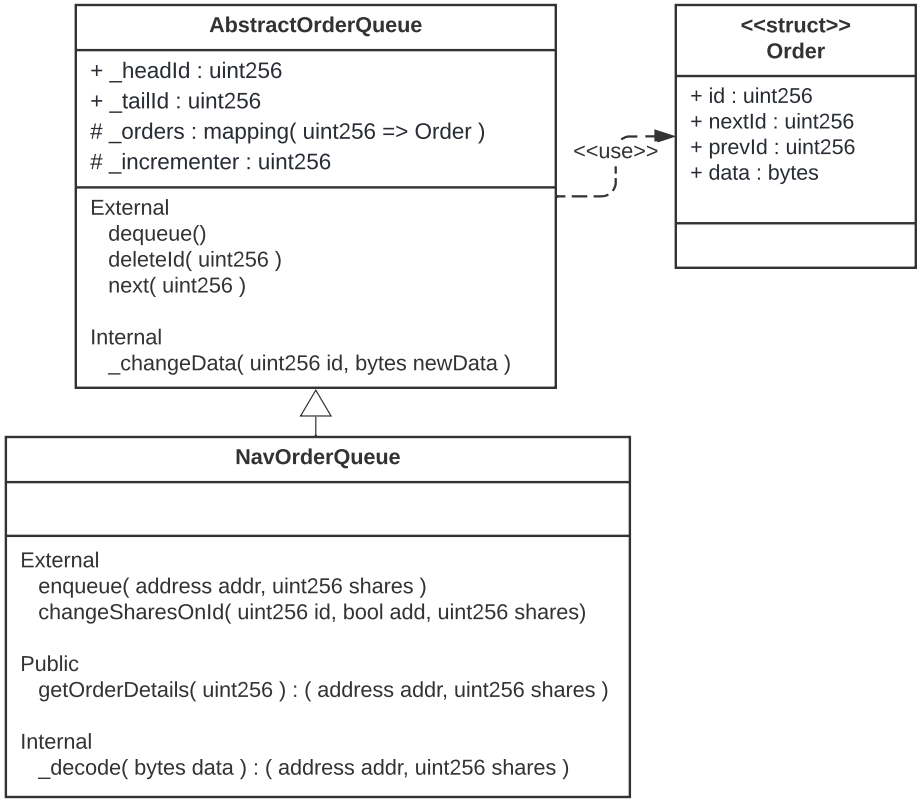
Figure 4.4: Order Queue Visualization



Implementation

After implementing this abstract queue, another file NAVOrderQueue.sol was created which implemented the data format of "(address, shares)" with the address being the address of the order placer (in our case a seller) and shares being the number of shares the order entailed. To decode this we provided an internal function "_decode" which returns the two tuple of the information in its proper format. Another requirement was that we can modify the data in the case of a sell order being actioned but not fully executed. To fulfill this we provide the function "changeSharesOnId". The enqueue implementation was based purely on timing: all new orders were placed at the back of the queue. The input to enqueue was our chosen data format therefore it had two arguments. Finally, we provided an event "OrderQueued" to log the address of the order placer and the ID of that order. This allows clients to reference the logs to check what orders they have outstanding, using this to, for example, delete orders. We provide the UML for the NavOrderQueue.sol contract in Figure 4.5.

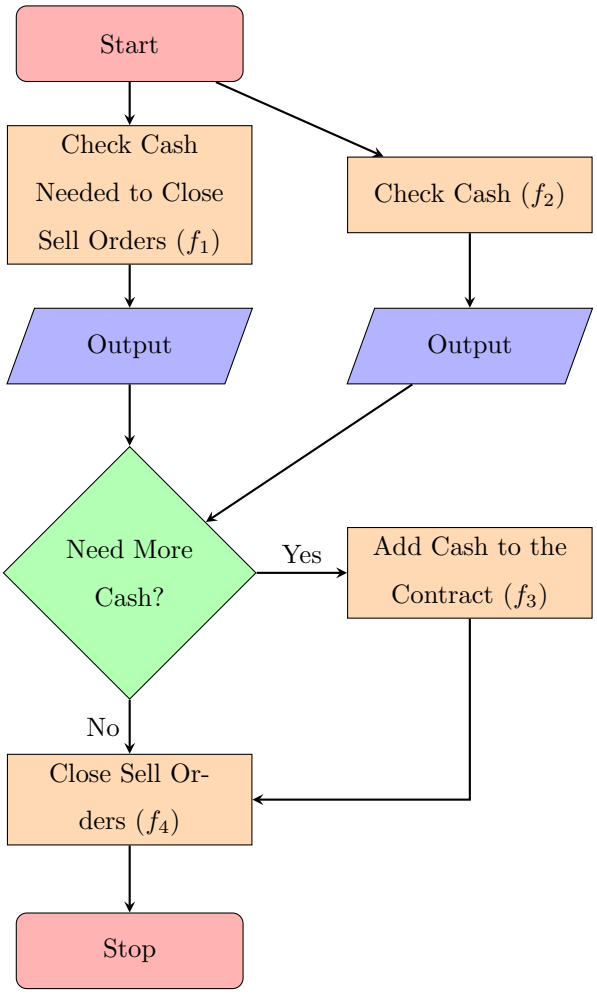
Figure 4.5: NavOrderQueue UML



4.3.2 Closing the Orders

With sell order functionality now in action, we can visualize the process the fund manager will go through each day to close the sell orders in Figure 4.6. All functions that are utilised by this workflow are provided in the footer of the figure, this allows the reader to link the code to the logic. We then provide the code implementation for "closeNavOrders" in Algorithm 4.2.

Figure 4.6: Close Orders Flow Chart



f_1: cashNeededToCloseSellOrders(), f_2: fundCashPosition(), f_3: topUp(), f_4: closeNavOrders()

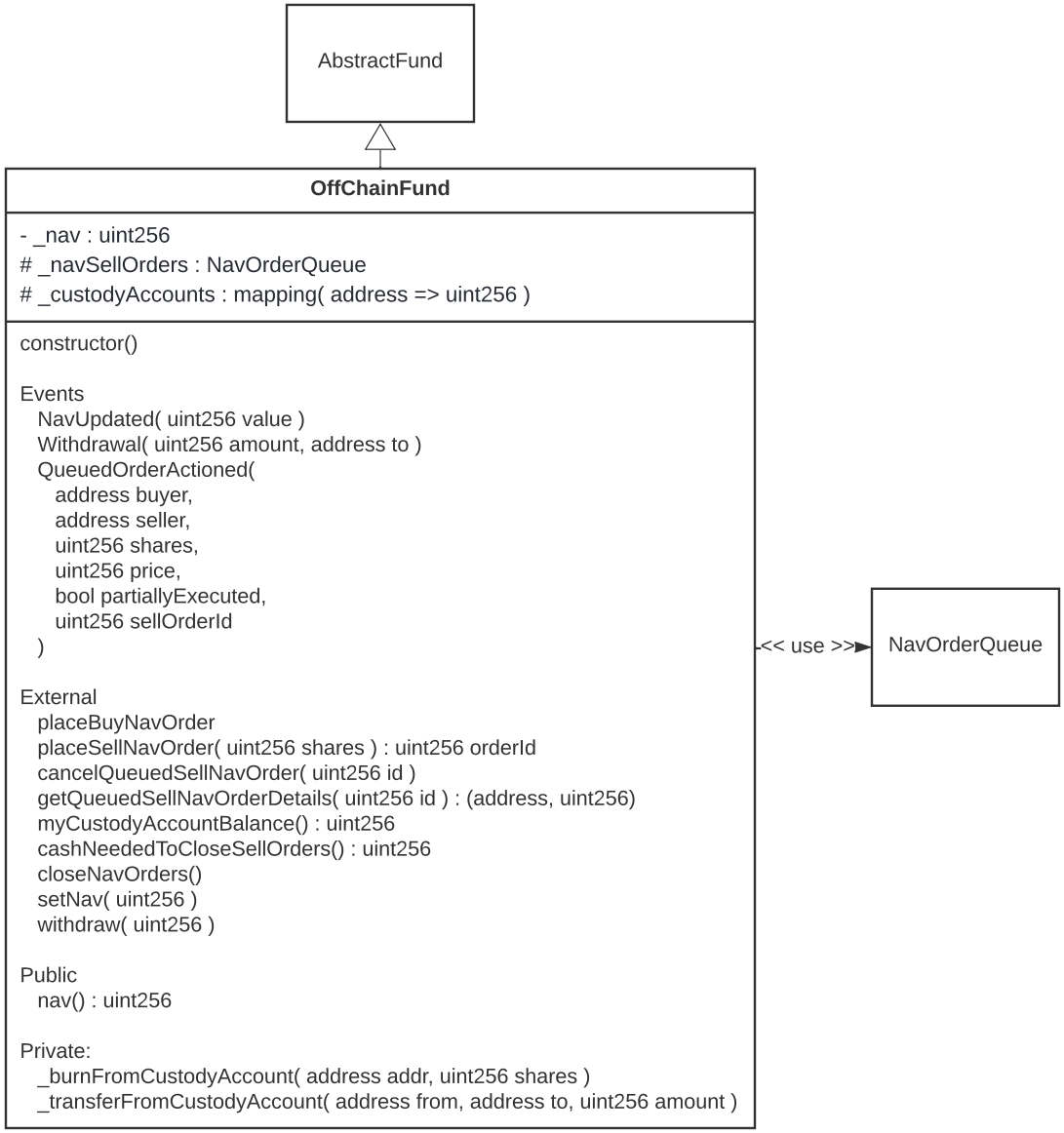
Algorithm 4.2 closeNavOrders in OffChainFund.sol

```
function closeNavOrders()
    external
    override
    onlyAccountant
{
    uint256 price = navPerShare();
    address clientAddr;
    uint256 clientShares;
    uint256 head;

    // Outstanding sell orders
    if (_navSellOrders._headId() != 0) {
        head = _navSellOrders._headId();
        while (head != 0) {
            (clientAddr, clientShares)
                = _navSellOrders.getOrderDetails(head);

            uint256 owedCash = price * clientShares;
            _createCashPosition(owedCash);
            payable(clientAddr).transfer(owedCash);
            _burn({addr : clientAddr, amount : clientShares});

            // Log
            emit QueuedOrderActioned({
                buyer : address(this),
                seller : clientAddr,
                shares : clientShares,
                price : price,
                partiallyExecuted : false,
                sellOrderId : head
            });

            _navSellOrders.dequeue();
            head = _navSellOrders._headId();
        }
    }

    // No sell orders outstanding
    return;
}
```

4.3.3 Final Product

Finally, in Figure 4.7 we can give the UML for the finished product OffChainFund.sol. On top of the previous detailed functionality, we also implement a variety of methods to increase transparency in the fund. The event "NavUpdated" can be used by clients and the fund to see when the last update of NAV was and the value at that time. The "Withdrawal" events logs when accountants take money out of the fund, detailing the amount and who the accountant was. Finally, the "QueuedOrderActioned" event gives sellers transparency as to the status of their queued order. For example, if a buy order results in the partial execution of their sell order, the seller may want to know what NAV per share was at the time (i.e. the price) as well as how many shares were sold. To provide further clarity to clients when it comes to the sell order queue we provide a variety of methods; allowing customers to check their custody account with "myCustodyAccountantBalance" and get the queued order details with "getQueuedSellNavOrderDetails".

Figure 4.7: OffChainFund UML



4.4 Swap Contract

To demonstrate how ERC20 allowances could be used for swaps we implemented a swap contract Swap.sol. The contract code can be found in Algorithm 4.3. In our test (found in /test/fund/Swap.t.sol) we swapped shares of some off-chain fund asset A (owned by customer A) for shares of some other off-chain fund asset B (originally owned by customer B). To interact with the external contracts we used the OpenZeppelin ERC20 interface (IERC20) which was also used in our own ERC20 implementation.

Algorithm 4.3 Swap.sol

```
contract Swap {

    /// ------------------------------
    ///         State
    /// ------------------------------

    address public _assetA;
    address public _assetB;

    constructor (address assetA, address assetB) {
        _assetA = assetA;
        _assetB = assetB;
    }

    /// ------------------------------
    ///         External
    /// ------------------------------

    /**
     * @dev Called when some criteria is met
     */
    function swap(
        address counterpartyA,
        address counterpartyB,
        uint256 amountAssetA,
        uint256 amountAssetB
    )
        external
    {
        IERC20(_assetA).transferFrom({
            from: counterpartyA,
            to: counterpartyB,
            amount: amountAssetA
        });

        IERC20(_assetB).transferFrom({
            from: counterpartyB,
            to: counterpartyA,
            amount: amountAssetB
        });
    }
}
```

Whilst perhaps a trivial example, many things can be built on top of this. A DEX could allow for a huge variety of share swaps between verified clients; swapping equity tokens with tokens of the fund or even the tokens of other fund managers.

4.5 Tests

The work in this chapter amounted to two abstract and four non-abstract contracts:

```
./
└── src
    ├── fund
    │   ├── AbstractFund.sol
    │   ├── ERC20.sol
    │   ├── OffChainFund.sol
    │   └── Swap.sol
    └── order
        ├── AbstractOrderQueue.sol
        └── NavOrderQueue.sol
```

We test all non-abstract contracts and produce the following test files:

```
./
└── test
    ├── fund
    │   ├── ERC20.t.sol
    │   ├── OffChainFund.t.sol
    │   └── Swap.t.sol
    └── order
        └── NavOrderQueue.t.sol
```

To give the user an idea of what these tests look like we now provide and explain several examples.

4.5.1 NavOrderQueue.t.sol

In this file we tested all functionality of the queue. The first test involved testing that enqueue functioned properly, we display this in Algorithm 4.4. This test ensures that the "OrderQueued" event is emitted by the contract upon enqueueing, then checks that all of the pointers in the doubly linked list are set correctly. In the single node case (let's call this node A) the head and tail should both point to A.

Algorithm 4.4 testOrderAdded in NavOrderQueue.t.sol

```
function testOrderAdded() public {
    uint256 orderId;

    vm.expectEmit(
        true, false, false, false,
        address(buyList)
    );
    emit OrderQueued(acc1, 1);
    orderId = buyList.enqueue({addr : acc1, shares : 50});
    assertTrue(orderId == 1, "incorrect index assigned");
    assertTrue(
        buyList._headId() == orderId,
        "Order not inserted at head"
    );
    assertTrue(
        buyList._tailId() == orderId,
        "Single entry should also be tail"
    );
}
```

4.5.2 OffChainFund.t.sol

We can see the process of constructing the fund in the constructor of our test contract (test/fund/OffChainFund.t.sol). This is provided in Algorithm 4.5.

Algorithm 4.5 setUp in OffChainFund.t.sol

```
function setUp() public {
    vm.startPrank(acc1);
    fund = new OffChainFund();
    // Add accountants
    fund.addAccountant(acc1);
    // Set price = 100
    fund.setNav(100);
    // Add verifiers
    fund.addVerifier(acc1);
    fund.addVerified(acc2);
    // Add verified
    fund.addVerified(acc3);
    vm.stopPrank();
}
```

After this set up, we performed a variety of tests, trialling all functionality of the contract. We'll provide two examples of testing. In the first example in Algorithm 4.6, we test the automatic execution of a buy order. We give acc2 (verified) enough money to buy exactly 10 shares of the fund, then call the buy function with value equal to this amount. We then check the shares are added and the supply of the fund has changed.

Algorithm 4.6 testBuyOrder in OffChainFund.t.sol

```
function testBuyOrder() public {
    vm.deal(acc2, 1000);
    vm.prank(acc2);
    // 10 * 100 Wei = 1000
    fund.placeBuyNavOrder{ value : 1000 }();
    assertTrue(
        fund.balanceOf(acc2) == 10,
        "Shares not added to account"
    );
    assertTrue(
        fund.totalSupply() == 11,
        "Supply not changed"
    );
}
```

The second example involves testing the sell order functionality in Algorithm 4.7. In this test we again buy 10 shares with acc2, then immediately place a sell order for those shares. Subsequently ensuring the shares have been moved out of the main account into the custody account, and the total shares of the fund have not changed.

Algorithm 4.7 testSellOrderQueued in OffChainFund.t.sol

```
function testSellOrderQueued() public {
    vm.deal(acc2, 1000);
    vm.startPrank(acc2);
    fund.placeBuyNavOrder{ value : 1000 }();
    fund.placeSellNavOrder(10);
    assertTrue(
        fund.myCustodyAccountBalance() == 10,
        "Funds not added to custody account"
    );
    assertTrue(
        fund.balanceOf(acc2) == 0,
        "Funds not removed from main account"
    );
    assertTrue(
        fund.totalSupply() == 11,
        "(False) change to supply"
    );
    vm.stopPrank();
}
```

4.6 Discussion

After implementing the fund, we can identify some improvements and opportunities for further research. Our implementation did not involve fee structures. We mentioned in our literature review that Ethereum users must send Ether to perform transactions, this is called gas, and gas is higher the more computationally expensive the transaction (Ethereum, 2022b). Viewing our completed contract, it is now apparent that buyers and the fund manager pay for the execution of sell orders because of the involved computation of traversing the sell order queue.

It would seem prudent to create some mechanism through which sellers pay some transaction fee, which is then paid to the buyer; be them another client or the fund itself. It is also quite apparent due to the cost of traversing being O(N) that the cost should be more orientated to a per order cost rather than a per share cost. Lots of small orders being in the queue could cost buyers more than a small number of large orders. This fee for sellers could potentially be partially refunded upon order deletion by the order placer. On top of this, it is apparent that there are associated investment based costs involved in the underlying assets. For example when someone sells shares, the fund may need to sell assets which have a cost, or when someone buys shares, buy them. These fees would need to be somehow incorporated in the fund for the fund manager to recuperate their costs. More research is required into charging fees on such infrastructure.

In regards to NAV, significant infrastructure will be needed to update NAV in real time in order for customers to transact at fair prices. Oracles seem to be a candidate to make this happen, but to implement an Oracle we need at least one, and preferably several, APIs which need to be highly accurate and reliable. If they don't exist, or are not automated, their maintenance will be equivalent to our "setNav" function, and therefore not an improvement at all. We end with a comment from the original article we mentioned in our literature review (Flood, 2022):

> *In an industry that still struggles to understand and supply APIs but instead insists on emailing excel sheets around - why not do baby steps first?*
>
> *How about we first require any fund to implement the same transparency mechanisms as the ARK family of funds are (required to)? This will ultimately result in the data being managed in a way that is actually suited for automation.*
>
> - Financial Times user Mmm42 (2022)

Chapter 5

On-Chain Fund Implementation

In this chapter we investigate contracts which have a hard coded investment strategy. Using such a contract, a fund manager can list funds for which they do not need to manage the investments. We implement this through a contract which invests in other on-chain assets. This implementation represents an important development in the fund management industry which is becoming increasingly automated.

5.1 Introduction

With our off-chain invested fund implemented, the final experiment now involves building a fund that invests in on-chain assets. As stated in the literature review, such an implementation already exists in the form of Enzyme (2022), but for reasons discussed in our literature review, we opt to create our own. By starting from scratch, we reveal a number of themes which make for a lengthy discussion at the end of this chapter.

To take automation to the extreme, we choose to build a fund that has a hard-coded strategy, removing all discretion from the fund manager for the sake of example. We start this experiment with defining some assets which we can invest in, then detailing our investment strategy. Finally, with this new infrastructure in place, we build our buy and sell functionality on top.

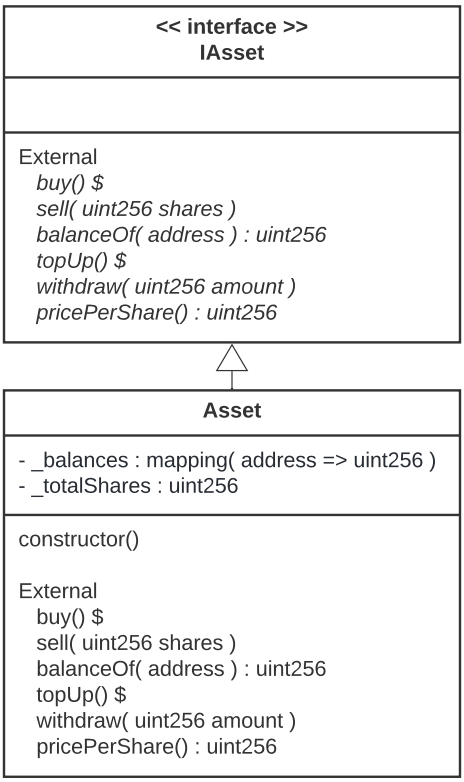
5.2 On-Chain Portfolio

5.2.1 Assets

To create an on-chain portfolio we first need some dummy assets to invest in. We define an interface in src/asset/IAsset.sol to define what is required to allow our contract to function, as-well as provide a way for our fund contract to interface with external asset contracts. Unlike our fund, the focus here is that the asset has the bare minimum functionality to allow the client in our fund to function. This implementation makes no assumptions as to how the asset trades, this is in comparison to our fund, which enforces that trading is only performed by verified customers. The bare bones nature of this implementation means we can focus on the functionality of the fund contract, and avoiding writing large amounts of code to "set up" our asset contracts when testing the fund. The implementation of the asset contract can be found in src/asset/Asset.sol. We provide the UML for the two contracts in Figure 5.1.

The proceeding code will assume all assets the fund manager may want to invest in follow the structure of this Asset.sol. A fundamentally important point to make here is that if interfaces for assets are different, then our code is worthless because the functions to buy or sell will be different. The lack of an ERC20 equivalent for tokenized assets is a significant block; diversity in token interfaces will make such work at worst impossible, and at best extremely convoluted, inefficient and difficult to manage.

Figure 5.1: Asset UML



5.2.2 Investment Strategy

We construct our fund contract with a list of addresses (of assets) and a corresponding list of weights. The constructor is provided in Algorithm 5.1. The weights detail the percentage allocation that the asset should ideally have, for example if a user enters [address(assetA), address(assetB)] and [50, 50] it means that the fund will seek to have a equal allocation portfolio consisting of two assets; A and B.

Algorithm 5.1 Constructor in InvestedFund.sol

```
constructor (address[] memory investments, uint256[] memory weights) {
    require(
        investments.length == weights.length,
        "Fund: length mismatch between investments and weights"
    );
    require(
        _sum(weights) == 100,
        "Fund: weights must add to 100"
    );

    _investments = investments;
    _nInvestments = investments.length;
    _weights = weights;
}
```

The allocation will work for each asset by first performing (NAV × weight) / 100 to decide the maximum cash it could invest in the asset. Then dividing this number by price to determine the maximum number of shares it could afford to buy with this allocation. Depending on the current position (we may have already bought shares), the fund then buys or sells the asset to achieve this figure. This logic is found within the function "_allocate" function (Algorithm 5.2). The logic relies on the funds ability to receive payment via selling assets. To allow for this, we supply a payable "receive" function so that the asset can transfer money to the fund.

Algorithm 5.2 _allocate in InvestedFund.sol

```
/**
 * @dev Invests `amount` according to the weightings. Analyses existing
 * investments to determine what changes need to be made.
 */
function _allocate(uint256 amount) private {
    // Get current allocation
    uint256[] memory actualShares = ownedShares();
    // Decide some new allocation
    uint256[] memory proposedShares = new uint256[](_nInvestments);
    for (uint i = 0; i < _nInvestments; ++i) {
        uint256 targetAmount = (amount * _weights[i]) / 100;
        uint256 price = IAsset(_investments[i]).pricePerShare();
        proposedShares[i] = targetAmount / price;
    }
    // Iterate through each investments
    // Action sell orders on investments that need adjustmenting down
    // Save the buy adjustments to memory for later
    uint256[] memory buyIndices = new uint256[](_nInvestments);
    uint256 buyIndex = 0;
    for (uint256 i = 0; i < _nInvestments; ++i) {
        if (proposedShares[i] < actualShares[i]) {
            IAsset(_investments[i]).sell(
                actualShares[i] - proposedShares[i]
            );
        }
        else if (proposedShares[i] > actualShares[i]) {
            buyIndices[buyIndex] = i;
            buyIndex += 1;
        }
        else {continue;}
    }
    // Do the buy adjustments
    for (uint256 i = 0; i < buyIndex; ++i) {
        // Get the investment of the index
        uint256 index = buyIndices[i];
        uint256 sharesToBuy = proposedShares[index] - actualShares[index];
        uint256 amountToSend =
            IAsset(_investments[index]).pricePerShare() * sharesToBuy;
        IAsset(_investments[index]).buy{ value : amountToSend }();
    }
}
```

A potential additional feature would be to allow an accountant to modify the additional weights via some external function "updateWeights(uint256[] weights)". However, for our example we wanted to have a tamper proof investment strategy as this is the most stark contrast to the typical discretionary model of a mutual fund.

5.2.3 Transparency

We included an event "Rebalance" to record every time the portfolio is re-balanced. This event logs the new shares allocation as well as who called the rebalance function. This creates a paper trail for the fund and gives transparency to customers as to how their money is being managed. Additionally we include the public function "ownedShares" to display at any time what the investment by investment breakdown is in terms of shares owned.

5.3 Order Implementation

With the ability for the fund to buy or sell assets where necessary the fund can now process orders. We will now detail the relatively simple implementations of buy and sell functionality.

5.3.1 Buy Orders

The function for placing a buy order is displayed in 5.3. The difference between the implementation in this contract and the one in our off-chain fund is firstly that we go straight to minting rather than checking sell orders, and we also immediately make use of the new cash by calling "_allocate". In the off-chain fund, the money from buyers simply sits as cash until an accountant withdraws it from the fund. A nuance is that when we call "nav()" after sending a "msg.value", it will include "msg.value" as we have materially increased the cash balance of the contract. However, we are interested in the price pre-transaction and therefore we must remove "msg.value" from NAV to calculate share price.

Algorithm 5.3 placeBuyNavOrder in InvestedFund.sol

```
function placeBuyNavOrder()
    external
    payable
    override
    onlyVerified
{
    uint256 oldNav = nav() - msg.value;
    uint256 price = oldNav / _totalShares;
    uint256 sharesToExecute = _handleBuyCash({
        addr : msg.sender,
        money : msg.value,
        price : price
    });
    _mint({addr : msg.sender, amount : sharesToExecute});
    _allocate(nav());
}
```

5.3.2 Sell Orders

The function for placing a sell order is displayed in Algorithm 5.4. After checking the order involves a non-zero number of shares and the seller has the sufficient balance, we check the implied cash and then ensure this position exists in the contract. To do this we call a function "_createCashPosition", the implementation of which is detailed in Algorithm 5.5. This function uses "_allocate" to shrink the investments where necessary, performing asset sales which have the effect of increasing the cash balance of the fund. This ensures the fund can pay the client selling the shares, and after payment we take the shares out of circulation by burning them.

Algorithm 5.4 placeSellNavOrder in InvestedFund.sol

```
/**
 * @dev Burns shares and pays seller. Ensures there is a sufficient cash
 * position to do so.
 */
function placeSellNavOrder(uint256 shares)
    external
    onlyVerified
    sharesNotZero(shares)
{
    require(
        _balances[msg.sender] >= shares,
        "Fund: insufficient balance to place sell order"
    );
    uint256 price = navPerShare();
    uint256 money = price * shares;
    _createCashPosition(money);
    payable(msg.sender).transfer(money);
    _burn({addr : msg.sender, amount : shares});
}
```

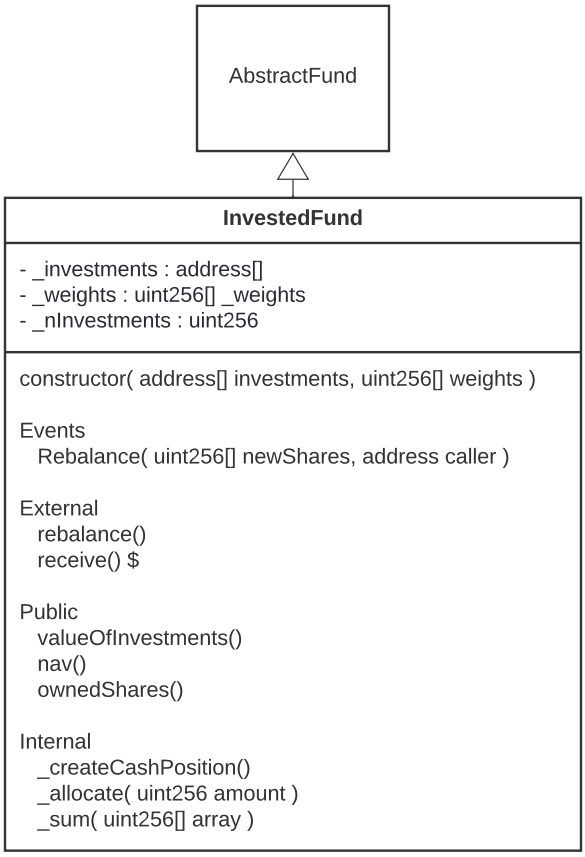
Algorithm 5.5 _createCashPosition in InvestedFund.sol

```
/**
 * @dev Ensures there is `amount` of cash available in the contract.
 * Won't do anything if required position alread exists.
 */
function _createCashPosition(uint256 amount) internal override {
    if (address(this).balance >= amount) {
        return;
    }

    // Free up the cash
    else {
        uint256 requiredCash = amount - address(this).balance;
        uint256 investableCash = valueOfInvestments() - requiredCash;
        _allocate(investableCash);
        return;
    }
}
```

5.4 Final Product

We provide the UML for InvestedFund.sol in Figure 5.2. The reader will notice the simplicity as compared to the UML for the off-chain fund in Figure 4.7. This really goes to show the strong suit of Solidity for dealing with on-chain logic. For our off-chain fund we had to work hard to introduce complicated logic for our fund to be a hybrid of off-chain and on-chain processes.

Figure 5.2: InvestedFund UML



5.5 Tests

The work in this chapter amounted to one interface and two contracts:

```
./
└── src
    ├── fund
    │   └── InvestedFund.sol
    └── asset
        ├── Asset.sol
        └── IAsset.sol
```

We test both contracts and produce the following test files:

```
./
└── test
    ├── fund
    │   └── InvestedFund.t.sol
    └── asset
        └── Asset.t.sol
```

We performed testing of our asset contract in Asset.t.sol, these tests were straight forward and we will leave this to reader to check on the GitHub.

For our tests of the fund, we created a fund which invested in two assets: A, which we initially set to be 25 Wei per share, and B, which we set to be 50 Wei per share. We gave these assets 50/50 weightings and then performed a variety of tests to ensure the portfolio adjusted where necessary. We set up the testing with a NAV of 100 via the "topUp" function. To give an example of testing we display our test for the buying functionality in Algorithm 5.6.

Algorithm 5.6 testBuy in InvestedFund.t.sol

```
function testBuy() public {
    vm.deal(acc2, 100);
    vm.prank(acc2);
    fund.placeBuyNavOrder{ value : 100 }();
    assertTrue(
        fund.balanceOf(acc2) == 1,
        "Balance not added"
    );
    assertTrue(
        fund.navPerShare() == 100,
        "Unexpected price change"
    );
    assertTrue(
        fund.valueOfInvestments() == 200,
        "New cash not invested"
    );
    uint256[] memory ownedShares = fund.ownedShares();
    // 100 / 25 = 4
    assertTrue(
        ownedShares[0] == 4,
        "Asset A not re-allocated"
    );
    // 100 / 50 = 2
    assertTrue(
        ownedShares[1] == 2,
        "Asset B not re-allocated"
    );
}
```

5.6 Discussion

This thesis is not one covering portfolio optimisation or investment strategy. The importance of our work is that we've created a fund with highly automated functionality. Processing buy orders and sell orders is completely automated and clients are guaranteed 24/7 liquidity. The only task the fund manager needs to do is KYC and occasionally rebalance the portfolio if there is no buy or sell activity and the underlying investments have price movements.

It has become quite apparent in this research that the underlying implementation of the assets is of fundamental importance to our work. Reflecting on results in this experiment this author is inclined to agree with the conclusions of Baum (2021). Our "Asset" contract was trivial and whilst was all that was

needed to demonstrate the potential of a contract that had automated investing, clearly represented that without the underlying tokenized assets, the fund is useless. Financial instruments, for example mortgages, oil futures, equities or bonds must first be tokenized in order for this automated investing to take place. Additionally, as mentioned when we built out the interface, in order for funds to exists which invest in these assets, it is paramount that the assets share an interface to make writing the fund code straightforward.

Such findings present the value of our work in the previous chapter; putting the fund administration on the block chain but allowing for off-chain investments. Funds realistically may have to gradually transition to investments in on-chain assets rather than having a purely on-chain fund from the outset. Such "hybrid" solutions would combine the work from both of our experiments, interacting with on-chain assets with potentially automatic re-weightings when fund shares are purchased, and making use of the sell order queue as cash may be outside of the blockchain.

A criticism of the work in this chapter is that our constant re-allocation of the fund every time someone buys or sells may begin to eat up transaction fees in the form of gas, which could result in high costs for clients. Instead of automatically investing the money obtained through buy orders we could hold it as cash which would mean that we could process sell orders without asset sell-offs. The fund manager could then call rebalance at specified intervals to put this cash into investments. This could result in less frequent trading of assets and reduce costs.

Chapter 6

Conclusion

6.1 Summary

Concluding our three experiments, we have demonstrated that KYC workflows can be integrated onto the blockchain, and that a fund can use the blockchain to administer funds consisting of either off-chain or on-chain assets. We now summarize our three experiments.

6.1.1 Know Your Customer

Whilst by far the simplest experiment, the code delivered built on community suggestions (EIP-884) and provided a strong foundation for future chapters. Implementing permissions on a role basis appeared to be a professional and secure way to manage employee and client access to the contract. The most privileged user, represented by the "admin" role, had their tenancy managed in a secure way through a votes based system.

6.1.2 Off-Chain Fund Implementation

This experiment was the most complicated experiment with the biggest codebase. The achievements of the code were to create a fund which for buyers guaranteed liquidity 24/7, and for sellers offered potential liquidity 24/7 and guaranteed liquidity daily. On top of this the fund manager could modulate the cash position of the fund at will for the purposes of investing in off-chain assets and paying sellers for their shares.

As detailed in the discussion section, the implementation delivered was far from a final product, notably the addition of fees, and a more complicated NAV reporting infrastructure put in place. Such improvements needing the input of actual funds, and contingent on their existing capabilities off-chain.

6.1.3 On-Chain Fund Implementation

Our final experiment leveraged all previous experiments to create a fund where the only input required from the fund manager was deciding a list of investments and their weights, then calling a re-balance function to update the portfolio. Whilst a revolutionary and exciting concept, fund managers will have to wait for underlying assets to first be tokenized in order for this to take off. Additionally fund managers and clients alike will have to decide between actively managed or passively managed strategies and whether those strategies are performed entirely on-chain or both; for example, a fund may decide to calculate the weights off-chain using some proprietary algorithm which uses a range of off-chain historical data then upload these weights on-chain.

6.1.4 Achievement of the Research Objective

The main objective for this research was to implement realistic off-chain invested and on-chain invested mutual funds. With the KYC functionality successfully implemented and our funds workflow suitably sophisticated we can conclude that our funds were realistic.

6.2 Contributions

This thesis has provided a model through which mutual funds can move some of, if not eventually all of, their operations onto the blockchain. The automation gains shown by our work are significant. In our off-chain case we saw how the fund manager must simply check and potentially update the cash balance of the contract in order to facilitate infrastructure in which buyers and sellers can place and execute orders 24/7. In our on-chain fund we saw that a fund manager could restrict their focus almost entirely on the investment thesis, with liquidity fully automated.

This work is of significant use to those fund managers who have heard about blockchain, believe in the benefits of adoption, and are interested in applying the technology to their workflows but don't know what the code or architecture would look like. This author hopes that the codebase created by this project can serve as a basis for further implementations by academics or industry practitioners. Finally, this work is also of interest to regulators who may seek to govern these implementations and ensure consumers of blockchain traded funds are protected.

6.3 Further Work

With this work being pioneering, there's lots more to do and many gaps to address. In particular this author identifies three areas for further research: (1) tokenizing underlying assets, (2) standardising fund interfaces, and (3) deciding what to keep off-chain.

6.3.1 Tokenizing Underlying Assets

It was identified in the third experiment that the concept of an on-chain fund was of course severely limited by the presence of on-chain assets. Of interest to us is not only tokenizing these assets, but also distributing the tokens across multiple blockchains. One option is multi-listing, another is creating a DEX that facilitates "atomic swaps" (Thyagarajan et al., 2022): allowing users to exchange tokens from different blockchains. But these DEXs would need to be permitted by the owner of the private blockchain. Therefore, we may see in the future that fund managers create their own or band together to create a DEX which serves as a gateway between say an oil future on the Ethereum main-net and a fund on a private network.

Another point of importance is that these tokens should share a common interface for interaction. Without a shared interface, the automation benefits of funds on the blockchain may be eroded by the sheer volume of code required to manage a fund consisting of heterogeneous on-chain assets.

6.3.2 Standardising Fund Interfaces

Evidence suggests there's interest from fund managers to creating blockchain traded funds (Flood, 2022). During initial forays where more specific tailored workflows are modelled on the blockchain, fund managers should seek to standardise interfaces to avoid confusion between funds and investors. The ERC20 protocol has created significant productivity gains for a range of parties, for example, developers can move companies and not have to learn a whole new interface for a token. Being specific to the traditional financial services sector, we provide the example of the SWIFT protocol. The problem before SWIFT was that there was no standard way to describe transactions, this meant that messages had to be interpreted and executed by the receiver which led to many human errors as well as slower processing times (Scott and Zachariadis, 2012). By creating a standardised format for a transaction, SWIFT has created productivity gains across the financial services industry.

When we look at our final funds products after all the inheritance, it is clear to see that there are a number of ways to name functions that do the same thing: "placeNavSellOrder", "placeSellNavOrder" or "placeNavOrder(sell=true)". If funds each implement their own naming conventions, developers looking to build on top of this code will face a challenge trying to integrate workflows; they will need to have separate code for each fund they want to interface with. To avoid this, this author recommends that joint proof of concepts should be conducted by funds or in the case of a single fund proof of concept that results should be shared across the industry. After this initial work has been done, industry bodies like the Investment Association in the UK can seek to standardise fund interfaces for the benefit of fund managers, clients and third party developers alike.

6.3.3 Deciding What To Keep Off-Chain

Our on-chain fund implementation highlighted the fact the contract has to be called to react to events. If an underlying asset changed price, the fund wouldn't automatically rebalance. Our KYC implementation also assumed that there were processes off-chain to decide whether a customer ought to be verified.

It remains to be decided exactly what work to keep on-chain and what work to keep off-chain. For example, the fund manager could crunch the portfolio weights using some market event driven algorithm, then re-upload them to the contract after some significant event. Such an algorithm could produce better alpha than simply calling the rebalance function at specified intervals.

In addition to deciding what business processes should remain on-chain and off-chain, there is also the decision between what code should live in layer 1 versus layer 2 blockchain. Layer 2 blockchain applications do things such as bundle transactions onto layer 1 to increase efficiency as compared to performing each transaction separately (Ethereum, 2022c). This author omitted a discussion of layer 2 from the literature review and the work throughout the thesis due to the simple fact that layer 2 applications require contracts to function. Our work has been focused on contract development. An investigation into the usage of layer 2 applications to reduce costs versus the layer 1 implementation could prove useful. Additionally, performance testing is needed to see if Ethereum is capable of handling the throughput required by fund issuers. Findings may reveal that layer 2 applications are required to achieve acceptable results.

Perhaps the most important consideration is how to integrate payment. Throughout our work we have used the Ether digital currency as a store of value. However, funds will almost definitely look to stablecoins as a replacement. These currencies need to be developed and as explained in our literature review, regulated. Code also needs to be adjusted to make use of them in smart contracts.

Ultimately, for the future, this author hopes that decentralized applications - whether partially or fully on-chain - become the new norm in a transparent, fair and efficient fund industry.

Bibliography

Amazon. What is ethereum?, 2022. URL https://aws.amazon.com/blockchain/what-is-ethereum/. Accessed: 2022-08-07.

A. Baum. Tokenization—the future of real estate investment? *The Journal of Portfolio Management*, 47 (10):41–61, 2021.

Broadridge. Charting a path to a post-trade utility: how mutualized trade processing can reduce costs and help rebuild global bank roe. Technical report, Broadridge, 2015. URL https://www.broadridge.com/_assets/pdf/broadridge-charting-a-path-to-a-post-trade-utility-white-paper.pdf.

Chainlink. What is a blockchain oracle?, 2022. URL https://chain.link/education/blockchain-oracles. Accessed: 2022-08-07.

F. D'Acunto, N. Prabhala, and A. G. Rossi. The promises and pitfalls of robo-advising. *The Review of Financial Studies*, 32(5):1983–2020, 2019.

S. Ehrlich. Ethereum infrastructure company consensys raises $200 million at $3.2 billion valuation. *Forbes*, 2021a.

S. Ehrlich. After an $850 million controversy, what everyone should know about bitfinex, tether and stablecoins. *Forbes*, 2021b.

Enzyme. Enzyme, 2022. URL https://enzyme.finance/. Accessed: 2022-08-07.

Ethereum. What is ethereum?, 2022a. URL https://ethereum.org/en/what-is-ethereum/. Accessed: 2022-08-07.

Ethereum. Gas and fees, 2022b. URL https://ethereum.org/en/developers/docs/gas/. Accessed: 2022-08-07.

Ethereum. Layer 2, 2022c. URL https://ethereum.org/en/layer-2/. Accessed: 2022-08-07.

C. Flood. Uk fund managers lobby for approval of blockchain-traded funds. *Financial Times*, 2022.

Franklin Templeton. Franklin u.s. government money fund, 2022a. URL https://www.franklintempleton.com/investments/options/money-market-funds/products/4311/A/franklin-u-s-government-money-fund/FMFXX. Accessed: 2022-08-30.

Franklin Templeton. Franklin onchain u.s. government money fund, 2022b. URL https://www.franklintempleton.com/investments/options/money-market-funds/products/29386/SINGLCLASS/franklin-on-chain-u-s-government-money-fund/FOBXX. Accessed: 2022-08-30.

HM Revenue & Customs. *IFM12237*. HM Revenue & Customs, London, United Kingdom, 2019.

HM Treasury. Uk regulatory approach to cryptoassets and stablecoins: consultation and call for evidence, 2022. URL https://www.gov.uk/government/consultations/uk-regulatory-approach-to-cryptoassets-and-stablecoins-consultation-and-call-for-evidence. Accessed: 2022-08-07.

Investment Company Institute. 2022 investment company factbook. Technical report, Investment Company Institute, 2022. URL https://www.icifactbook.org/pdf/2022_factbook_ch1.pdf.

D. Jung, V. Dorner, F. Glaser, and S. Morana. Robo-advisory. *Business & Information Systems Engineering*, 60(1):81–86, 2018.

P. Laurent, T. Chollet, M. Burke, and T. Seers. The tokenization of assets is disrupting the financial industry. are you ready. *Inside magazine*, 19:62–67, 2018.

C. Metz. The rise of the artificially intelligent hedge fund. *Wired*, 2016.

Mmm42. Uk fund managers lobby for approval of blockchain-traded funds — comments section, 2022. URL https://www.ft.com/content/b29b8e67-01ae-41b2-a238-aecfa7be0609?commentID=b098ac48-75d9-4b87-a8bc-6cdfc01faa8e. Accessed: 2022-08-07.

L. Noonan. Blockchain-powered breakthrough on mutual fund. *Financial Times*, 2021.

M. Röscheisen, M. Baldonado, K. Chang, L. Gravano, S. Ketchpel, and A. Paepcke. The stanford infobus and its service layers: Augmenting the internet with higher-level information management protocols. *Digital Libraries in Computer Science: The MeDoc Approach*, pages 213–230, 1998.

D. Sag. Eip-884: Dgcl token. *Ethereum Improvement Proposals*, 884, 2018.

S. V. Scott and M. Zachariadis. Origins and development of swift, 1973–2009. *Business History*, 54(3): 462–482, 2012.

Solidity. Solidity about page, 2022. URL https://soliditylang.org/about/. Accessed: 2022-08-07.

Statista. 10-day ethereum eth/usd realized volatility until may 6, 2022, 2022. URL https://www.statista.com/statistics/1278411/ethereum-price-swings/. Accessed: 2022-08-07.

S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sánchez. Universal atomic swaps: Secure exchange of coins across all blockchains. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1299–1316. IEEE, 2022.

F. Vogelsteller and V. Buterin. Eip-20: Token standard. *Ethereum Improvement Proposals*, 20, 2015.

Vyper. Vyper documentation, 2022. URL https://vyper.readthedocs.io/en/stable/. Accessed: 2022-08-07.